Estructuras de datos

1. **Tipos abstractos de datos y algoritmia**
2. Listas, pilas, colas y conjuntos
3. Árboles
4. Grafos

Universidad Rey Juan Carlos
www.urjc.es

Grado en Ingeniería de Computadores
Curso 2021/2022

Prof. Dr. Carlos Grima Izquierdo
www.carlosgrima.com

v1

Introducción I

- Algoritmo
  - Método o proceso para resolver un problema (generalmente iterativo o recursivo)
  - Conjunto de reglas para efectuar algún cálculo
    - Las reglas deben ser objetivas (no subjetivas) y claras
    - Bien sea a mano o, más frecuentemente, en una máquina
    - Para nosotros y por simplificar: algoritmo es sinónimo de programa
- Algoritmia
  - Ciencia que estudia el concepto, diseño y construcción de algoritmos de forma eficiente

Introducción II

- Estructura de datos
  - Conjunto de datos, relacionados entre sí, que tienen un propósito y uso común
    - Un “struct” en C es una estructura de datos
  - Una estructura de datos suele venir acompañada por varios algoritmos para manipularla
- La descripción formal (matemática, sin programarla) de una Estructura de Datos se llama “Tipo Abstracto de Datos” (TAD)
  - En este curso no vemos la descripción matemática de TADs
- Una “clase” es la programación de un TAD
  - Para programar una “clase”, necesitamos un lenguaje de programación que sea “orientado a objetos”. Ej: C++, Java
  - Una clase es una estructura de datos (conjunto de sus atributos) más los algoritmos para manipularla (conjunto de métodos)

Visual Studio C++ I

- En este curso vamos a utilizar C++ como lenguaje de programación
  - C++ es una extensión de C: tiene la misma sintaxis y funciones de C
  - C++, además, incorpora clases y todo lo necesario para programar con "orientación a objetos"
  - Al igual que C, permite un control total de la memoria mediante punteros
  - La mejor página de referencia es www.cplusplus.com, en donde podemos encontrar la descripción de todas las funciones predefinidas de C y C++, tutoriales, sintaxis, artículos, foro, etc.
  - En este curso se asume que el alumno ya sabe Lenguaje C. Si no es así, se puede aprender C++ fácilmente desde cero (sin saber previamente C) en el siguiente tutorial gratuito de internet: http://c.conclase.net/curso

Visual Studio C++ II

- Un programa sencillo de C++ podría compilarse y ejecutarse online con http://cpp.sh o similares
- Si ya queremos programar profesionalmente, necesitamos un entorno completo, como puede ser Microsoft Visual Studio
  - Sirve para muchos lenguajes, no sólo para C++
  - La edición gratuita se llama “Community” y se descarga desde la web oficial de Visual Studio: https://visualstudio.microsoft.com
  - Al instalarlo, nos pedirá qué lenguaje queremos usar. Seleccionamos únicamente “Desarrollo para el escritorio con C++”
  - Al iniciar por primera vez, nos preguntará por la configuración de desarrollo. Es muy importante elegir aquí “Visual C++”, no “General”

Visual Studio C++ III

- Manejo básico de Visual Studio C++:
  - Los proyectos están contenidos en “soluciones”. Al crear un proyecto, se crea también una solución que lo contiene
  - Crear proyecto: menú archivo, nuevo, proyecto, proyecto vacío, marcar “colocar la solución y el proyecto en el mismo directorio”
    - Los archivos fuentes (.cpp en C++) van en “archivos de origen” (botón derecho, agregar nuevo elemento)
    - Los archivos de cabecera (.h en C y C++) van en “archivos de encabezado”
  - Llevar un proyecto a otro ordenador. Botón derecho sobre el nombre del proyecto, “abrir carpeta en el explorador de archivos” y llevarse toda la carpeta
    - Podemos borrar las subcarpetas “x64” y “.vs” porque únicamente contienen el historial de compilaciones
  - Abrir un proyecto ya existente en Visual C++. Hacer doble click en el archivo “.sln” de dicha carpeta (también menú, archivo, abrir proyecto o solución)
  - Para cerrar un proyecto y su solución, menú archivo, cerrar solución

Visual Studio C++ IV

- Manejo básico de Visual Studio C++ (continuación):
  - Para compilar y ejecutar: Control+F5 o bien menú depurar, iniciar sin depurar
  - Para ejecutar línea a línea (depuración): menú depurar, ejecutar paso a paso por instrucciones (F11)
  - Los errores de compilación aparecen abajo
  - Ejemplo guiado: vamos a crear un proyecto nuevo, de tipo “aplicación de consola”, para ver y explicar un “Hola Mundo” básico en C++
    - Aprovechamos para explicar la salida en C++ con el flujo “cout”
    - Si queremos usar "printf", hay que saber que todas las bibliotecas .h que teníamos en C ahora en C++ se llaman igual pero empezando por "c" y sin el ".h"
      - Ejemplo: stdio.h ahora se llama "cstdio". Por lo tanto habrá que poner #include "cstdio" o #include <cstdio> si queremos usar printf

Eficiencia de los algoritmos I

- Eficiencia de un algoritmo
  - Un algoritmo es más eficiente cuanto menos recursos (tiempo y memoria) emplee para la resolución de su tarea, en relación al tamaño de sus parámetros de entrada
    - Ejemplo: ¿qué es más eficiente en tiempo? Dicho de otra manera: ¿cuál de los dos consigue ordenar una lista más grande en el mismo tiempo?
      - Un algoritmo que ordena una lista de 500 números en 1000 minutos. 500 números /1000 minutos = 0,5 números/minuto
      - Un algoritmo que ordena una lista de 50 números en 200 minutos. 50 números / 200 minutos = 0,25 números/minuto

Eficiencia de los algoritmos II

- El tiempo de ejecución de un programa depende de:
  - Lo bien hecho que esté el algoritmo base del programa (la eficiencia del algoritmo)
  - Potencia de hardware
  - El tamaño de los datos de entrada
    - Ej: número de componentes de un array que queremos ordenar
  - La complejidad de los datos de entrada
    - No tardamos lo mismo en ordenar un vector que ya está ordenado (caso mejor), que uno que está ordenado al revés (caso peor), o que uno que está ordenado aleatoriamente (caso medio)

