Parte II. Estructuras de datos y algoritmos

UNIVERSIDAD DE CANTABRIA

Tema 10. Introducción al análisis y diseño de algoritmos.

- Diseño de un programa.
- Concepto de algoritmo.
- Descripción de algoritmos: el pseudolenguaje y diagramas de flujo.
- Algoritmos recursivos.
- Tiempo de ejecución.
- La notación O(n).
- Ejemplos de análisis.

1. Diseño de un programa: Necesidad de la programación modular

UNIVERSIDAD DE CANTABRIA

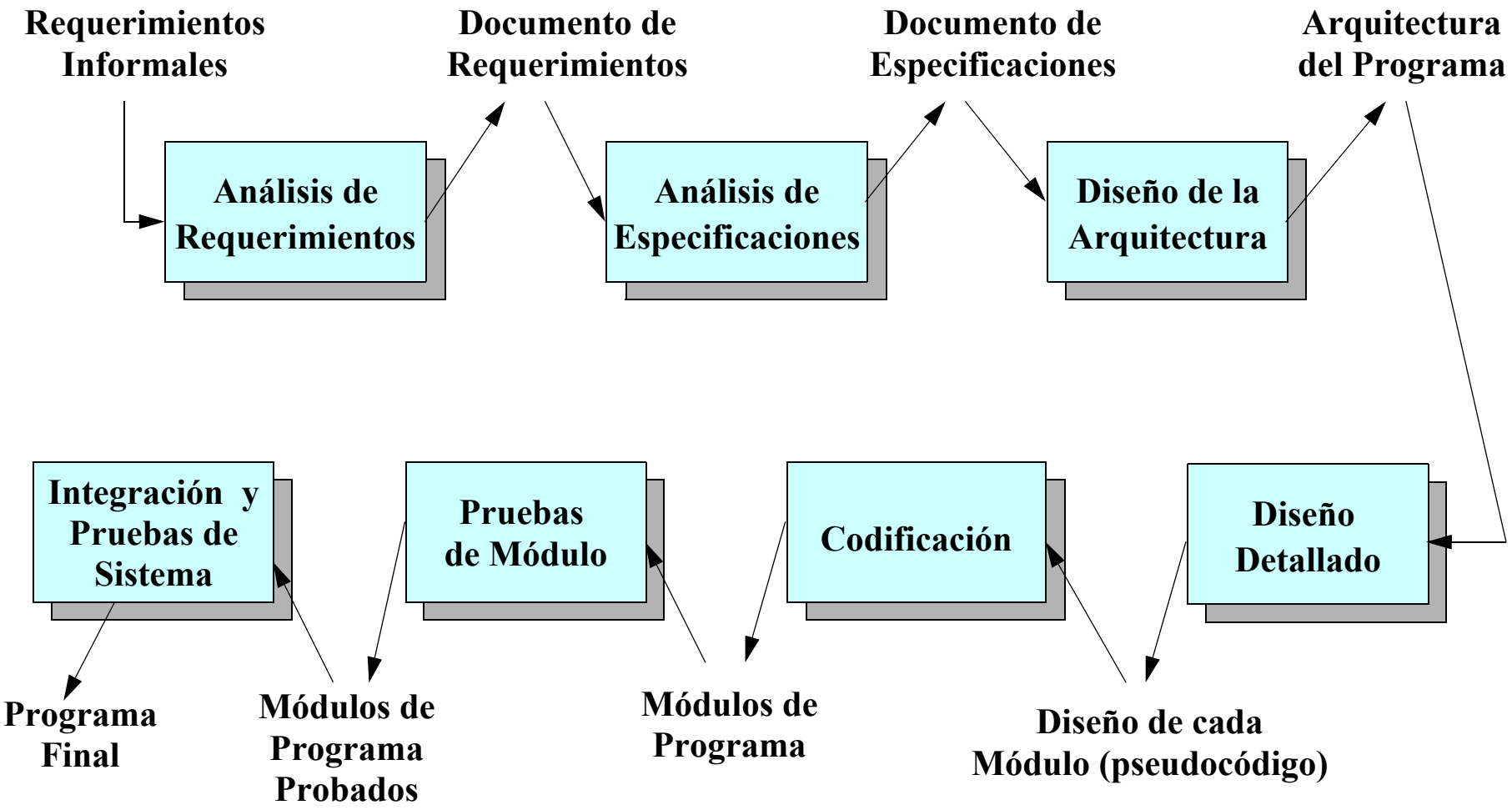
Los programas pueden llegar a ser muy complejos y para poder gestionar esta complejidad se dividen en módulos

