

Algoritmos

Indice

- Definición y propiedades
- Representación
- Algoritmos alternativos y equivalentes
- Programación estructurada
- Pseudocódigo
- Análisis de algoritmos
- Desarrollo
- Algoritmos iterativos
- Algoritmos recursivos
- Colección de algoritmos

Definición

- **Algoritmo** es una secuencia ordenada de instrucciones que resuelve un problema concreto
- Ejemplos:
 - Algoritmo de la media aritmética de N valores.
 - Algoritmo para la resolución de una ecuación de segundo grado.
- Niveles de detalle de los algoritmos:
 - Alto nivel: no se dan detalles.
 - Bajo nivel: muchos detalles.

Algoritmos en la vida diaria₁

- **Receta (Algoritmo) de cocina:** Camarones con mayonesa de albahaca.
- Ingredientes - **Input** (datos de entrada o iniciales)
- Resultado u **Output**



Algoritmos en la vida diaria₂

Ingredientes (para 4 porciones):

- 2 cucharadas de aceite de oliva
- ½ cucharadita de ajo picado fino
- 2 tazas de camarones medianos, sin caparazón y desvenados
- 1 cucharadita de sal
- ½ cucharadita de pimienta
- 2 cucharadas de albahaca picada

Mayonesa de albahaca:

- 1 taza de mayonesa baja en grasa
- 4 cucharadas de albahaca
- 3 pimientas negras molidas
- ½ limón, el jugo

Guarnición:

- 2 aguacates grandes, maduros en rebanadas
- 1 limón en cuartos
- 4 hojas de lechuga escarola
- 8 jitomates cherry partidos por la mitad
- Salsa picante al gusto

Algoritmos en la vida diaria₃

Preparación: (Algoritmo)

1. En una sartén caliente **agrega** el aceite de oliva, **agrega** el ajo, camarones, sal, pimienta y albahaca. **Cocínalos** 1 minuto. **Apaga** el fuego y deja los camarones un minuto más. Pasa los camarones a un tazón que esté sobre agua con hielo, déjalos dentro solamente el tiempo necesario para que se **enfríen**.
2. En un tazón **mezcla** todos los ingredientes de la mayonesa albahaca.
3. **Corta** los aguacates en rebanadas y acompaña los camarones. **Sirve** en un plato decorado con hojas de lechuga, tomates cherry, limón y si lo deseas agrega un poco de salsa picante.

Propiedades

Necesarias o básicas:

- Corrección (sin errores).
- Validez (resuelve el problema pedido)
- Precisión (no puede haber ambigüedad).
- Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).
- Finitud (termina en algún momento). Número finito de órdenes no implica finitud.
- Eficiencia (lo hace en un tiempo aceptable)

Deseables:

- Generalidad
- Fácil de usar
- Robustez

Representación Verbal

- **Verbal:** usa oraciones del lenguaje natural.

Pago Bruto

Si las horas trabajadas son menores o iguales a 40, el pago es el producto del número de horas trabajadas y la tarifa (100 €/hora). Si se ha trabajado más de 40 horas, el pago es de 150 (50% más de la tarifa normal) por cada hora en exceso a 40.

Si horas trabajadas = 25

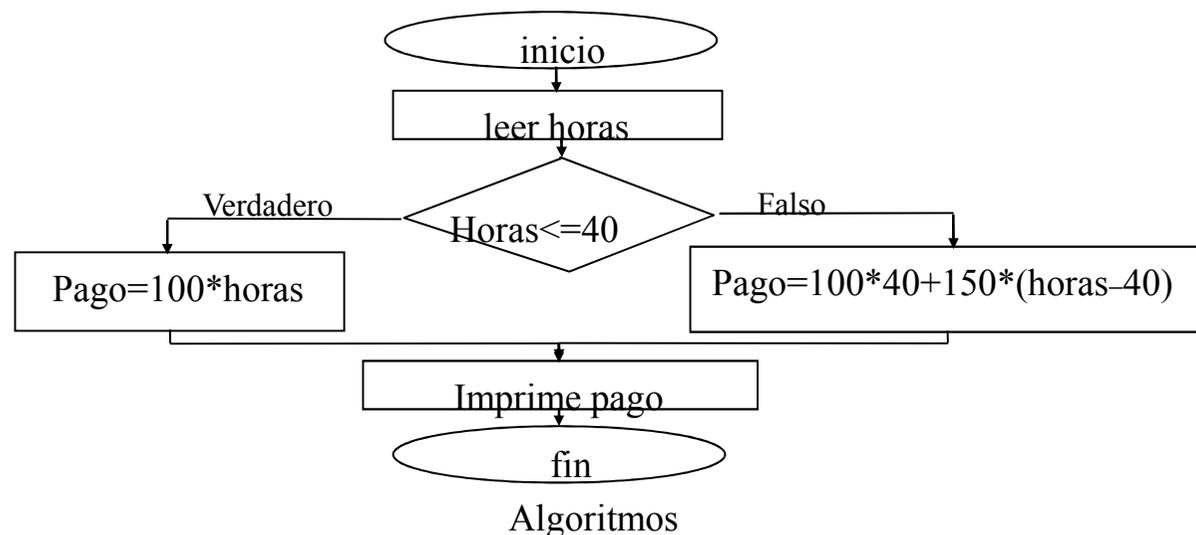
$$\text{pago} = 100 \times 25 = 2500$$

Si horas trabajadas = 50

$$\text{pago} = 100 \times 40 + 150 \times (50 - 40) = 5500$$

Representación Diagramas de flujo

- **Diagramas de flujo:** Representación gráfica mediante cajas conectadas con flechas. Símbolos habituales:
 - *Cajas ovales:* indican inicio y fin.
 - *Cajas rectangulares:* representan acciones
 - *Rombos:* representan decisiones a tomar. Contiene la condición que determina la dirección a seguir.