Eficiencia de los algoritmos III

- En cuanto a la complejidad de la entrada:
  - Caso peor
    - Si no nos dicen nada, calculamos la eficiencia en el caso peor
    - Lo normal en cualquier ingeniería es ponernos en el caso peor
  - Caso medio
    - Habría que calcular la media ponderada de todas las posibles entradas
    - Se necesita, por lo tanto, saber la probabilidad de ocurrencia de cada una, información que no siempre se tiene

Eficiencia de los algoritmos IV

- ¿La memoria es importante?
  - La eficiencia puede ser en tiempo o en memoria
  - Actualmente nos importa mucho más el tiempo que la memoria porque:
    - La memoria es barata, hay mucha, y podemos conseguir más con facilidad.
    - El tiempo es escaso y es el que es (no podemos hacer que un día tenga más de 24 horas)
    - Podemos conseguir más memoria comprándola y juntándola con la que ya teníamos, pero no ocurre lo mismo con el tiempo
      - Si juntamos dos pastillas de memoria de 1 giga cada una, tenemos en total 2 gigas
      - Si juntamos dos ordenadores con 1 gigaherzio de velocidad cada uno, no tenemos un ordenador que vaya a 2 gigaherzios

Eficiencia de los algoritmos V

- ¿Cómo decidir el mejor algoritmo para un problema? Para un mismo problema es posible que haya disponibles varios algoritmos para resolverlo… ¿cuál elegimos?
  - Si solamente tenemos que resolver uno o dos casos pequeños de un problema más bien sencillo, o bien el programa se va a ejecutar pocas veces, podríamos seleccionar el algoritmo más sencillo de programar, o aquél para el que ya exista un programa que podamos comprar e incorporar
  - Pero si tenemos que resolver muchos casos, o el problema es difícil, o el tamaño de la entrada puede llegar a ser muy grande, tendremos que seleccionar el algoritmo de forma más cuidadosa, generalmente el más eficiente
  - Si reducir tiempo y reducir memoria entran en contradicción, elegiremos de forma general el reducir tiempo a costa de aumentar la memoria

Enfoques de decisión I

- Enfoque empírico o «a posteriori»
  - Consiste en programar en la misma máquina los diferentes algoritmos candidatos y elegir el que menos tiempo o memoria gaste (EjercicioPosteriori)
  - No es un enfoque muy adecuado porque:
    - Requiere un esfuerzo de programación
      - Lo ideal sería que pudiéramos elegir sólo con el pseudocódigo, sin necesidad de programar nada
    - La ejecución de los algoritmos puede durar mucho
    - Incluso la misma máquina nos puede dar distintos resultados según su carga actual, si estamos en un entorno multitarea

Enfoques de decisión II

- Enfoque teórico (a priori)
  - Determinar matemáticamente la cantidad de recursos necesarios para el algoritmo como función del tamaño de los casos de entrada
  - La ventaja es que no depende de la computadora que se use, ni del lenguaje de programación, ni de las habilidades del programador
  - A partir de ahora nos centraremos en este enfoque, con una visión simplificada
    - La visión adecuada y completa se verá en otra asignatura

Cálculo de T(n) I

- T(n) será la “función de tiempo” de un algoritmo. Nos estima el tiempo de ejecución del algoritmo, sin necesidad de programarlo o ejecutarlo
  - Con una entrada de tamaño n, asumiendo que n es muy grande (que es lo que nos interesa)
  - En el caso peor en cuanto a la complejidad de la entrada
  - Es un cálculo “a priori”. No es necesario programar ni ejecutar el algoritmo.
  - Por lo anterior, el tiempo no será “exacto”, ni podrá ser medido en segundos...
    - Será medido en número de operaciones elementales que tiene el algoritmo
    - Por tanto un algoritmo de una sola operación elemental tendría $T(n) = 1$
- $T(n) > 0$ para todo valor de n, ya que un algoritmo no puede tardar un tiempo negativo en ejecutarse (no puede tener 0 operaciones o menos)
- T(n) siempre va a ser una suma de términos
  - Ej: un algoritmo concreto tiene una función de tiempo de $T(n)=8n^3+3n^2+\log(n)$

Cálculo de T(n) II

- Una “operación elemental” es aquélla operación cuyo tiempo de ejecución no depende de “n”
- Las operaciones elementales suelen ser:
  - Sumas, restas, multiplicaciones, divisiones, módulo, operaciones booleanas, comparaciones, asignaciones a variables o a posiciones de un array, leer de una variable o posición de un array, saltos en el código, llamadas a funciones, terminación de funciones, declaraciones, retorno (return) de una función, reservas de trozos de memoria de cualquier tamaño (malloc, new), liberación de memoria reservada…

Cálculo de T(n) III

- Ejemplo: vamos a calcular el T(n) de la “funcion5()”
  - Para ello tendremos que contar el número de operaciones elementales (OE) en el peor caso, en función de “n”, y suponiendo que “n” es muy grande
  - Sabemos los T(n) de las 4 funciones a las que llamamos, que son:
    - funcion1(): $T(n) = 5n^2+1$
    - funcion2(): $T(n) = 2n+10\log n+1$
    - funcion3(): $T(n) = 7$
    - funcion4(): $T(n) = n+1$

```
void funcion5 (int n) {
    int i = 5 + 10;
    int j;
    int a, b = 3;
    funcion1(n);
    funcion2(n);
    funcion3(n);
    a = i * funcion4(n);
    if (funcion4(n) == a * b) {
        funcion2(n);
        funcion3(n);
        int z = 10 * a + b;
    }
    else {
        funcion1(n);
        int i = 5 + 10;
    }
    i = 0;
    while (i < n) {
        funcion2(n);
        funcion1(n);
        i++;
    }
}
```