- independientes entre sí
- llamados paquetes en Ada

Un módulo de programa contiene datos y operaciones para manipular datos

Cada una de esas operaciones representa un algoritmo

El ciclo de vida de los programas



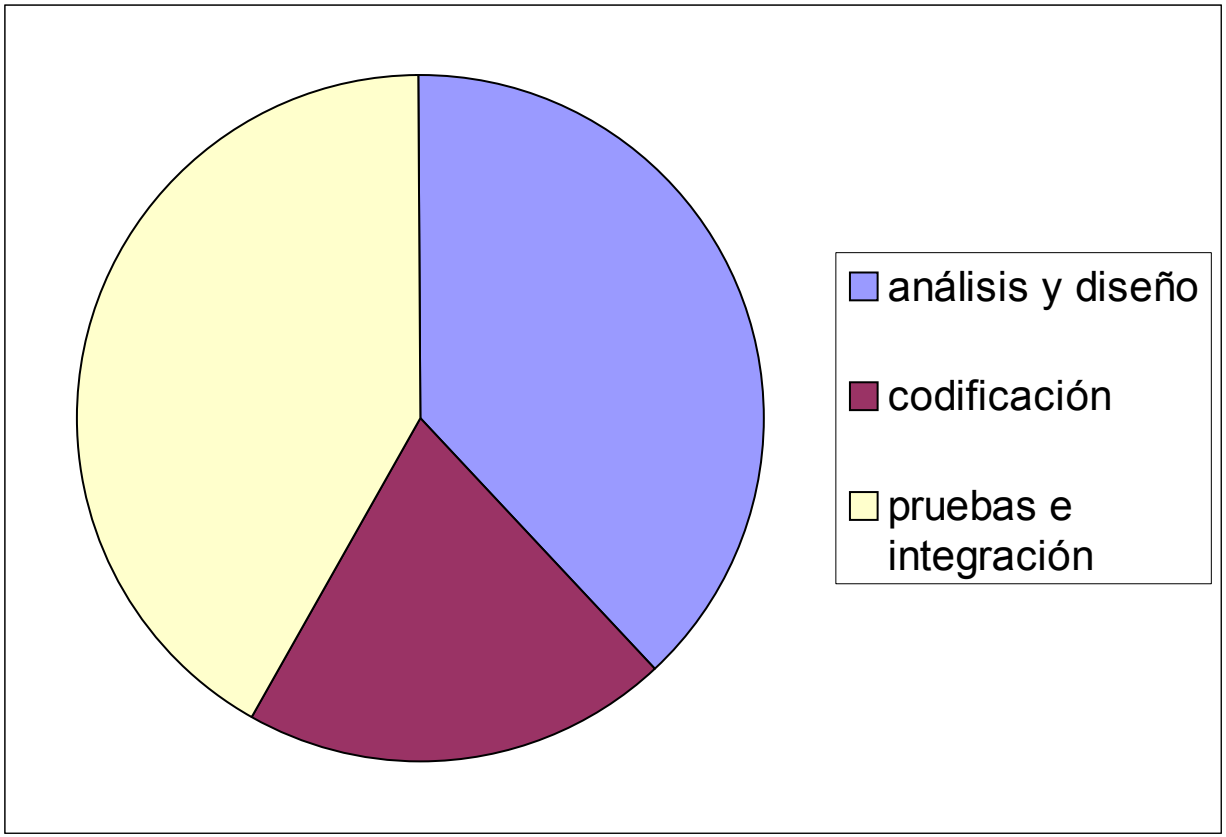
Notas:

UNIVERSIDAD DE CANTABRIA

Los pasos que se siguen generalmente a la hora de desarrollar un programa son los siguientes:

- *Análisis de requerimientos*: Se define el problema a resolver y todos los objetivos que se pretenden, pero sin indicar la forma en la que se resuelve.
- *Especificación*: Se determina la forma en la que se resolverá el problema, pero sin entrar aún en su implementación informática. Se determina asimismo la interfaz con el usuario.
- *Diseño del programa:* Se divide el problema en módulos, se especifica lo que hace cada módulo, así como las interfaces de cada uno de ellos.
- *Diseño detallado de los módulos*: Para cada módulo se diseñan detalladamente las estructuras de datos y los algoritmos a emplear, normalmente descritos mediante pseudocódigo.
- *Codificación*: Se escribe el programa en el lenguaje de programación elegido.
- *Pruebas de módulos*: Se prueban los módulos del programa aisladamente y se corrigen los fallos hasta conseguir un funcionamiento correcto.
- *Integración y Prueba de sistema*: Se unen todos los módulos, y se prueba el funcionamiento del programa completo.

Distribución del Esfuerzo



Notas:

UNIVERSIDAD DE CANTABRIA

Los siguientes datos se han obtenido del estudio de muchos proyectos software reales.

Distribución del esfuerzo de la actividad software (sin tener en cuenta el mantenimiento):

- Análisis y Diseño : 38%
- Codificación : 20%
- Pruebas e integración: 42%.

2. Concepto de algoritmo

UNIVERSIDAD DE CANTABRIA

Un algoritmo es:

- una secuencia finita de instrucciones,
- cada una de ellas con un claro significado,
- que puede ser realizada con un esfuerzo finito
- y en un tiempo finito

El algoritmo se diseña en la etapa de diseño detallado y se corresponde habitualmente con el nivel de subprograma

Los programas se componen habitualmente de muchas clases que contienen algoritmos, junto con datos utilizados por ellos

Notas:

UNIVERSIDAD DE CANTABRIA

El algoritmo se diseña durante la etapa de diseño detallado dentro del ciclo de vida del programa. Habitualmente se corresponde con el nivel de subprograma, es decir, cada subprograma implementa un algoritmo.

Un algoritmo es una secuencia finita de instrucciones, cada una de ellas con un claro significado, que puede ser realizada con un esfuerzo y un tiempo finitos. Por ejemplo, una asignación como x:=y+z es una instrucción con estas características

Un algoritmo que suministra una solución buena, pero no necesariamente óptima, se le denomina algoritmo heurístico y suele utilizarse en aquellos problemas denominados “NP-completos” y cuya solución óptima pasa por “intentar todas las posibilidades”, donde el número de posibilidades es una función exponencial del tamaño del problema.

Estructuras algorítmicas

UNIVERSIDAD DE CANTABRIA

Las estructuras algorítmicas permiten componer instrucciones de un computador para que se ejecuten en el orden deseado

Las principales estructuras son:

- **Composición secuencial**
  - las instrucciones se ejecutan en secuencia, una tras otra
- **Composición alternativa o condicional**
  - en función de una condición se eligen unas instrucciones u otras
- **Estructura iterativa o bucle o lazo**
  - se repiten unas instrucciones mientras se cumple una condición
- **Estructura recursiva**
  - se repiten unas instrucciones mediante un método que se invoca a sí mismo

Instrucciones que implementan las estructuras algorítmicas

UNIVERSIDAD DE CANTABRIA

Estructura algorítmica	Instrucción
Composición secuencial	Ninguna. Simplemente se ponen las instrucciones una detrás de otra
Composición alternativa	Instrucciones de control: condicionales
Estructura iterativa	Instrucciones de control: bucles
Estructura recursiva	procedimiento o función que se invoca a sí mismo