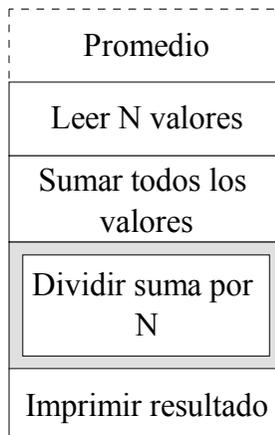


Representación Diagramas de cajas

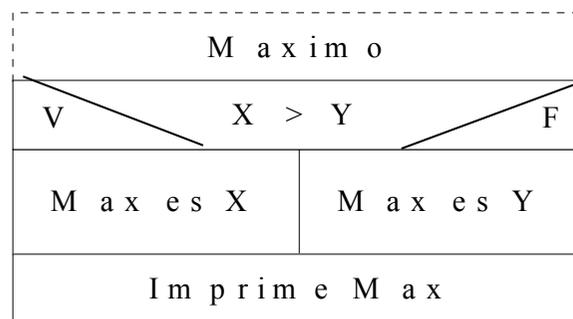
- **Diagramas de Bloques, Cajas o de Nassi-Shneiderman**

Ventaja principal: **no** usan flechas. Básicamente se trata de cajas rectangulares que se apilan de acuerdo al algoritmo.

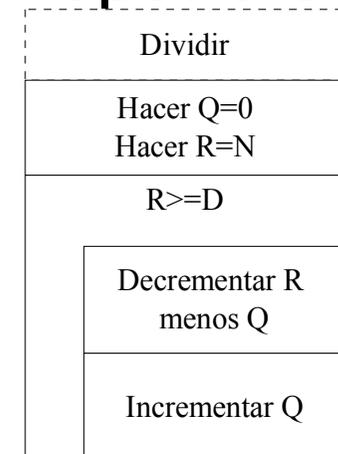
Secuencia



Selección



Repetición



Representación NS EasyCode

- **Diagramas de Bloques, Cajas o de Nassi-Shneiderman**

Programa que soporta NS: **EasyCode**

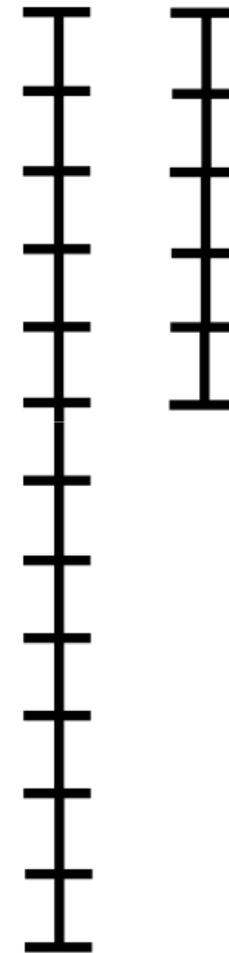
http://www.easycode.de/v8_cpp.html?&L=1

- genera código a partir del diagrama de cajas.
- a partir del código genera el diagrama de cajas.

Ejemplo NS : Máximo común divisor

- **Algoritmo de Euclides:**
formulación geométrica del problema del MCD: encontrar una "medida" común para el largo de dos líneas.

El algoritmo consiste en **repetir** la resta de la línea más corta del otro segmento *mientras* las líneas sean diferentes.



Ejemplo MCD con Easy Code

EasyCODE(C++) Demo - [(Untitled) * - Top:Algoritmo MCD (Eu...

File Edit Insert View Add-In Options Window ?

Toolbox

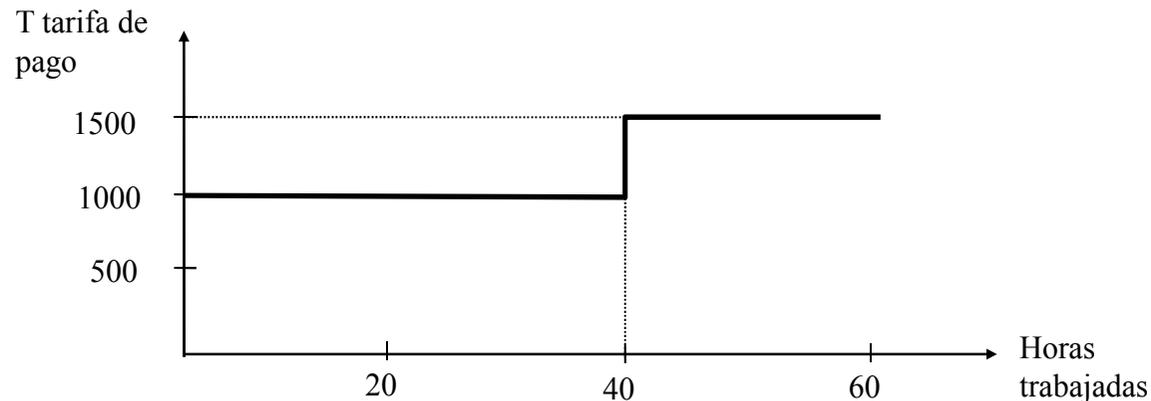
Algoritmo MCD (Euclides)

Input Leer A y B	
while (A != B)	
if (A > B)	
then	else
A = A - B	B = B - A
Output Imprimir A	

Ready W x H 1 x 1 NUM

Representación Gráficos y Pseudocódigo

- **Gráficos:**



- **Pseudocódigo:** descripciones cortas que usan una mezcla de matemáticas, lógica y lenguaje natural.