Cálculo de T(n) IV

- En las primeras 8 líneas, veamos cuántas OE hay en cada línea, para luego sumarlas:
  1. En la línea 1 hay 2 OE. Una es por la declaración del parámetro formal. La otra es por el salto de código desde el sitio en donde se llama a la función (ej: el main) hasta el comienzo de la función.
  2. Hay 3 OE (declaración, asignación y suma)
  3. Hay 1 OE (la declaración)
  4. Hay 3 OE (declaración, declaración y asignación)
  5. Hay $5n^2+2$ ($5n^2+1$ son las OE que nos han dicho que tiene funcion1(), más la asignación del valor actual de n en el parámetro formal que tenga funcion1)
  6. Hay 2n+10logn+2 OE (2n+10logn+1 son las que nos han dicho que tiene funcion2, más la asignación del valor actual de n en el parámetro formal que tenga funcion2)
  7. Hay 8 OE (funcion3 nos han dicho que tiene siempre 7 OE, más la asignación del valor actual de n en el parámetro formal que tenga funcion3)
  8. Hay n+4 OE (funcion4 nos han dicho que tiene n+1, más la asignación del valor actual de n en el parámetro formal que tenga funcion4, más la multiplicación, más la asignación de "a")
- Por tanto, sumando, tenemos un total de: $2+3+1+3+5n^2+2+2n+10\log n+2+8+n+4 = 5n^2+3n+10\log n+25$ operaciones elementales en esta porción de código

```
1 void funcion5 (int n) {
2     int i = 5 + 10;
3     int j;
4     int a, b = 3;
5     funcion1(n);
6     funcion2(n);
7     funcion3(n);
8     a = i * funcion4(n);
```

Cálculo de T(n) V

- La siguiente porción de código es un if-else
  - Para calcularlo, tendremos que contar las OE de cada posible alternativa y quedarnos con las que tenga más OE (ya que asumimos que estamos en el peor caso). Si estuviéramos en un switch-case, habría más de 2 alternativas
  - También hay que contar las OE de la evaluación de la condición, y sumarlas a las OE de la peor alternativa
  - Si la peor alternativa no es la primera, tendremos que hacer un salto en el código (desde la condición hasta la peor alternativa), y eso es una operación elemental más
- La evaluación de la condición tiene n+4 OE:
  - La funcion4 nos han dicho que tiene n+1 OE
  - 1 OE más por la asignación del valor actual de n al parámetro formal de funcion4
  - 1 OE más por la comparación
  - 1 OE más por la multiplicación
- Veamos cuántas OE tiene cada una de las 2 alternativas:
  - La primera alternativa (la que se ejecuta si se cumple la condición) tiene un total de 2n+10logn+15 OE:
    - 2n+10logn+2 OE en la línea 10
    - 8 OE en la línea 11
    - 4 OE en la línea 12
    - 1 OE para saltar desde la línea 13 hasta la línea 18
  - La segunda alternativa (la que se ejecuta si no se cumple la condición) tiene un total de $5n^2+6$ OE
    - 1 OE para saltar desde la línea 9 hasta la 14
    - $5n^2$ +2 OE en la línea 15
    - 3 OE en la línea 16
- Si n es muy grande, la peor alternativa (la que tiene más OE) es la segunda, pues $5n^2+6 > 2n+10logn+15$ cuando "n" es suficientemente grande
  - Podemos pintar ambas funciones en www.wolframalpha.com para comprobarlo. Luego veremos cómo se hace
- Sumando, tenemos un total de $n+4+5n^2+6 = 5n^2+n+10$

```
9   if (funcion4(n) == a * b) {
10      funcion2(n);
11      funcion3(n);
12      int z = 10 * a + b;
13  }
14  else {
15      funcion1(n);
16      int i = 5 + 10;
17  }
```

Cálculo de T(n) VI

- La siguiente porción de código es una instrucción y un bucle
  - En la línea 18 tenemos 1 OE (asignación)
  - En el bucle lo primero hay que averiguar cuántas iteraciones se ejecutan en el peor caso, en función de n.
    - En este ejemplo siempre se van a ejecutar “n” iteraciones (desde i=0 hasta i=n-1, ambos límites inclusive)
  - ¿Cuántas OE hay por cada iteración? En cada iteración tenemos:
    - 2n+10logn+2 OE en la línea 20
    - $5n^2$ +2 OE en la línea 21
    - 2 OE en la línea 22 (asignación y suma, porque equivale a i=i+1)
    - 1 OE en la línea 23, para saltar de nuevo hasta la línea 19 (para volver a comprobar la condición)
    - La suma de OE por cada iteración realizada es $5n^2+2n+10\log n+7$
  - Antes de ejecutar cada iteración, tenemos que comprobar si la condición se cumple. La condición tiene 1 OE (una comparación), por lo tanto hay que añadir 1 OE a cada iteración que realizamos.
    - Por tanto, contando la evaluación de cada condición, cada iteración tiene un total de $5n^2+2n+10\log n+8$ OE
  - Para calcular el número de OE que tiene todo el bucle, hay que multiplicar el número de OE de cada iteración por el número de iteraciones que vamos a tener:
    - Tenemos un total de: $n * (5n^2+2n+10\log n+8) = 5n^3+2n^2+10n\log n+8n$ OE
  - Finalmente tenemos que comprobar de nuevo la condición, y ésta ya no se cumplirá.
    - Comprobar por última vez la condición es 1 OE
    - Al ver que no se cumple, tenemos que saltar desde la línea 19 hasta la línea 24, por lo tanto hay que sumar, por el salto, 1 OE más
  - En la línea 24, funcion5() se acaba y por lo tanto tenemos que saltar al lugar en donde se llamó a esta función (ej: el main). Por lo tanto hay que añadir 1 OE más.
- Por lo tanto, sumando, entre las líneas 18 y 24 tenemos un total de: $5n^3+2n^2+10n\log n+8n+4$ OE

```
18      i = 0;
19      while (i < n) {
20          funcion2(n);
21          funcion1(n);
22          i++;
23      }
24  }
```

Cálculo de T(n) VII

- El T(n) de todo el código, por lo tanto, es:
  - De la línea 1 a la 8 (ambas inclusive), $5n^2+3n+10\log n+25$ OE
  - De la línea 9 a la 17 (ambas inclusive), $5n^2+n+10$
  - De la línea 18 a la 24 (ambas inclusive), $5n^3+2n^2+10n\log n+8n+4$

```
1  void funcion5 (int n) {
2      int i = 5 + 10;
3      int j;
4      int a, b = 3;
5      funcion1(n);
6      funcion2(n);
7      funcion3(n);
8      a = i * funcion4(n);
9      if (funcion4(n) == a * b) {
10         funcion2(n);
11         funcion3(n);
12         int z = 10 * a + b;
13     }
14     else {
15         funcion1(n);
16         int i = 5 + 10;
17     }
18     i = 0;
19     while (i < n) {
20         funcion2(n);
21         funcion1(n);
22         i++;
23     }
24 }
```

- Sumando todo, tenemos, para todo el algoritmo, un $T(n) = 5n^3+12n^2+10n\log n+12n+10\log n+39$