3. Descripción de algoritmos

UNIVERSIDAD DE CANTABRIA

Un algoritmo se puede especificar mediante la utilización de un lenguaje de programación

Sin embargo, generalmente se suelen utilizar técnicas de descripción de algoritmos más o menos independientes del lenguaje de programación:

- ***diagrama de flujo***
  - **es un gráfico que muestra el orden en el que se van ejecutando las diferentes instrucciones**
- ***pseudolenguaje* o *pseudocódigo***
  - **utiliza instrucciones de control y lenguaje natural**

Notas:

UNIVERSIDAD DE CANTABRIA

Un algoritmo se puede especificar mediante la utilización de un lenguaje de programación. En este caso el algoritmo se describe en términos del conjunto de instrucciones del lenguaje.

Sin embargo, generalmente se suelen utilizar técnicas de descripción de algoritmos más o menos independientes del lenguaje de programación en el que se codifican. Tal es el caso de los **diagramas de flujo** o del **pseudolenguaje.**

El diagrama de flujo es un gráfico en el que se representa el orden en el que se van ejecutando las diferentes instrucciones que forma el algoritmo.

El **pseudolenguaje** se puede considerar como un método para la descripción de algoritmos en el que se utiliza una combinación de instrucciones informales escritas en lenguaje natural y de órdenes de control estructuradas.

Ejemplos

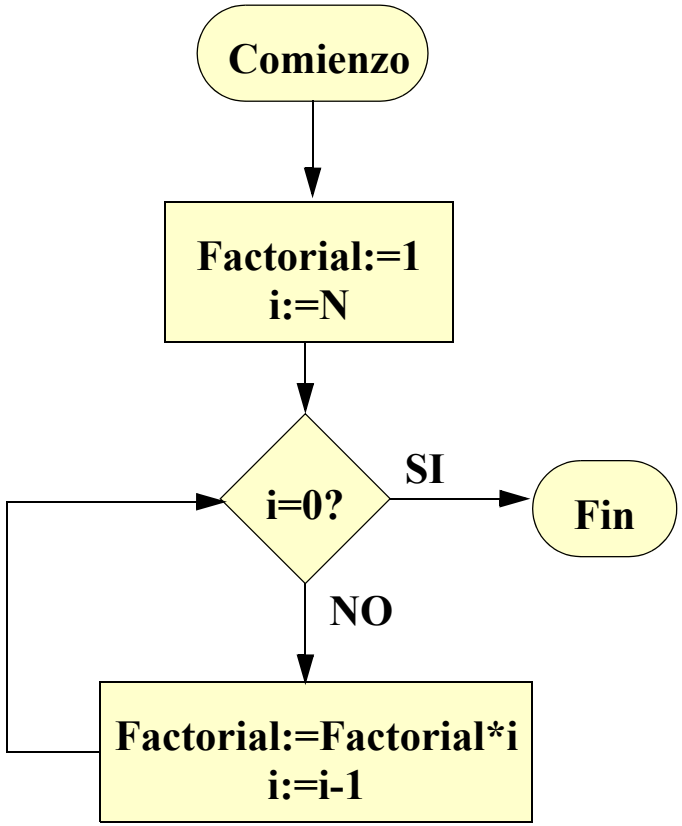


Diagrama de flujo

```
algoritmo factorial
    (N : Natural)

factorial:=1
para i desde 1 hasta N
   factorial:=factorial*i
fin para
```

Pseudocódigo

Notas:

UNIVERSIDAD DE CANTABRIA

Puede observarse en estos ejemplos que el diagrama de flujo es más voluminoso que el pseudocódigo. También es más difícil de modificar.

Por otro lado el pseudocódigo se puede escribir con un editor de texto, mientras que el pseudocódigo necesita un editor gráfico. Por estos motivos, es normalmente más recomendable el pseudocódigo, aunque hay algunos problemas cuya naturaleza se expresa mejor mediante diagrama de flujo.

Descripción de algoritmos mediante pseudocódigo

UNIVERSIDAD DE CANTABRIA

Una técnica muy habitual para describir algoritmos es el pseudocódigo. Tiene como objetivos

- descripción sencilla, sin los formalismos de un lenguaje de programación
- descripción independiente del lenguaje de programación
  - directamente traducible a código en cualquier lenguaje

El pseudocódigo contiene:

- instrucciones de control presentes en todos los lenguajes
- expresiones con cálculos
- acciones expresadas sin el formalismo de los lenguajes

Pseudocódigo (cont.)

UNIVERSIDAD DE CANTABRIA

Instrucciones de control

condicional	bucle while	bucle for

`si condición entonces`
`  instrucciones`
`si no`
`  instrucciones`
`fin si`
	`mientras condición`
	`  instrucciones`
	`fin mientras`
		`para cada i desde 1 hasta n`
		`  instrucciones`
		`fin para`

Expresiones con cálculos

```
i=suma+3*x
```

Acciones expresadas sin el formalismo de los lenguajes

```
leer i y j de teclado
escribir resultado en la pantalla
```

Ejemplo: suma de los 100 primeros enteros positivos

UNIVERSIDAD DE CANTABRIA

Ada	Pseudocódigo

`suma : Integer:=0;`
`for i in 1..100 loop`
`  suma:=suma+i;`
`end loop`
	`suma : Entero :=0;`
	`para i desde 1 hasta 100`
	`  suma:=suma+i;`
	`fin para`

También para incrementos distintos de uno (ej: hacia atrás):

Ada	Pseudocódigo

`suma : Integer:=0;`
`for i in reverse 1..100`
`loop`
`  suma:=suma+i;`
`end loop`
	`suma : Entero :=0;`
	`para i desde 2 hasta 100 paso -1`
	`  suma:=suma+i;`
	`fin para`