Leer horas

Si horas > 40

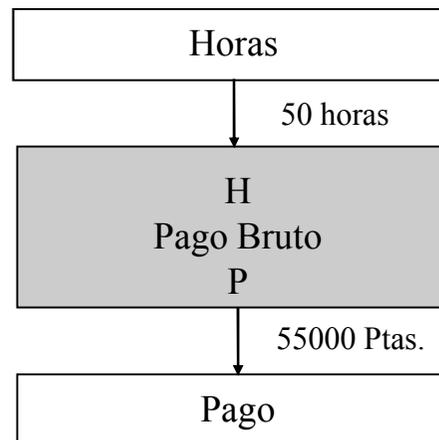
$$\text{Pago Bruto} = \text{Tarifa} \times 40 + 150 \times (\text{Horas} - 40)$$

sino

$$\text{Pago Bruto} = 100 \times \text{Horas}$$

Representación Flujo de datos y tabulares

- **Diagramas de flujo de datos:** Representación gráfica de alto nivel que muestran los datos de entrada y de salida. Indican *qué* es lo que hace y los diagramas de flujo o de cajas indican *cómo* se hace.



- **Representaciones tabulares** (tablas, arrays y matrices). Son útiles para resumir colecciones de datos grandes.

Representación matemática

- **Representaciones algebraicas (fórmulas y expresiones).** En ingeniería y matemáticas, los algoritmos se expresan como fórmulas o expresiones algebraicas. Usualmente es una representación muy concisa.

Media y Varianza de una serie de números:

$$M = (x_1 + x_2 + x_3 + \dots + x_N) / N$$

$$V = [(x_1 - M)^2 + (x_2 - M)^2 + \dots + (x_N - M)^2] / N$$

Factorial de un número:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Fórmula del seno:

$$\text{seno}(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

Modificación de algoritmos

- ***Generalización y extensibilidad***: proceso de aplicar el algoritmo a más casos y de incluir más casos dentro del algoritmo
- ***Robustez***: proceso de hacer un algoritmo mas fiable o robusto (se recupera de errores), anticipando errores de entrada u otras dificultades.

Algoritmos alternativos y equivalentes

- Pueden haber muchas formas de llevar a cabo un algoritmo.
- En esos casos la elección se basa en la eficiencia (memoria y velocidad).
- El *análisis de algoritmos* estudia la cantidad de recursos que demanda la ejecución de un algoritmo.
- Preocupa más el tiempo de ejecución de un algoritmo: *Complejidad del algoritmo*

Programación estructurada

- Método para construir algoritmos a partir de un número pequeño de bloques básicos.
- Formas fundamentales:
 - ***Secuencia***: indica secuencia temporal lineal de las acciones a realizarse.

A

B

- ***Selección***: especifica una condición que determina la acción a realizarse.

if C

D

else

E

Programación estructurada

- *Repetición*: indica que una o más acciones deben repetirse un determinado número de veces.

```
while G do  
    H
```

- *Invocación*: corresponde al grupo de acciones agrupadas bajo un nombre.

```
Calcula_promedio
```

Pseudocódigo

- Lectura o entrada de datos

Input

- Repetición

while expr
instrucción
endwhile

for i = 1 to m (en C: for(i=1;i<=m;i++))
instrucción
endfor

do
instrucción
while expr

Pseudocódigo

- Decisión
 - if** expr
 - instrucción
 - endif**
- Escritura o salida de datos
 - Output**

Análisis de algoritmos

Problema: Buscar el mayor valor en una lista de números desordenados (array)

Algoritmo: (n = número de elementos)

```
1  max =  $S_1$ 
2  i = 2
3  while i <= n
4      if  $S_i > \text{max}$  then
5          max =  $S_i$ 
6          i = i + 1
7  endwhile
```

Análisis de algoritmos

Número de operaciones realizadas (unid):

Línea	Operaciones	Tiempo
1	indexado y asignación	2
2	asignación	1
3	comparación	1
4,5,6	2 indexado, comparación, 2 asignación, suma	6

Tiempo total:

$$t(n) = 2 + 1 + (n - 1) + 6 \cdot (n - 1) = 3 + 7 \cdot (n - 1) = 7n - 4$$

Notación asintótica

- Es útil concentrarse en la tasa de crecimiento del tiempo de ejecución t como función del tamaño de la entrada n .
- Se usa la notación **O grande** (cota superior al ritmo de crecimiento de un algoritmo):

Sean $f(n)$ y $g(n)$ funciones no negativas, $f(n)$ es $O(g(n))$ si hay un valor $c > 0$ y

$n_0 \geq 1$ tal que $f(n) \leq cg(n)$ para $n \geq n_0$

- Se dice que $f(n)$ es de orden $g(n)$
- Ej: $7n - 4$ es $O(n)$ si $c=7$ y $n_0 = 1$
- Generalmente para cualquier polinomio $a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ es $O(n^k)$

Experimento de Algoritmo

- Contar el número de personas en una sala
 1. Ponerse de pie.
 2. Pensar para sí mismo “Soy el N° 1”
 3. Emparejarse con alguien; sumar sus números; tomar la suma como su nuevo número.
 4. Uno de la pareja debe sentarse.
 5. Ir al paso 3 si hay alguien de pie.