Orden de un algoritmo I

- ¿Qué T(n) son más grandes (y por lo tanto peores) cuando “n” es grande?
  - A continuación se muestra una lista, ordenada de los T(n) más rápidos a los más lentos
  - “c” es una constante cualquiera
  - Para ver fácilmente si una función es mayor que otra, podemos pintar ambas a la vez en la web www.wolframalpha.com
    - Ej: queremos comparar n·logn con n^2+1, sólo con “n” positiva y hasta que “n” valga 2. Lo hacemos con: `“plot nlog(n), n^2+1 from n=0 to n=2”` (sin las comillas)

Tipo de T(n)	T(n)
Constante	T(n) = c
Logarítmico	T(n) = log(n)
Lineal	T(n) es un polinomio de grado 1
Cuadráticos	T(n) es un polinomio de grado 2
Cúbicos	T(n) es un polinomio de grado 3
Polinómico	T(n) es un polinomio de grado k
Exponencial	$T(n) = c^n$
Factorial	T(n) = n!
Doblemente exponencial	$T(n) = n^n$

Orden de un algoritmo II

- El orden de un algoritmo es:
  - Su consumo "a priori" de tiempo o de memoria adicional
  - En función del tamaño de los operandos
  - Poniéndonos en el caso peor
  - Cuando n tiende a infinito
    - Cuando el tamaño de los operandos es pequeño, hemos dicho que nos da igual escoger un buen o mal algoritmo porque va a tardar muy poco en ejecutarse
- Calcular el orden de un algoritmo es nuestro verdadero objetivo. T(n) sólo nos sirve como paso intermedio en nuestra visión simplificada de esta asignatura

Orden de un algoritmo III

- Asumiendo que la función T(n) es una suma de términos, el “orden” de T(n) es su término mayor (según la tabla anterior), sin su constante multiplicativa (si la tiene)
  - Ej: si tenemos $T(n) = \log(n)+3n^2+8n^3$, decimos que es de orden n^3 (orden polinómico de grado 3)
  - Ej: si tenemos $T(n) = \log(n)+3n^2+8n^3+2^{2n}$, decimos que es de orden 4^n (orden exponencial)
- Esto es así porque asintóticamente (es decir, cuando n tiende a infinito o es muy grande) el resto de términos y las constantes multiplicativas son despreciables
  - Como nuestro objetivo es saber el orden de T(n), y no la T(n) en sí misma, no es importante que calculemos exactamente las constantes que aparecen en el polinomio T(n), pues al final las vamos a despreciar
  - Y, ante un pseudocódigo no programado, tampoco podríamos calcular esas constantes exactamente, pues no podemos estar seguros de qué operaciones son elementales y cuáles no lo son (dependerá del hardware, y eso no lo sabemos en un pseudocódigo)

Orden de un algoritmo IV

- Si el orden de una T(n) de un algoritmo es una función g(n), decimos que $T(n) \in O(g(n))$
  - Decimos que el algoritmo es de orden temporal O(g(n))
  - También podemos decir que el algoritmo tiene una complejidad computacional en tiempo de O(g(n))
  - O, más abreviadamente, podemos decir que la complejidad temporal del algoritmo es O(g(n))
  - O(g(n)), matemáticamente, es el conjunto de funciones cuyo orden es g(n). Por eso decimos que $T(n) \in O(g(n))$
- Ejemplos:
  - Ej: si tenemos un algoritmo con $T(n) = \log(n)+3n^2+8n^3+1$, decimos que su complejidad temporal es $O(n^3)$
  - Ej: si tenemos un algoritmo con $T(n) = \log(n)+3n^2-8n^3+8\cdot 2^{2n}+1$, decimos que su complejidad temporal es $O(4^n)$
  - Ej: si tenemos un algoritmo con T(n) = c (cualquier constante), entonces decimos que su complejidad temporal es O(1)

Ejemplos de orden temporal

- Cálculo de determinantes
  - Basado en el teorema de Laplace: O(n!)
  - Por eliminación de Gauss-Jordan: $O(n^3)$ (muchísimo mejor)
- Ordenación:
  - Algoritmo de selección: $O(n^2)$
  - Algoritmo QuickSort: $O(n \cdot \log n)$ (mucho más rápido que el de selección porque la función nlogn es menor que n^2 cuando n es grande)
  - Si los componentes del vector a ordenar están acotados: O(n)
- Búsqueda:
  - Búsqueda secuencial en un vector: O(n)
  - Búsqueda binaria en un vector ordenado: O(logn) (mucho mejor que la búsqueda secuencial)
  - Búsqueda en una tabla hash poco ocupada: O(1)
- Operaciones matemáticas:
  - Multiplicar dos números muy grandes (de “m” y “n” cifras respectivamente): $O(m \cdot n)$
  - Sucesión de Fibonacci (0,1,1,2,3,5,8,13,21…): O(i) si se quiere calcular el elemento i-ésimo
- Transformada de Fourier (típico algoritmo que se usa constantemente para telecomunicaciones e inteligencia artificial):
  - Algoritmo clásico: $O(n^2)$
  - Transformada rápida de Fourier: $O(n \cdot \log n)$
  - En algunos casos particulares, se llega a la increíble O(n/logn)
- Hacer el EjercicioComplejidad1.

Complejidad espacial

- Hasta ahora hemos calculado la complejidad temporal de un algoritmo
  - Es decir, el tiempo que tarda, en el peor caso, cuando el tamaño de su entrada tiende a infinito
- Pero no podemos olvidar el otro gran recurso (aunque sea menos importante que el tiempo) por el cual se mide la eficiencia de los algoritmos: la memoria que se usa
- De manera similar a como hemos hecho con la complejidad temporal, podríamos calcular la espacial
  - Pero no lo vemos en este curso

Definición de recursividad I

- Un algoritmo es recursivo cuando el propio algoritmo se llama a sí mismo en algún punto (o varios puntos) de su código
  - Esa llamada se llama “llamada recursiva”
- La recursividad es una forma más natural (que la iteratividad) de ver muchos algoritmos
  - Algunos algoritmos, no obstante, no se pueden hacer recursivos

Definición de recursividad II

- En general:
  - Diremos que un objeto es recursivo si forma parte o se define a si mismo
    - Ej: animaciones o imágenes recursivas