Ejemplo

UNIVERSIDAD DE CANTABRIA

Vamos a escribir una función para obtener el valor del *logaritmo* de `y=1+x` de acuerdo con el siguiente desarrollo en serie

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n-1}\frac{x^n}{n}, \quad -1 < x \leq 1$$

Para calcular de manera eficiente el signo y el numerador:

- **no usaremos potencias**
- **el signo va cambiando de un término al siguiente**
- **el numerador siempre es el del término anterior por x**

Diseño

```
función logaritmo (real y, entero n) retorna real
   x:real :=y-1;
   log:real :=0; // para recoger el resultado
   numerador:real :=x; // primer numerador
   signo:entero :=1;   // primer signo
   para cada i desde 1 hasta n
      log:=log+signo*numerador/i;
      // calculamos el numerador y denominador
      // para la próxima vez
      numerador:=numerador*x;
      signo:=-signo;
   fin para
   retorna log;
fin función
```

Refinamiento paso a paso

UNIVERSIDAD DE CANTABRIA

Cuando se describe un algoritmo:

- se suele comenzar con una descripción a nivel alto, con instrucciones generales
- es preciso desarrollarla en conjuntos de instrucciones más simples y precisas

Esto ocurre tanto con diagramas de flujo como con pseudocódigo

Por tanto, en estos casos, se suele realizar un proceso de refinamiento paso a paso

- el algoritmo se describe con un nivel de detalle cada vez mayor

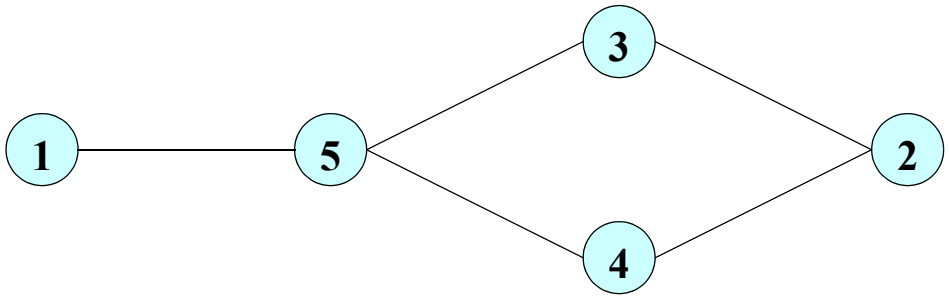
Las transparencias siguientes muestran un ejemplo de esto

Ejemplo de refinamiento paso a paso

UNIVERSIDAD DE CANTABRIA

Problema de 'coloreado' de un grafo de forma que:

- **dos vértices contiguos no sean del mismo color**
- **se minimice el número de colores**



Una solución:

Rojo => 1,3,4
Verde => 5,2

La solución óptima sólo se consigue probando todas las posibilidades

Existen algoritmos heurísticos para buscar una solución buena

- **ejemplo: algoritmo voraz**

Notas:

UNIVERSIDAD DE CANTABRIA

Un grafo es una estructura de datos que sirve para establecer relaciones de muchos a muchos. Tiene dos tipos de datos:

- *Vértices*: son los datos que vamos a relacionar (los círculos, en el dibujo)
- *Arcos o Aristas*: son relaciones entre dos vértices. Puede tener dirección, o no (las rayas, en el dibujo, son aristas sin dirección)

Se dice que dos vértices son *contiguos* si están unidos mediante una arista.

El problema del coloreado de grafos tiene muchas aplicaciones. Por ejemplo, se puede usar para diseñar un sistema de semáforos en el que el objetivo es obtener un conjunto, lo más pequeño posible, de giros permitidos para asociar una fase de los semáforos a cada grupo.

Los algoritmos *heurísticos* encuentran una solución más o menos buena, sin garantizarse que encuentran la solución óptima.

Algoritmo Voraz

UNIVERSIDAD DE CANTABRIA

Descrito mediante lenguaje natural:

1. **Seleccionar un vértice no coloreado aún, y asignarle un color**
2. **Recorrer la lista de vértices sin colorear. Para cada uno de ellos determinar si tiene un arco conectándolo con cualquier otro vértice coloreado con el nuevo color. Si no existe tal arco, colorear ese vértice con el nuevo color**

Hay que repetir el algoritmo hasta que no queden vértices sin colorear

Notas:

UNIVERSIDAD DE CANTABRIA

Un posible algoritmo heurístico es el conocido como algoritmo voraz, que se puede describir en una primera instancia mediante lenguaje natural:

1. Seleccionar un vértice no coloreado aún, y asignarle un color
2. Recorrer la lista de vértices sin colorear. Para cada uno de ellos determinar si tiene un arco conectándolo con cualquier otro vértice coloreado con el nuevo color. Si no existe tal arco, colorear ese vértice con el nuevo color

Este algoritmo voraz, por ejemplo, no encontraría la solución indicada arriba si se recorren los vértices según el orden de su numeración, sino que asignaría:

- Rojo => 1,2
- Verde => 3,4
- Azul => 5

Que es una solución peor.

Ejemplo de refinamiento (cont.)