Desarrollo de un Algoritmo

- Problema:
 - Se desea viajar de la estación A a la estación B de un metro.



Ejemplo - Algoritmo Metro

Algoritmo Metro

Input A, B [las dos estaciones]

if A y B están en la misma línea L

then

viajar en la línea L de A a B

else [A y B están en diferentes líneas]

hallar todos los pares de líneas (L_i, L_j) tal que A es una estación de L_i y B es una estación de L_j y las dos líneas tienen una estación en común.

if hay solo un par

then

tomar L_i hacia una estación también en L_j y después tomar L_j a la estación B

else [más de un par posible]

usar par (L_i, L_j) que tome menos paradas

endif

endif

Output Llegada a la estación B

Observaciones

- Qué hacer si dos pares de líneas tienen el mismo número de paradas.
- Qué hacer si hay líneas con más de una estación en común.
- Qué pasa si el cambio de una estación a otra requiere una larga caminata.
- Formato de algoritmo para humanos (resuelven ambigüedades rápidamente) NO para un ordenador (sigue instrucciones precisas).

Desarrollo de algoritmos

- Análisis del problema
 - Dominio del problema
 - Modelo
- Diseño del algoritmo
 - Refinamiento sucesivo
 - Top down o botton up
- Análisis del algoritmo
 - Cuánto tarda en dar una solución? Se puede modificar para aumentar la eficiencia?
 - Análisis de la Complejidad
- Verificación del algoritmo
 - Comprobar que es correcto

Algoritmos iterativos

- Muchos algoritmos se basan en *ciclos o bucles*, es decir en la ejecución de una serie de pasos repetitivos.
- **Iteración** significa hacer algo de forma repetida.

Algoritmos iterativos

Ejemplo: Multiplicación de dos enteros considerando que sólo está disponible la operación suma

Algoritmo Mult

Input x [Entero ≥ 0]
 y [cualquier entero]
prod = 0; u = 0 [Inicialización]
while u < x
 prod = prod + y
 u = u + 1
endwhile
Output prod

Verificación Algoritmos Método Traza

Input $x = 3$

$y = 4$

$prod = 0$

$u = 0$

while $u < x$		T		T		T		F
$prod = prod + y$		4		8		12		
$u = u + 1$		1		2		3		

endwhile

Output $prod$ 12

Recursión

- **Recursión** mecanismo de repetición que expresa una solución como función de la solución de un problema de menor tamaño.

Ejemplo: Suma de una lista de números ($a_i, i=1, \dots, n$).

$$\text{Sum}(a_i, i=1, \dots, n) = a_n + \text{Sum}(a_i, i=1, \dots, n-1)$$

$$\text{si } i=1 \text{ Sum} = a_1$$

Input $a_i, i=1, \dots, n$

SumaListaRec(n)