Esquema básico I

- Un algoritmo recursivo siempre sigue un esquema similar:
  - Una condición «si...»
  - Salida del algoritmo en el caso trivial
    - El “caso trivial” es cuando ya no necesitamos hacer una llamada recursiva para resolver el problema
  - En el caso no trivial, se «simplifica» el problema y se llama de nuevo al algoritmo con el problema simplificado como entrada

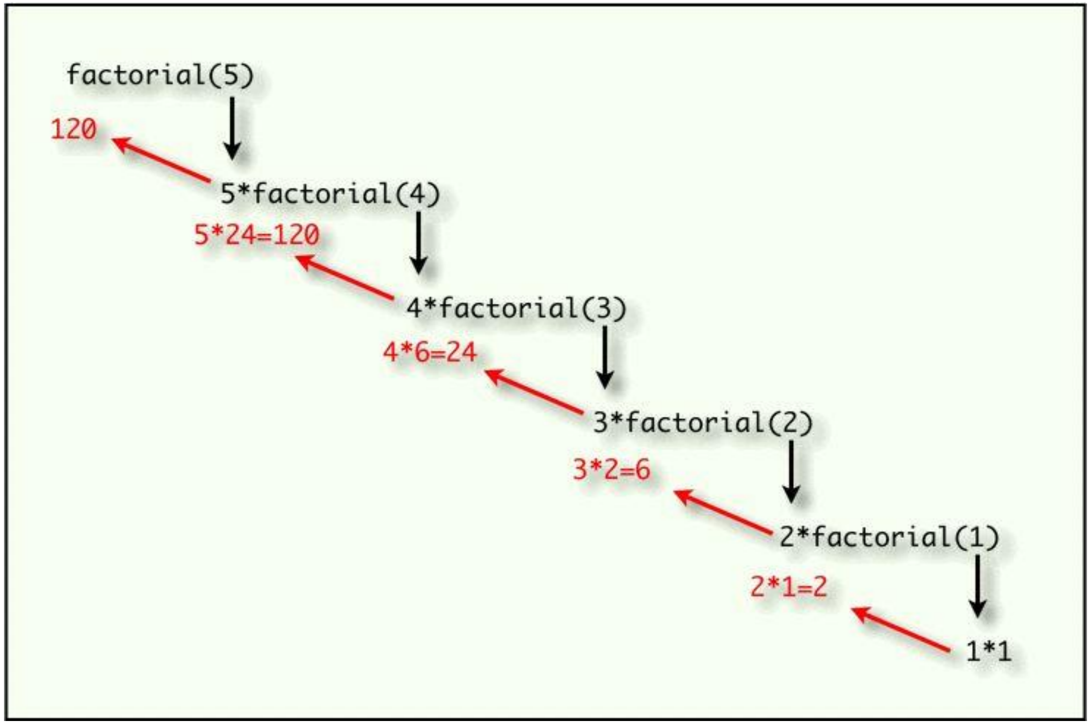
Esquema básico II

- Ejemplo de cálculo recursivo del factorial de un número “n”
  - Recordemos: n! = n·(n-1)·(n-2)·…·1 (esta es la “versión iterativa”)
  - La versión recursiva sería: n! = n·(n-1)!

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

Esquema básico III

- Para cualquier algoritmo recursivo podemos dibujar un “árbol de llamadas recursivas” para intentar comprender la traza del algoritmo:



Ventajas de la recursividad I

- La recursividad se integra perfectamente y sigue la estrategia de “divide y vencerás” para resolver problemas:
  - Se divide un problema en 2 o más subproblemas (y éstos a su vez en otros subproblemas, y así sucesivamente)
  - Cada subproblema es de naturaleza similar al problema inicial…
  - … pero cada subproblema es de tamaño menor que el problema inicial
  - Se resuelve cada subproblema por separado
  - Una vez resueltos, se combinan sus resultados para producir la solución del problema original

Ventajas de la recursividad II

- A menudo es más fácil realizar un programa recursivo que su equivalente iterativo
  - Porque muchas definiciones y problemas son de naturaleza recursiva…
  - … y, por lo tanto, convertirlos en algoritmo recursivo es inmediato y sencillo

Ventajas de la recursividad III

- Hay ciertas estructuras de datos que se definen recursivamente
  - Listas, árboles, grafos…
  - Al ser recursivas, se manipulan muy fácilmente con algoritmos recursivos
  - Ej: se puede ordenar una lista ordenando cada mitad por separado y luego juntando los resultados (quicksort)
  - Ej: se puede buscar en una lista mirando si en la primera posición está lo buscado... si no está, volvemos a buscar en una lista más pequeña (la que empieza a partir del segundo elemento)

¿Funciona la recursividad? I

- Pensar, programar, probar, demostrar o asegurarse de que un algoritmo recursivo funciona es sencillo. Basta con asegurarse de que se cumple todo lo siguiente:
  - Tiene que existir una salida no recursiva del algoritmo (caso trivial)
    - En el factorial, cuando n es 0
  - Cada llamada recursiva se refiere a un problema más simple
    - La llamada recursiva se hace con n-1, más simple que si la hiciéramos con n
  - Suponiendo que las llamadas recursivas funcionan, el algoritmo debe de funcionar
    - En el factorial, si la llamada recursiva funciona y devuelve (n-1)!, entonces hay que comprobar que el algoritmo devuelve n!
- Por lo tanto, no hay que pensar el árbol de llamadas recursivas para programar o entender el algoritmo recursivo en cuestión o para demostrar y/o asegurarse de que efectivamente sí funciona

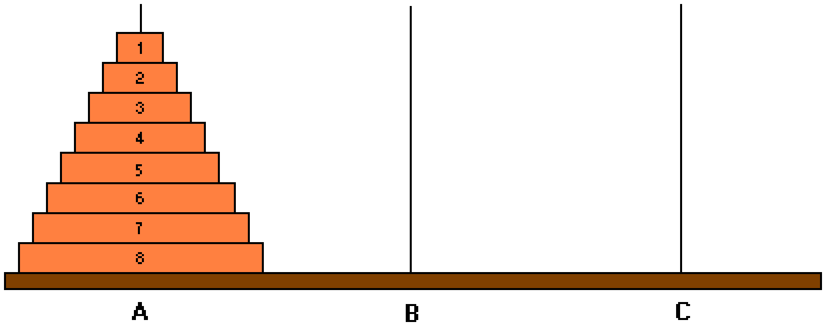
¿Funciona la recursividad? II

- Veamos si hemos comprendido cómo demostrar que un algoritmo recursivo funciona. Usemos para ello el ejemplo de las torres de Hanoi:
  - Tenemos tres postes llamados A, B y C
  - Tenemos n discos, todos ellos de tamaños diferentes
  - Inicialmente tenemos todos los discos en A, de manera que cada disco está encima de todos los que son mayores que él
  - El objetivo es dejar todos los discos en C
  - Reglas:
    - Sólo se puede mover un disco a la vez
    - No se puede poner un disco encima de otro menor
    - Sólo se puede coger el disco más pequeño de cada poste