UNIVERSIDAD DE CANTABRIA

Se muestra aquí la primera aproximación del pseudocódigo:

```
algoritmo Voraz (G : in out grafo; nuevo_color : out lista) es
     -- Asigna a la lista de vértices nuevo_color los vértices
     -- del grafo G a los que se les puede dar el mismo color
comienzo
     nuevo_color := vacío
     para cada vértice v no coloreado en G hacer
        si v no es adyacente a ningún vertice en nuevo_color entonces
           colorea v
           añade v a nuevo_color
        fin si
     fin para
fin Voraz
```

Segunda aproximación

UNIVERSIDAD DE CANTABRIA

```
algoritmo Voraz (G : in out grafo; nuevo_color : out lista) es
      -- Asigna a la lista de vértices nuevo_color, los vértices
      -- del grafo G a los que se les puede dar el mismo color
comienzo
      nuevo_color := vacío
      para cada vértice v no coloreado en G hacer
         encontrado := falso
         para cada vértice w en nuevo_color hacer
            si hay arco entre v y w entonces
               encontrado:=cierto
            sin si
         fin para
         si encontrado = falso entonces
            colorea v
            añade v a nuevo_color
         fin si
      fin para
fin Voraz
```

Notas:

UNIVERSIDAD DE CANTABRIA

Puede observarse que en la primera aproximación hay instrucciones que resultan difíciles de programar directamente en un lenguaje de programación

- En particular: **`v no es adyacente a ningún vertice en nuevo_color`**

En la segunda aproximación se detalla la forma en la que se hace este cálculo, con un segundo bucle y una variable llamada **`encontrado`**.

- Las instrucciones de la segunda aproximación son fáciles de llevar a la práctica si disponemos de una implementación del grafo y de la lista con las operaciones habituales de:

  - iterar sobre los elementos de una lista
  - iterar sobre los vértices de un grafo
  - crear una lista vacía
  - añadir un elemento a una lista
  - saber si hay un arco entre dos vértices del grafo

4. Algoritmos recursivos

UNIVERSIDAD DE CANTABRIA

Muchos algoritmos iterativos pueden resolverse también con un algoritmo *recursivo*

- el algoritmo se invoca a sí mismo
- en ocasiones es más natural
- en otras ocasiones es más engorroso

El *diseño recursivo* consiste en diseñar el algoritmo mediante una estructura condicional de dos ramas

- *caso directo*: resuelve los casos sencillos
- *caso recursivo*: contiene alguna llamada al propio algoritmo que estamos diseñando

Ejemplos

UNIVERSIDAD DE CANTABRIA

Definición iterativa para el cálculo del *factorial* de un número natural

$$n! = 1 \quad , n = 0$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n, \quad n \geq 1$$

Definición recursiva para el cálculo del *factorial* de un número natural

$$n! = 1 \quad , n = 0$$

$$n! = n \cdot (n-1)!, \quad n \geq 1$$

La definición es correcta pues el número de recursiones es finito

Ejemplo: factorial recursivo

```
función factorial (n entero) retorna entero
   si (n<=1) entonces
      // caso directo
      retorna 1;
   si no
      // caso recursivo
      retorna n*factorial(n-1);
   fin si
fin factorial
```

Fases del diseño recursivo

UNIVERSIDAD DE CANTABRIA

Obtener una definición recursiva de la operación a implementar a partir de la especificación

- **Establecer caso directo**
- **Establecer caso recursivo**

Diseñar el algoritmo con una instrucción condicional

Argumentar sobre la terminación del algoritmo

Consideraciones sobre los datos

UNIVERSIDAD DE CANTABRIA

Datos compartidos por todas las invocaciones del algoritmo

- parámetros in out
- valores apuntados por punteros
- estado de otros objetos externos

Datos para los que cada invocación tiene una copia posiblemente distinta

- variables locales (internas) del algoritmo
- parámetros in o out
- valor de retorno de la función

5. Tiempo de ejecución de un algoritmo

UNIVERSIDAD DE CANTABRIA

Entre los criterios a tener en cuenta al diseñar o elegir un algoritmo están su complejidad, y su tiempo de ejecución

El tiempo de ejecución depende de factores variados y, muy en particular, del tamaño del problema

El tiempo de ejecución puede ser:

- exacto: es muy difícil de predecir;
  - ni siquiera midiéndolo
- predicción del ritmo de crecimiento del tiempo de ejecución con respecto al tamaño del problema
  - medida aproximada

Notas:

UNIVERSIDAD DE CANTABRIA

Al realizar la implementación de un programa, es preciso en muchas ocasiones buscar una solución de compromiso entre dos aspectos:

- El algoritmo debe ser fácil de entender, codificar y corregir.
- El algoritmo debe ser lo más rápido posible.