if $n=1$ **then** $\text{sum} = a_1$

else $\text{sum} = \text{sum} + \text{SumaListaRec}(n-1)$

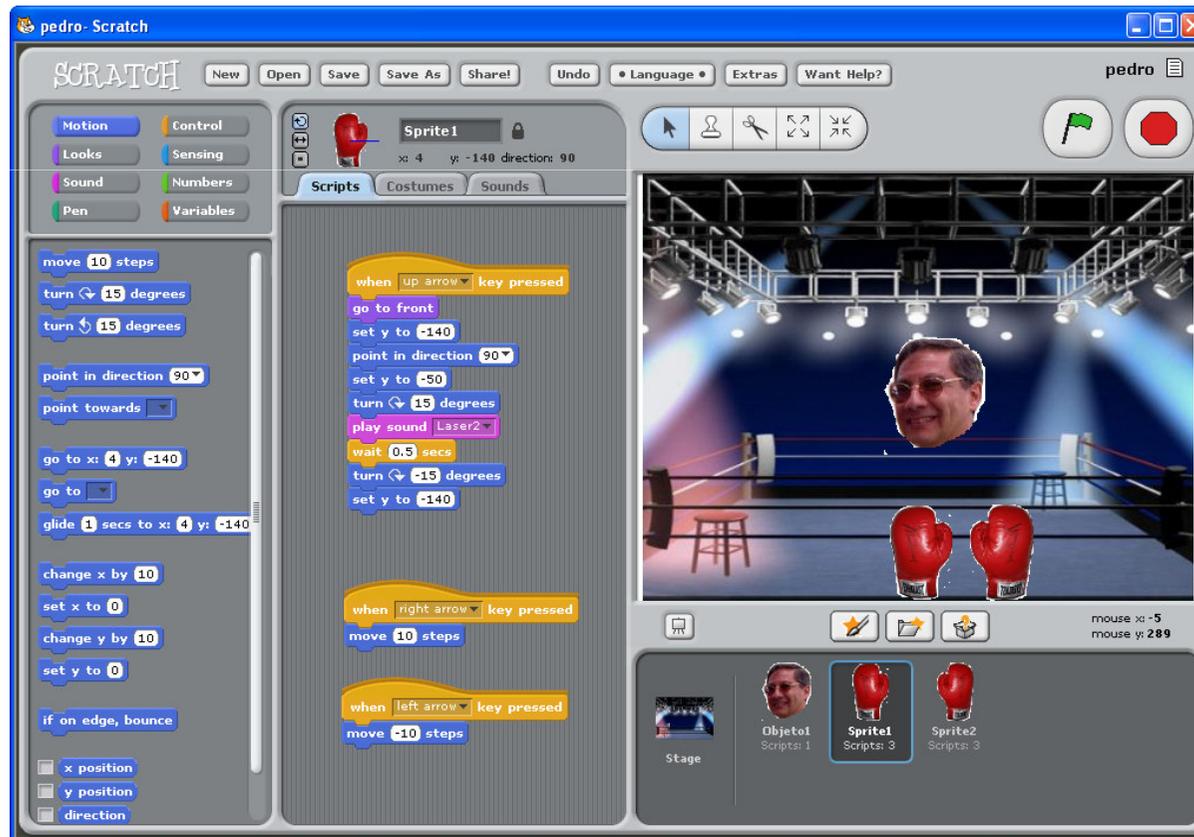
endif

return sum

Scratch

- Programa gratuito para aprender a programar de forma divertida (indicado para niños)

<http://scratch.mit.edu/>



Colección de Algoritmos

Algoritmos iterativos

- Ejemplo 1: Suma de una lista de números.

Lista con elementos individuales indicados por: a_i , $i=1, 2, \dots, n$ (donde n es el número de elementos de la lista)

Input n, a_i

sum = a_1

for $i=2$ **to** n [C:for($i=2$; $i \leq n$; $i++$)]

 sum = sum + a_i

endfor

Output sum

Algoritmos iterativos

- Ejemplo 2: Hallar el entero positivo n tal que $2^n \leq N$ utilizando sumas y multiplicaciones

Input N

$n = 0$

$pot2 = 2$

while $pot2 \leq N$

$n = n + 1$

$pot2 = pot2 * 2$

endwhile

Output n

Ejercicios

- Repetir el ejemplo 1 sin usar for
- Escribir un algoritmo para calcular $n!$ ($= n \times n-1 \times n-2 \dots 2 \times 1$) colocando el resultado en la variable fact
- Calcular x^m siendo m una potencia positiva de 2
- Suponer que m no es una potencia de 2. Usar el algoritmo anterior y completar el cálculo de x^m
- Dada una lista de enteros n_1, n_2, \dots, n_m y un entero p , contar el número de pares (n_i, n_j) , $i \neq j$ tal que $n_i = p * n_j$ o $n_j = p * n_i$

Solución a ejercicios

- Ej. 1
Input a_i $i=1, \dots, n$
sum = 0
 $i = 1$
while $i \leq n$
 suma = suma + a_i
 $i = i + 1$
endwhile
Output suma

Solución a ejercicios

- Ej. 2

Input n

fact = 1

i = 2

while i <= n

 fact = fact * i

 i = i + 1

endwhile

Output fact

Solución a ejercicios

- Ej. 3

Input x, m

power = x

cont = 1

while cont < m

 power = power * power

 cont = 2 * cont

endwhile

Output power

Solución a ejercicios

- Ej. 4
Input x, m
power = x
cont = 1
while 2*cont <= m
 power = power * power
 cont = 2 * cont
endwhile
while cont < m
 power = power * x
 cont = cont + 1
endwhile
Output power

Solución a ejercicios

- Ej. 5

Input n_i $i=1, \dots, m$, p

$cont = 0$

for $i=1$ **to** m [C:for($i=1$; $i \leq m$; $i++$)]

for $j=i+1$ **to** m

if $n_i = p * n_j$ **or** $n_j = p * n_i$

$cont = cont + 1$

endif

endfor

endfor

Output $cont$

Ejercicios algoritmos recursivos

- Escribir un algoritmo recursivo para calcular $n!$
- Escribir un algoritmo recursivo para calcular x^m

Solución a ejercicios

- Ej. 1

Factorial(n)

if n = 1 **then** fact = 1

else

fact = n * Factorial(n-1)

endif

return fact

Solución a ejercicios

- Ej. 2

Powerofx(n)

if n = 1 **then** power = x

else

 power = x * Powerofx(n-1)

endif

return power

Teoría de números

- Generación de números primos
 - Como todos los primos excepto 2 son números impares, tratamos 2 como un caso especial.
 - Probar todos los números impares divisibles por números impares menores al que se prueba. (No es muy eficiente).
 - Por eficiencia sólo se prueban números menores o iguales a la raíz cuadrada del número que se prueba.
 - Criterio de divisibilidad:
Resto de división entera (operador %) es 0.