¿Funciona la recursividad? III

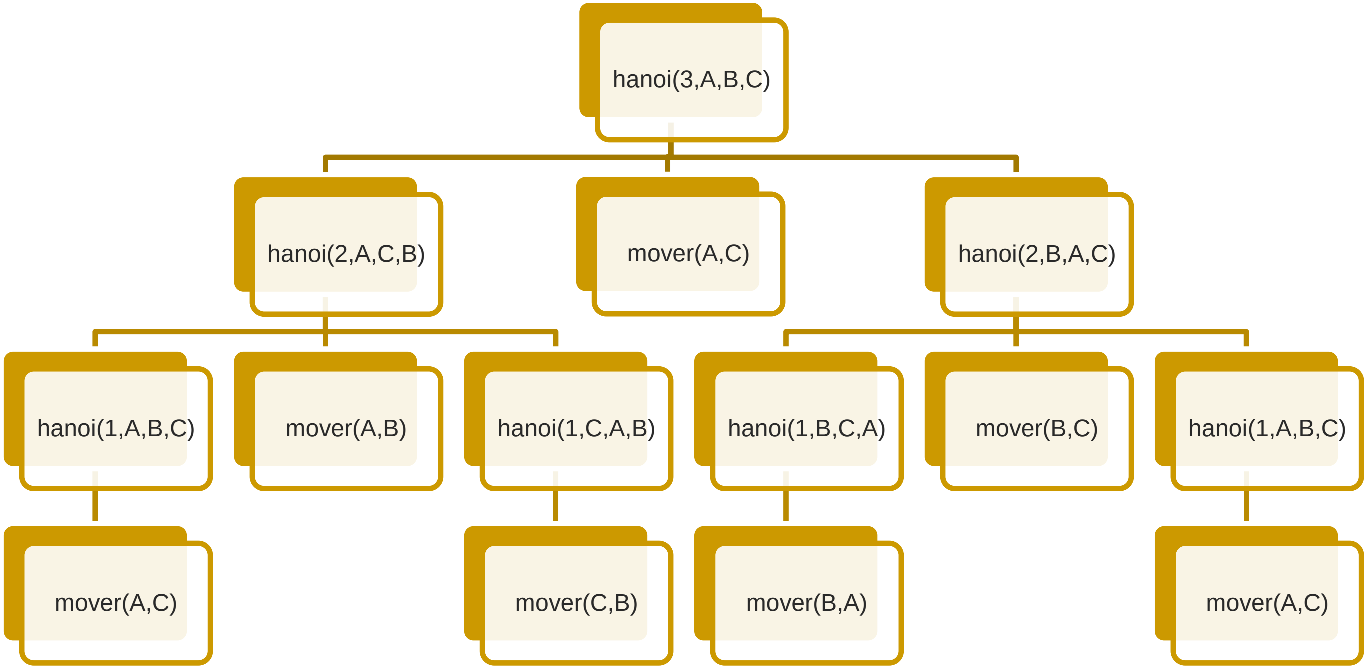
- Pseudocódigo de las Torres de Hanoi (el origen es “A”, el auxiliar es “B” y el destino es “C”):

algoritmo *hanoi*
entrada *entero n*; *poste origen*, *auxiliar*, *destino*
si $n > 0$ **entonces**
    hanoi($n-1$, *origen* , *destino* , *auxiliar*)
    mover(*origen* , *destino*)
    hanoi($n-1$, *auxiliar* , *origen* , *destino*)



¿Funciona la recursividad? IV

- Podemos dibujar el árbol de llamadas recursivas (ej: para n=3), pero, como dijimos, no es necesario para comprender que el algoritmo realmente funciona



¿Funciona la recursividad? V

- EjercicioRecursividad: vamos a ensayar programando varios algoritmos recursivos:
  - Algoritmo de Euclides
  - Potenciación
  - Sumatorio
  - Fibonacci
  - Búsqueda
  - Multiplicación a la rusa

Recursividad y complejidad I

- El T(n) de un algoritmo recursivo es una función/ecuación en recurrencia
  - Una función en recurrencia o recurrente es aquélla que se incluye a sí misma en su definición
  - Todas las funciones recurrentes tienen:
    - Una o varias condiciones iniciales
      - Las sacaremos del caso trivial
    - Una expresión general
      - La sacaremos del caso no trivial
  - Veamos algunos ejemplos

Recursividad y complejidad II

¿Cuánto tarda en ejecutarse el algoritmo recursivo para calcular el factorial de un número "n"?

- La condición inicial (cuando n=0), la cual sacamos del caso trivial del algoritmo, es T(0)=5, porque:
  - 1 OE por el salto hacia la línea 1 desde el main o desde donde sea, 1 OE por la declaración del parámetro formal, 1 OE por la evaluación de la condición, 1 OE por el return, 1 OE por el salto hacia el main
- La condición general (cuando n no es 0) la sacamos del caso no trivial y es T(n)=T(n-1)+9, porque:
  - 4 OE por el salto al comienzo de la función, declaración del parámetro formal, evaluación de la condición y salto al else
  - En la línea 3: 5 OE por la multiplicación, resta, return, asignación de n-1 al parámetro formal, y salto hacia el main (o hacia la función que nos haya llamado)
  - En la línea 3: T(n-1) OE va a tardar la ejecución de la función factorial(n-1). Como no sabemos lo que es exactamente, lo dejamos indicado como T(n-1)

```
1 int factorial(int n) {
2     if (n == 0) return(1);
3     else return(n * factorial(n - 1));
4 }
```

Recursividad y complejidad III

T(n) del algoritmo recursivo de Hanoi:

- T(0)=8, porque:
  - Salto hasta nuestra función, 4 declaraciones de parámetros formales, evaluación de la condición, salto el final de la función, salto hasta el main)
- T(n)=2·T(n-1)+20 para n>0, porque:
  - 5 OE por salto hasta nuestra función y 4 declaraciones de parámetros formales
  - 1 OE por evaluación de la condición
  - T(n-1)+5 OE por la primera llamada recursiva (incluye una resta y 4 asignaciones)
  - 3 OE por la línea del mover() (suponemos que mover() es 1 OE, y 2 OE por la asignación de los dos parámetros que tiene)
  - T(n-1)+5 OE por la primera llamada recursiva (incluye una resta y 4 asignaciones)
  - 1 OE por salto hasta el main

```
algoritmo hanoi
entrada entero n; poste origen, auxiliar, destino
si n > 0 entonces
    hanoi( n − 1 , origen , destino , auxiliar )
    mover( origen , destino )
    hanoi( n − 1 , auxiliar , origen , destino )
```

Recursividad y complejidad IV

- De una función en recurrencia no podemos obtener directamente su complejidad
- Para ello, primero tenemos que convertirla en una función no recurrente que sea equivalente a la recurrente
  - Es decir, convertirla en una función que no dependa de sí misma
  - A partir de la función no recurrente equivalente ya podríamos obtener su O(…) según las reglas vistas
  - Una forma de eliminar recurrencias sencillas es desarrollar la serie recurrente y a partir de ahí intentar deducir la fórmula general sin recurrencia
    - Existe un procedimiento general para eliminar cualquier recurrencia, pero no lo vemos en esta asignatura
    - Siempre podemos usar www.wolframalpha.com

Recursividad y complejidad V

- Ejemplo: factorial
  - Según calculamos, tenemos T(0)=5 y T(n)=T(n-1)+9
  - Desarrollamos unos cuantos elementos de la serie, hasta que consigamos deducir intuitivamente la fórmula general sin recurrencia:
    - T(0)=5
    - T(1)=T(0)+9=5+9=14
    - T(2)=T(1)+9=14+9=23
    - T(3)=T(2)+9=23+9=32
    - T(4)=T(3)+9=32+9=41
    - …
  - Nos damos cuenta, intuitivamente, que:
    - T(n)=9n+5
  - En www.wolframalpha.com, pondríamos "T(0)=5, T(n)=T(n-1)+9" (sin las comillas) y nos daría "T(n)=9n+5"
  - Ya podemos averiguar la complejidad, con las reglas anteriormente vistas. **Por lo tanto: T(n)=9n+5 $\in$ O(n)**

Recursividad y complejidad VI

- Ejemplo: Hanoi
  - Según calculamos, tenemos T(0)=8 y T(n)=2T(n-1)+20
  - Desarrollamos unos cuantos elementos de la serie:
    - T(0)=8
    - T(1)=2T(0)+20=16+20=36
    - T(2)=2T(1)+20=72+20=92
    - T(3)=2T(2)+20= 184+20=204
    - …
  - Nos podríamos dar cuenta, intuitivamente, que:
    - $T(n)=28\cdot 2^n-20$
  - Si no lo averiguamos intuitivamente (que, en este caso difícil, es lo más probable), no nos queda más remedio que utilizar www.wolframalpha.com
    - Escribimos "T(0)=8, T(n)=2T(n-1)+20" y nos da el resultado "$T(n)=4(7x2^n-5)$", que es equivalente a $T(n)=28\cdot 2^n-20$
  - Ya podemos averiguar la complejidad, con las reglas anteriormente vistas. **Por lo tanto: $T(n)=28\cdot 2^n-20 \in O(2^n)$**

Recursividad y complejidad VII

- EjercicioComplejidad2. Vamos a calcular el T(n) y la complejidad temporal de varios algoritmos recursivos sencillos
  - Para ello, habrá que hacer varios pasos:
    1. Calcular su T(n) recurrente, no olvidando las condiciones iniciales.
    2. Intentar eliminar la recurrencia intuitivamente, desarrollando la serie
    3. Si no te es posible obtener intuitivamente la solución, utilizar WolframAlpha
    4. A partir del T(n) no recurrente obtenido, calcular su O(n)

Definición de un TAD I

- Recordemos que una estructura de datos está formada por dos partes:
  - Un conjunto de datos, relacionados entre sí, que tienen un propósito y uso común
  - Varios algoritmos para manipular ese conjunto de datos
- La descripción formal (matemática, sin programarla) de una Estructura de Datos se llama “Tipo Abstracto de Datos” (TAD)

Definición de un TAD II

- Un ejemplo típico de TAD es un número racional (una fracción). Vamos a describir el TAD por dentro usando lenguaje natural:
  - Datos que contiene el TAD por dentro: numerador (tipo entero) y denominador (tipo entero)
  - Algunos algoritmos para manipular este TAD:
    - De creación de la estructura de datos, a partir de dos enteros que el usuario debería proporcionar
      - Dichos enteros pasarán a ser el numerador inicial y el denominador inicial de la fracción
    - De creación de un nuevo racional a partir de otros racionales
      - Ej: sumar, multiplicar
    - De obtención de información de la estructura
      - Ej: getNumerador, getDenominador
    - De modificación de la estructura
      - Ej: simplificar. Se haría calculando el máximo común divisor del denominador y el denominador, por ejemplo mediante el algoritmo de Euclides. Luego se divide el numerador y el denominador entre ese máximo común divisor.

Definición de un TAD III

- Una vez descrito el TAD por dentro, ya podemos usarlo como queramos, incluso para definir otros TADs
- Ejemplo con el TAD "Número racional":
  - Ahora podemos crear tantas fracciones distintas como necesitemos para nuestros cálculos
  - Podemos sumarlas o multiplicarlas entre ellas para crear nuevas fracciones resultado
  - Podemos simplificar una fracción, y quedará modificada por dentro (su numerador y denominador será distinto)
  - Podemos usar el TAD "Número Racional" para definir otro TAD, por ejemplo el TAD "Tarta".
    - El TAD "Tarta" tendrá un dato interno de tipo fracción que nos indique la fracción de tarta que nos queda, después de ejecutar su algoritmo "comerPorción".