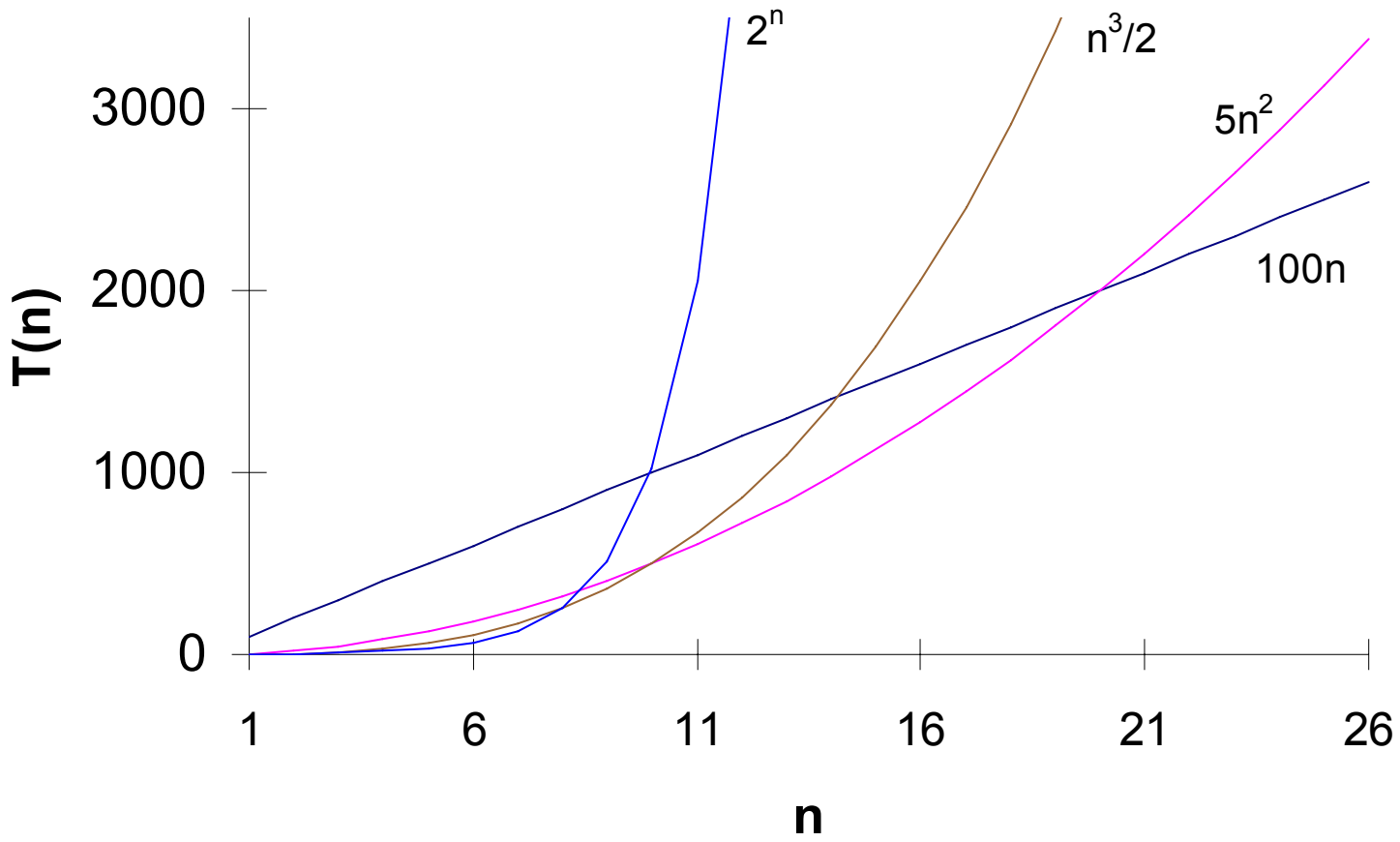
Cuando el programa se escribe para utilizarlo unas pocas veces, es más importante la primera premisa, ya que el tiempo y esfuerzo de programación es lo más costoso. Sin embargo, cuando un programa se va a utilizar a menudo el costo de la ejecución del programa es mayor que el de desarrollo, y se hace más importante la segunda premisa.

El tiempo exacto de ejecución de los programas es muy difícil de calcular, ya que depende de:

- Los datos de entrada al programa
- Calidad del código generado por el compilador
- Naturaleza y velocidad de las instrucciones de la máquina.
- Complejidad natural del algoritmo que soporta el programa.

Sin embargo, es más fácil de calcular la dependencia del tiempo con la cantidad de datos de entrada, es decir, el tamaño del problema.

La "tiranía" del ritmo de crecimiento



Notas:

UNIVERSIDAD DE CANTABRIA

La gráfica de arriba muestra un diagrama de diversos tipos de dependencias temporales con respecto a *n*, que es el número de datos a procesar por el programa, es decir, el tamaño del problema. Para pocos datos la curva de tipo lineal (100*n*) es peor que las otras, pero cuando el número de datos es alto, el tiempo de ejecución de las otras gráficas es prácticamente inabordable

De la gráfica anterior se desprende que el ritmo de crecimiento determina incluso el tamaño de los problemas que se pueden resolver con ese algoritmo sobre un computador. Incluso con computadores mucho más rápidos, si el ritmo de crecimiento es del tipo 2^n o una potencia entera de n alta (n^4, n^5, etc.), el problema es prácticamente irresoluble en un tiempo razonable:

- Por ejemplo, para la gráfica anterior con 25 datos, y suponiendo que los tiempos de ejecución están en milisegundos, los resultados son:

Función	n=25	n=50
100n	2.5 seg	5.0 seg
$5n^2$	3.12 seg	12.5 seg
$n^3/2$	7.81 seg	62.5 seg
2^n	33554.43 seg	35677 años

5. La notación $O(n)$

UNIVERSIDAD DE CANTABRIA

El tiempo de ejecución depende no sólo de la cantidad de datos (n) sino también de cuáles son los datos; por ello distinguimos:

- tiempo de peor caso: $T(n)$
- tiempo promedio: $T_{avg}(n)$

Para expresar los ritmos de crecimiento se suele usar la notación $O(n)$:

- decimos que $T(n)$ es $O(f(n))$ si existen constantes c y n_0 tales que $T(n) \leq c \cdot f(n)$ para todo $n \geq n_0$

La notación $O(n)$ muestra una cota superior al ritmo de crecimiento de un algoritmo

Notas:

El tiempo de ejecución se manifiesta habitualmente como una función del número de datos de entrada, **T(n)**, donde n es el número de datos.

Las unidades de T(n) serán inespecíficas, aunque pueden interpretarse como el número de instrucciones a ejecutar sobre un computador.

En muchos programas el tiempo de ejecución es función de la entrada particular y no sólo de su dimensión. Entonces T(n) será el tiempo de ejecución del **peor caso.** También se puede considerar el **tiempo de ejecución medio**, sobre todas las entradas posibles de dimensión n: $\mathbf{T_{avg}(n)}$. Sin embargo, la evaluación del tiempo de ejecución medio es normalmente muy difícil de realizar.

La notación utilizada para representar la función T(n) es generalmente la notación O(n).Se dice que T(n) es O(f(n)) si existen constantes positivas c y n_o tales que $T(n) \leq c \cdot f(n)$ para cualquier $n \geq n_o$.

Por ejemplo la función $T(n) = 3n^3+2n^2$ es $O(n^3)$. Basta hacer c=5 y $n_o=0$ para comprobarlo,

$$3n^3+2n^2 \leq 5n^3$$

En definitiva, cuando decimos que T(n) es O(f(n)), esto significa que f(n) es el límite de la velocidad de crecimiento de T(n).