Números Primos

Input N [Núm. de primos a calcular]

Algoritmo NumPrimo

$p_1 = 2$ [primer primo p_1]; $n = 2$ [n indica el p_n primo]

$i = 3$ [número a probar]

while $n \leq N$

$j = 3$ [primer divisor]

while $j < \sqrt{i}$

if $i \% j = 0$ **then exit** [es divisor]

$j = j + 2$ [siguiente divisor]

endwhile

if $j = i$ **then** $p_n = i$ [es siguiente primo]

$n = n + 1$

endif

$i = i + 2$ [sig. entero a probar]

endwhile

Output p_n , $n=1, \dots, N$

Máximo común divisor

- **Algoritmo de Euclides** para hallar el máximo común divisor (mcd) de dos números enteros no negativos.
 - Sean m y n dos enteros. El $\text{mcd}(m,n)$ se basa en la ecuación $m = nq + r$ donde
 - q es el cociente entero $q \geq 0$ y
 - r es el resto $0 \leq r < n$
 - $\text{mcd}(m,n) = \text{mcd}(n,r)$ cuando r es cero el mcd es n .

Máximo común divisor

Input m, n

Algoritmo MCD

num = m

denom = n

while denom \neq 0

 resto = num%denom

 num = denom

 denom = resto

endwhile

Output num

Máximo común divisor

Como función:

Algoritmo MCD(m, n)

num = m

denom = n

while denom \neq 0

 resto = num%denom

 num = denom

 denom = resto

endwhile

return num

Mínimo común múltiplo

- El mínimo común múltiplo (mcm) de dos números enteros positivos m y n se puede calcular usando:

$$\text{mcd}(m,n) * \text{mcm}(m,n) = m * n$$

Input m, n

Algoritmo MCM

$\text{num} = \text{MCD}(m,n)$

$\text{mcm} = m * n / \text{num}$

Output mcm

Partición de enteros

- Una partición de un número entero positivo n es cualquier conjunto de enteros positivos cuya suma es n .

- Ejemplo: 4 tiene 5 particiones

4
3 1
2 2
2 1 1
1 1 1 1

- **Problema:** dado un número entero n , generar todas las particiones de n , empezando con n y en *orden lexicográfico*.

Partición de enteros (I)

- Dos particiones están en *orden lexicográfico* si, empezando por la izquierda, en el primer dígito en el que difieren, el dígito en el primero es mayor que en la segunda.
- Debemos determinar, dada una partición P de n , la siguiente en orden lexicográfico. Casos:
 - El último dígito de P no es 1. Entonces se reduce el último dígito en 1 y se añade uno al final.
 - El último dígito es 1. Se realiza lo siguiente:
 - Empezando por la derecha, contar el número de 1s, asignar a c , hasta conseguir un dígito d mayor a 1.
 - Reducir d a $d - 1$ lo que significa que se tiene $c+1$ para añadir al final de la partición usando dígitos no mayores a $d - 1$
 - Dividir de forma entera $c+1$ por $d-1$ (resultado q y r). Añadir q ($d - 1$)s seguidos de un r (si $r > 0$) al final.

Partición de enteros (Alg.)

Input n

Algoritmo Particiones

$p_1 = n$

$k = 1$

do

Output $p_m, m=1,..k$

if $p_1 = 1$ **exit**

if ($p_k \neq 1$) **then** $p_k - = 1$
 $p_{k+1} = 1$

else

$c = 0; i = k$

while ($p_i == 1$)

$c++$

$i--$

endwhile

Partición de enteros (Alg.)

```
pi --  
q = (c+1) / pi  
r = (c+1) % pi  
j = 1  
while (j <= q)  
    pi+1 = pi  
    i ++  
    j ++  
endwhile  
if (r = 0) then k = i  
else k = i + 1  
    pk = r  
endif  
endif  
while (1)
```

Algoritmos del álgebra

- Ecuación de segundo grado

$$ax^2 + bx + c = 0$$

solución:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Casos especiales:

$$a=0, b=0, c=0$$

Discriminante $b^2 - 4ac$ negativo (raíces complejas conjugadas)

Ec. de segundo grado

Input a, b, c

Algoritmo Ec2grado

if a = 0 **then**

if b = 0 **then**

if c = 0 **then** **Output** “Ec. Vacía”

else **Output** “Ec. Falsa”

endif

else

$x = -c/b$

Output “Raíz real:”, x

endif

else

$disc = b^2 - 4ac$

Ec. de segundo grado (cont)

if disc < 0 **then**

real = $-b/2a$ [Parte real raiz]

imag = $\sqrt{-\text{disc}}/2a$ [Parte imag raiz]

Output “R. complejas”, real, imag

else

$x_1 = (-b - \sqrt{\text{disc}})/2a$

$x_2 = (-b + \sqrt{\text{disc}})/2a$

Output “Raices reales”, x_1 , x_2

endif

endif

Evaluación de polinomios

- La expresión general de un polinomio es:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

- Forma compacta de un polinomio: $P_n(x) = \sum_{i=0}^n a_i x^i$

- **Problema:** dados los coeficientes $a_n, i=1, \dots, n$ y un valor x evaluar $P_n(x)$

- Algoritmo de Horner

$$P_n(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \dots x \cdot (a_n) \dots))$$

requiere menos multiplicaciones

Evaluación de polinomios

Input $n, a_0, a_1, \dots, a_n, x$

Algoritmo Evalpoli

sum = a_n

for $k = n-1$ **downto** 0

 sum = sum $\cdot x + a_k$

endfor

Output sum

Multiplicación de polinomios

$$P_n(x) = \sum_{i=0}^n a_i x^i$$

$$Q_m(x) = \sum_{i=0}^m b_i x^i$$

Input $n, a_0, a_1, \dots, a_n; m, b_0, b_1, \dots, b_m$

Algoritmo Multpoli

for $k = 0$ **to** $m+n$

$c_k = 0$

$i = 0$

$j = k$

while $i \leq n$ and $j \geq 0$

if $j \leq m$ **then**

$c_k = c_k + a_i b_j$

endif

$i = i + 1; j = j - 1$

endwhile

endfor

Output $c_k, k=0, 1, \dots, m+n$

Permutaciones

- Permutación aleatoria de $S = \{1, 2, \dots, n\}$.