Clases y objetos I

- ¿Cómo programaríamos un TAD? Mediante una “clase”.
  - Para programar una “clase”, necesitamos un lenguaje de programación que sea “orientado a objetos”. Ej: C++, Java
  - Recordemos que una clase es una estructura de datos (conjunto de sus atributos) más los algoritmos para manipularla (conjunto de métodos)
  - A partir de las clases, se crearán objetos (los veremos ahora)

Clases y objetos II

- Un objeto es básicamente cualquier elemento que se pueda nombrar:
  - un árbol específico (el que está en frente de mi ventana),
  - una persona (Juan González),
  - un ordenador (el primero que tuve, por ejemplo),
  - una pared (la que separa el baño de la habitación principal),
  - un avión (el que no pude comprar el año pasado),
  - un automóvil (mi propio coche)...
- Una clase es un tipo de objetos similares: árboles, personas, ordenadores, paredes, aviones, automóviles...

Clases y objetos III

- Así pues, un objeto es un elemento particular de su clase:
  - Juan es un objeto de la clase "Personas"
  - El árbol que está en frente de mi ventana es un objeto particular de la clase "Árboles"
  - Mi coche es un elemento particular de la clase "Coches"

Clases y objetos IV

- En un lenguaje orientado a objetos, se escriben las clases en el código fuente
- A partir de dichas clases, creamos todos los objetos que queramos
- De este modo, una clase actúa como un "molde" para crear objetos
  - Si tenemos ya escrita la clase "Personas", podemos crear a Juan, Pepe, Luisa y Menganito (4 objetos distintos, pero de la misma clase)

Definir una clase I

- ¿Qué hay dentro de una clase?
  - Un conjunto de atributos o características
    - Es el nombre de las características que posee cualquier objeto de esa clase
    - Ej: la clase Persona (es decir: cualquier persona) tiene una altura, un color de ojos, una edad, una profesión, etc.
  - Un conjunto de operaciones o métodos
    - Son las acciones que puede realizar cualquier objeto de esa clase.
    - Ej: cualquier persona anda, piensa, corre, trabaja, duerme, etc.
    - Ej: cualquier avión despega, aterriza y vuela

Definir una clase II

- ¿Qué tiene un objeto por dentro?
  - Valores concretos de los atributos de la clase.
    - Es el valor concreto que toma un atributo en cada objeto de su clase
    - Ej: Juan (de la clase Persona) mide 173 cm (valor concreto del atributo altura), tiene ojos marrones (valor concreto del atributo color de ojos), su edad es 31 años, es abogado...
  - Operaciones. Todos los objetos de una clase pueden realizar las operaciones definidas en esa clase, sin más.
    - Ej: tanto Juan como María (miembros de la clase Personas) pueden ambos andar y pensar.

Definir una clase III

- Así pues, al escribir una clase, deberemos decir:
  - Nombre de la clase ("Persona")
  - Qué atributos tiene (altura, color de ojos, edad y profesión)
  - Qué métodos tiene (andar, correr, trabajar y dormir)
- Al crear un objeto a partir de una clase (llamado también "instanciar un objeto a partir de una clase"), tendremos que decir:
  - El nombre del objeto ("Juan")
  - De qué clase es ("Persona")
  - Qué valores concretos tienen sus atributos (173 cm de altura, ojos marrones, 31 años, abogado)
- Varios objetos de la misma clase se diferencian entre sí sólo en el valor concreto de sus atributos

Definir una clase IV

- Cada atributo de una clase puede ser, además, un objeto de otra clase:
  - Ej: la clase Coche tiene 4 ruedas concretas. Cada rueda concreta es un objeto de la clase Rueda
- Una clase también puede ser una lista de objetos
  - ¿Por qué no? Una lista es también algo que podemos nombrar. Sus atributos son los objetos (elementos que contiene) y sus operaciones podrían ser obtener un elemento, ordenar todos, etc.

Usar objetos I

- Para poder crear una aplicación necesitarás más de un objeto, y estos objetos no pueden estar aislados unos de otros.
- Pues bien, para comunicarse esos objetos entre sí se envían mensajes.
- Los mensajes son simples llamadas de un objeto (o main) a un método de otro objeto para decirle que haga cualquier cosa.
  - Ej: yo (objeto Carlos de la clase "Personas") me monto en mi coche (objeto específico de la clase Coches) y le envío el mensaje "arranca". Es decir, ejecuto el método arrancar() de la clase Coche sobre el objeto que es mi coche concreto

Usar objetos II

- Encapsulación significa que los atributos de una clase no pueden ser leídos o modificados directamente desde el exterior (desde otro objeto o desde el main)
- Para cambiar el valor de un atributo de un objeto, habría que hacerlo a través de uno de los métodos
- Ejemplo: otra persona no puede cambiar mi profesión directamente, sino que para ello tiene que llamar a mi método "contratarme"
- Así pues, los datos internos de un objeto están "protegidos", están encapsulados. Sólo mis propios métodos pueden acceder directamente a ellos. Por lo tanto, los demás deberán llamar a mis métodos

Programación de un TAD I

- Ejemplo guiado (EjemploRacional):
  - Vamos a programar el TAD número racional en C++, utilizando la clase "Racional"
  - Por cada clase, se necesitan dos archivos:
    - Un archivo de cabecera .h que declare lo que tiene la clase por dentro (atributos y métodos)
    - Un archivo de código fuente .cpp que contenga el código de cada uno de los métodos de la clase
  - Además todo programa de C++ necesita una función "main" que coordine todo el programa
  - Vamos a ir explicando todos los detalles del programa y lo que es una clase y un objeto mientras vemos el código fuente

Programación de un TAD II

- EjercicioPunto2D: programar el TAD "Punto2D" en C++ y probar todos sus métodos con un "main"
  - El TAD representa a un punto en 2 dimensiones (es decir, en las coordenadas "x" e "y")
  - La clase "Punto2D" tendrá dos atributos: la coordenada x, y la coordenada y
  - Métodos de la clase "Punto2D" para manipular la estructura de datos:
    - Constructor: le pasamos dos números reales y con ellos inicializa las dos coordenadas
    - Desplazar: le pasamos el desplazamiento en el eje x, y el desplazamiento en el eje y. Modificará las coordenadas x e y según ese desplazamiento.
    - CalcularDistanciaAlOrigen: nos devuelve la distancia del punto hasta el origen de coordenadas, usando el Teorema de Pitágoras para calcular la hipotenusa (el valor absoluto de las coordenadas son los catetos)
      - Busca en www.cplusplus.com la biblioteca "cmath" y encuentra una función que te permita hacer la raíz cuadrada
    - Visualizar: imprime por pantalla el punto. Ej: si x=5 e y=3, se imprimirá por pantalla "(5, 3)" (sin las comillas)