La notación $\Omega(n)$

UNIVERSIDAD DE CANTABRIA

También se puede expresar un límite inferior al ritmo de crecimiento de $T(n)$ mediante la notación $\Omega(n)$:

- **decimos que $T(n)$ es $\Omega(g(n))$ si existe una constante c tal que $T(n) \geq c.g(n)$ para un número infinito de valores de n**

Recordar siempre que tanto $O(n)$ como $\Omega(n)$ son medidas relativas, no absolutas

Notas:

UNIVERSIDAD DE CANTABRIA

También se puede especificar un límite inferior para la función T(n) mediante la notación $\Omega(g(n))$, que significa que existe una constante c tal que $T(n) \geq c \cdot g(n)$ para un número infinito de valores de n. Por ejemplo, $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$. Basta probar para c=1 y n>0.

Al comparar los tiempos de ejecución de diversos algoritmos es preciso tener en cuenta el número de entradas **n** que se va a aplicar, pero también es preciso tener en cuenta que tanto las notaciones O(n) como $\Omega(n)$ no son absolutas.

Por ejemplo, supongamos dos algoritmos cuyos tiempos de ejecución son $O(n^2)$ y $O(n^3)$. En principio el segundo presenta un tiempo de ejecución peor. Sin embargo es preciso tener en cuenta las constantes de proporcionalidad debidas al compilador, la máquina y el propio algoritmo.

- Supongamos que estas constantes son $100n^2$ y $5n^3$. En este caso el segundo algoritmo será mejor para un número de datos n<20. Sin embargo, a medida que crece n, el tiempo de este algoritmo crece enormemente.

En definitiva, la búsqueda de algoritmos eficientes para resolver los problemas, es tan necesaria como la de disponer de computadores muy rápidos. Hay que tener en cuenta, sin embargo, que el ritmo de crecimiento en el peor caso no es el único, ni a veces el criterio más importante para evaluar un algoritmo o programa.

Cálculo del tiempo de ejecución de un programa

UNIVERSIDAD DE CANTABRIA

Regla de las sumas:

- si $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces
- $T_1(n)+T_2(n)$ es $O(\max(f(n),g(n)))$

Es decir, que la suma de los tiempos de ejecución de dos algoritmos tiene un ritmo de crecimiento igual al del máximo de los dos

Por ejemplo, para tiempos polinómicos $T(n)=an^p+bn^{p-1}+...$

- entonces $T(n)$ es $O(n^p)$

Notas:

UNIVERSIDAD DE CANTABRIA

En ocasiones, determinar el tiempo de ejecución de un programa con un factor constante de diferencia, no es una tarea excesivamente difícil utilizando la notación O(n).

Reglas de las sumas: Suponer que Ta(n) y Tb(n) son los tiempos de ejecución de dos fragmentos de programa A y B, y que Ta(n) es O(f(n)) y Tb(n) es O(g(n)). Entonces el tiempo de ejecución de A seguido del de B, Ta(n)+T(b(n), viene dado por,

O(**max**(f(n),g(n))

Para ver la razón puede observarse que para algunas constantes ca, cb, na, nb,

si $n \geq na \Rightarrow Ta(n) \leq ca \cdot f(n)$ y si $n \geq nb \Rightarrow Tb(n) \leq cb \cdot g(n)$

suponiendo que $n \geq n_0 = \max(na,nb)$ entonces $Ta(n)+Tb(n) \leq ca \cdot f(n)+cb \cdot g(n)$. Por tanto se concluye que,

si $n \geq n_0 \Rightarrow Ta(n)+Tb(n) \leq (ca+cb)\max(f(n),g(n))$

Es decir que cuando se ejecuten, secuencialmente, varias partes de un programa y cada una de las partes tiene una O(n) distinta, la ejecución secuencial de ambas será de la forma O(del máximo).

Cálculo del tiempo de ejecución de un programa (cont.)

UNIVERSIDAD DE CANTABRIA

Regla de los productos:

- si $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces
- $T_1(n) \cdot T_2(n)$ es $O((f(n) \cdot g(n)))$

Es decir, que el producto de los tiempos de ejecución de dos algoritmos (p.e. cuando uno está anidado en el otro), tiene un ritmo de crecimiento igual al producto de los dos

Por ejemplo si $T(n)$ es $O(c \cdot f(n))$ entonces también es $O(f(n))$, ya que c es $O(1)$

Notas:

UNIVERSIDAD DE CANTABRIA

Regla de los productos:

La regla de los productos se plantea en términos parecidos, es decir si Ta(n) y Tb(n) son de la forma O(f(n)) y O(g(n)) respectivamente, entonces Ta(n)Tb(n) es O(f(n)g(n)).

La prueba es totalmente similar al caso anterior. Cabe señalar, a partir de esta regla que O(cf(n)), siendo c cualquier constante positiva, significa lo mismo que O(f(n)). Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Ritmos de crecimiento más habituales

UNIVERSIDAD DE CANTABRIA

1. $O(1)$, o constante
2. $O(\log(n))$, o logarítmico
3. $O(n)$, o lineal
4. $O(n \cdot \log(n))$
5. $O(n^2)$, o cuadrático
6. $O(n^x)$, o polinómico
7. $O(2^n)$, o exponencial

Notas:

UNIVERSIDAD DE CANTABRIA

La tabla que aparece a continuación muestra valores para los ritmos de crecimiento más habituales. Puede observarse que para valores de N grandes, la dependencia logarítmica muestra tiempos muy bajos. La dependencia del tipo n·log(n) no crece demasiado con respecto al caso lineal. En cambio, las dependencias de tipo polinómico crecen muy deprisa, y las dependencias exponenciales son intratables.

n	1	$\log(n)$	$n \cdot \log(n)$	n^2
128	1	7	896	16384
256	1	8	2048	65536
512	1	9	4608	262144
1024	1	10	10240	1048576
2048	1	11	22528	4194304
4096	1	12	49152	16777216
8192	1	13	106496	67108864
16384	1	14	229376	268435456
32768	1	15	491520	1073741824