Input n

Algoritmo Permutar

Flag = 0 [Iniciar todos los comp. a 0]

for i = 1 **to** n

do

[Rand: Genera numeros aleatorios]

r = Rand[1, n]

while Flag(r) != 0

Perm(i) = r

Flag(r) = 1

endfor

Output Perm

Secuencia Fibonacci

- Una secuencia (conjunto ordenado de números) útil es la de Fibonacci definida por:

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = f_1 = 1$$

Input N

Algoritmo `Fibon_iter`

$$f_0 = 1$$

$$f_1 = 1$$

for `i = 2 to N`

$$f_i = f_{i-1} + f_{i-2}$$

endfor

Output f_N

Secuencia Fibonacci

- Versión recursiva (ineficiente!)

Input n

Algoritmo Fibonacci

```
function Fib(n)
```

```
if n = 0 or n= 1
```

```
    then Fib = 1
```

```
    else Fib = Fib(n-1) + Fib(n - 2)
```

```
endif
```

```
return Fib
```

```
endfunction
```

respuesta = Fib(n)

Output respuesta

Máximo y mínimo

- Sea $S = \{s_1, s_2, \dots, s_n\}$ un conjunto de n elementos donde cada elemento es un número real. Se desea hallar el máximo (mínimo) número en el conjunto.

Input n, s_1, s_2, \dots, s_n

Algoritmo MaxNum

max = s_1

for $i = 2$ **to** n

if $s_i > \text{max}$ **then** max = s_i

endfor

Output max

Análisis del algoritmo

Número de operaciones realizadas:

```
1 max = s1
2 for i = 2 to n
3     if si > max then
4         max = si
endfor
```

- Línea 1: dos operaciones primitivas (indexado y asignación) → 2 unid
- Línea 2: la instrucción for inicializa contador → 1 unid y compara → n-1 unid.
- Líneas 3 y 4: Se ejecutan n-1 veces en el peor de los casos 6 unid (indexado, comparación, indexado, asignación, suma y asignación).
- Tiempo total = $t(n) = 2+1+(n-1)+6(n-1) = 3+7(n-1) = 7n - 4$

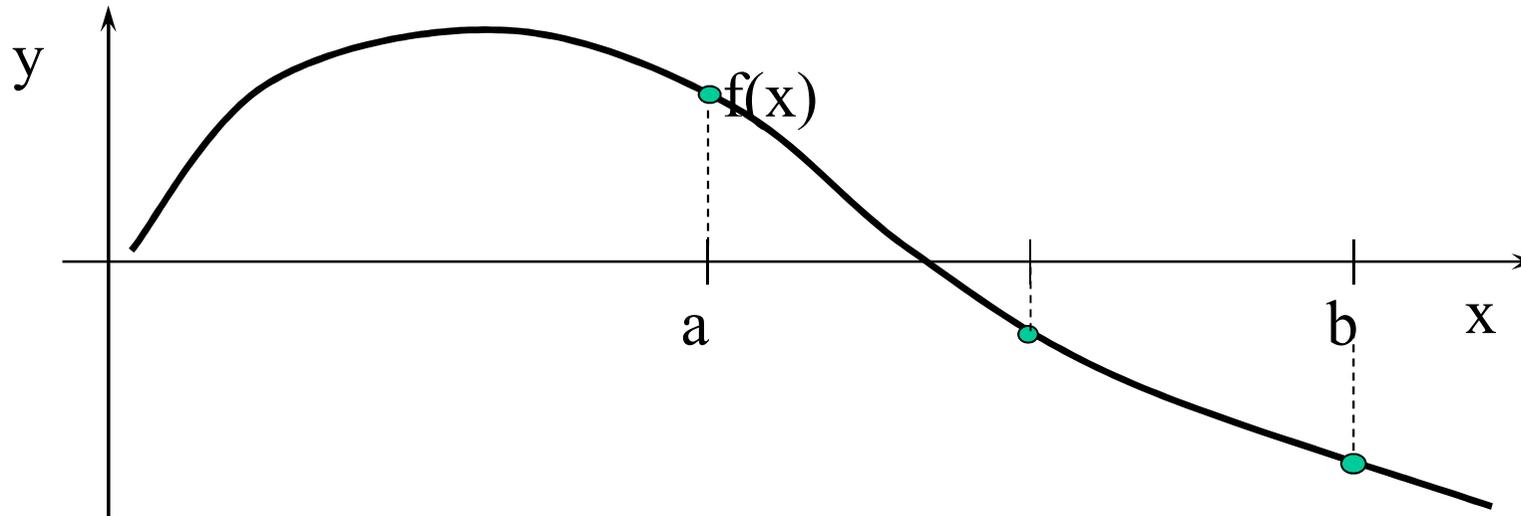
Notación asintótica

- Es útil concentrarse en la tasa de crecimiento del tiempo de ejecución t como función del tamaño de la entrada n .
- Se usa la notación O grande: Sean $f(n)$ y $g(n)$ funciones no negativas $f(n)$ es $O(g(n))$ si hay un valor $c > 0$ y $n_0 \geq 1$ / $f(n) \leq cg(n)$ para $n \geq n_0$
- Se dice que $f(n)$ es **de orden** $g(n)$
- Ejemplo: $7n - 4$ es $O(n)$ si $c=7$ y $n_0 = 1$
- Cualquier polinomio $a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ es $O(n^k)$
- *Ejercicio*: comparar las funciones para valores de n (potencias de 2) entre 2 a 1024