Pistas para el análisis

UNIVERSIDAD DE CANTABRIA

Instrucciones simples (asignación y op. aritméticas): O(1)

Secuencia de instrucciones simples: O(max(1,1,1)) = O(1)

Instrucción condicional:

- **si es un "if" simple y no se conoce el valor de la condición, se supone que la parte condicional se ejecuta siempre**
- **si es un "if" con parte "else" y no se conoce el valor de la condición, se elige la que más tarde de las dos partes**

Lazo: número de veces, por lo que tarden sus instrucciones

Procedimientos recursivos: número de veces que se llama a sí mismo, por lo que tarda cada vez

Notas:

UNIVERSIDAD DE CANTABRIA

Es de señalar que no existen reglas generales y por tanto, aunque en ocasiones es relativamente simple encontrar un límite superior del tiempo de ejecución de un programa, también puede ser una tarea sumamente compleja.Algunas reglas que se deben aplicar con cautela son:

1. El tiempo de ejecución de cada asignación, lectura o escritura es O(1). Sólo en raras ocasiones pueden realizarse estas operaciones sobre arrays arbitrariamente grandes.
2. El tiempo de ejecución de una secuencia de instrucciones viene determinado por la regla de las sumas. Esto es, el tiempo de ejecución de la secuencia es (sin tener en cuenta un factor constante) el de la instrucción de la secuencia que más tarda.
3. El tiempo de ejecución de una instrucción 'if' es el costo correspondiente a la ejecución de las instrucciones condicionales, más el tiempo para evaluar la condición O(1). En el caso de la construcción 'if-then-else' corresponde al tiempo máximo entre el de la condición cierta y falsa.
4. El tiempo para ejecutar un lazo es la suma, sobre el número de veces que se ejecuta, del tiempo empleado por el cuerpo, más el tiempo empleado en evaluar la condición, que usualmente es O(1). Despreciando factores constantes, este tiempo es, a menudo, el producto del número de veces que se ejecuta por el tiempo para la ejecución de su cuerpo.
5. Cuando existen procedimientos en nuestro programa, se comienza la evaluación por aquellos que no llaman a otros procedimientos, es decir por la parte más interna.
6. Cuando el procedimiento es recursivo, lo usual es considerar un tiempo desconocido T(n) para el procedimiento (donde n es la medida de la dimensión de los argumentos de llamada) y a continuación tratar de encontrar una función de recurrencia para T(n).

6. Ejemplo de análisis

UNIVERSIDAD DE CANTABRIA

Analizar el tiempo de ejecución del siguiente algoritmo:

```
   procedure Burbuja(n : in Positive;
      A :in out array (Positive range <>) of Integer) is
         temporal : Integer;
   begin
(1)     for i in 1..n-1 loop
(2)        for j in reverse i+1..n loop
(3)           if A(j-1) > A(j) then
                 -- intercambia A(j-1) y A(j)
(4)              temporal := A(j-1);
(5)              A(j-1) := A(j);
(6)              A(j) := temporal;
              end if;
           end loop;
        end loop;
      end Burbuja;
```

Notas:

UNIVERSIDAD DE CANTABRIA

Considerar el procedimiento 'burbuja' de arriba que ordena un array de enteros en orden creciente. En este ejemplo, el nº de elementos a ordenar es la medida de la dimensión de entrada. Vamos a analizar el programa desde el interior hacia el exterior,

- Cada instrucción de asignación toma una cantidad de tiempo constante, independiente de la dimensión de entrada, es decir, cada instrucción (4), (5) y (6) toma un tiempo de la forma O(1) y por tanto por la regla de la suma, la ejecución secuencial de este grupo de instrucciones es de la formaO(max(1,1,1))=(1).
- Igualmente ocurre con la instrucción **if** y aunque no conocemos cuando se ejecuta su cuerpo, podemos suponer el pero caso, es decir siempre, pero nuevamente eso sólo implica que el grupo de instrucciones (3)-(6) toma un tiempo O(1).
- El lazo que comienza en la línea (2) y abarca hasta la línea (6) tiene un cuerpo que toma un tiempo de la forma O(1) en cada iteración, y como toma n-i iteraciones, el tiempo gastado en ese lazo es O((n-i)·1) que es O(n-i).
- El último lazo que contiene todas las instrucciones ejecutables del programa, se ejecuta n-1 veces, de forma que el tiempo total de ejecución esta limitado superiormente por:

$$\sum_{i=1}^{n-1} (n-i) = \frac{1}{2}n(n-1) = \frac{n^2}{2} - \frac{n}{2}$$

que es $O(n^2)$.

Ejemplo de análisis recursivo

UNIVERSIDAD DE CANTABRIA

```
function Factorial (n : in Natural)
                  return Natural is
  begin
(1)          if n <= 1 then
(2)              return 1;
             else
(3)              return n*Factorial(n-1);
             end if;
end Factorial;
```

La función se llama a sí misma n veces, y cada ejecución es O(1), luego en definitiva factorial es O(n)

Notas:

UNIVERSIDAD DE CANTABRIA

Suponemos T(n) el tiempo de ejecución para FACTORIAL(n). El tiempo de ejecución de las líneas (1) y (2) es O(1) y para la línea (3) es O(1)+T(n-1). Por tanto para algunas constantes c y d,

- $T(n)=c+T(n-1)$ si $n>1$
- $T(n)=d$ si $n \leq 1$

si hacemos lo mismo para n>2, n>3, ..,n>i, tendremos,

- $T(n) = i \cdot c + T(n-i)$ si $n > i$

por último, cuando i=n-1,

- $T(n) = c(n-1) + T(1) = c(n-1) + d$

y por tanto T(n) es del tipo O(n).