$\log n$ \sqrt{n} n $n \log n$ n^2 n^3 2^n $n!$

Bisección

- Método simple y efectivo para hallar la raíz de una función *continua* $f(x)$
- Se suponen conocidos dos valores de x , a y b , tal que $a < b$ y que $f(a)f(b) < 0$, tolerancia de error ϵ



Biseción

Input $f(x)$, a , b , ε

Algoritmo Bisecion

izq = a ; der = b

while (der - izq) > ε

$x = (izq + der)/2$

if $f(x) = 0$ **then** **exit**

else

if $f(x)f(a) < 0$ **then** der = x

else izq = x

endif

endif

endwhile

 raiz = $(izq + der)/2$

Output raiz

Búsqueda secuencial

Problema: buscar un elemento k en un conjunto de elementos K_i $i = 1, \dots, n$

Búsqueda secuencial: Aplicable a listas desordenadas

Input n , [Número de elementos]
 K_i , $i = 1, 2, \dots, n$ [Lista de claves]
 k [clave a buscar]

Algoritmo Busqsec

$i = 1$

while (true)

if $k = K_i$ **then** **Output** i ; **stop**

else $i = i + 1$

endif

if $i > n$ **then** **Output** “no hallado”

stop

endwhile

Búsqueda binaria

Aplicable a listas previamente **ordenadas**

Input n , [Número de elementos]
 $K_i, i = 1, 2, \dots, n$ [Lista de claves]
 k [clave a buscar]

Algoritmo Busqbin

$I = 1; D = n$

while (true)

$i = (I + D)/2$

if $k = K_i$ **then** **Output** i
stop

else if $k < K_i$ **then** $D = i - 1$

else $I = i + 1$

endif

if $I > D$ **then** **Output** “no hallado”
stop

endwhile

Búsqueda binaria

- Versión recursiva: Invocación Busqbinrec(1, n)

Input n, [Número de elementos]

$K_i, i = 1, 2, \dots, n$ [Lista de claves]

k [clave a buscar]

Algoritmo Busqbinrec(k, I, D)

if I > D **then**

Output “no hallado”; **return**

else

$i = (I + D)/2$

if $k = K_i$ **then** **Output** i; **return** i

else if $k < K_i$ **then** Busqbinrec(I, i-1)

else Busqbinrec(i+1, D)

endif

endif

Análisis del algoritmo

- Número de candidatos inicial: n
- Primera llamada: $n/2$
- i -ésima llamada: $n/2^i$
- En el peor de los casos las llamadas recursivas paran cuando no hay más candidatos.
- Máximo número de llamadas recursivas: menor entero m tal que $n/2^m < 1$
- $m > \log n$ $m = \lfloor \log n \rfloor + 1$
- Por tanto: la búsqueda binaria es de orden $O(\log n)$

Ordenamiento por burbuja

Problema: Poner en orden (ascendente o descendente) un conjunto de elementos K_i , $i = 1, \dots, n$

Input n, K_i , $i = 1, 2, \dots, n$

Algoritmo Burbuja

$i = n - 1$

while $i \neq 0$

for $j = 1$ **to** i

if $K_j > K_{j+1}$ **then**

$\text{temp} = K_j$ [intercambio]

$K_j \leftrightarrow K_{j+1}$

$K_{j+1} \leftrightarrow \text{temp}$

endfor

$i = i - 1$

endwhile

Output K_1, K_2, \dots, K_n en orden

Ordenamiento por inserción

- Toma un elemento cada vez y lo inserta en su lugar en la parte de la lista que ha sido ordenada.

Input $n, K_i, i = 1, 2, \dots, n$

Algoritmo Insercion

```
for i = 2 to n
    temp =  $K_i$ 
    j = i - 1
    while j  $\neq$  0 and temp  $\leq$   $K_j$ 
         $K_{j+1} = K_j$ 
        j = j - 1
    endwhile
     $K_{j+1} = temp$ 
endfor
```

Output K_1, K_2, \dots, K_n en orden

Multiplicación matricial

- Sea $A=[a_{ij}]$ una matriz de $m \times k$ y $B=[b_{ij}]$ una matriz de $k \times n$. El producto matricial de AB es $C=[c_{ij}]$ de $m \times n$ donde

$$c_{ij} = \sum_{r=1}^k a_{ir} b_{rj}$$

Input m, k, n, A, B

Algoritmo Mult_matr

for $i = 1$ **to** m

for $j = 1$ **to** n

$c_{ij} = 0$

for $r = 1$ **to** k

$c_{ij} = c_{ij} + a_{ir} b_{rj}$

endfor

endfor

endfor

Output C

Eliminación de Gauss

- Método de solución de la ecuación $Ax = d$
- Eliminar x_i desde la ecuación $i+1$ hasta la n para $i=1,2,\dots, n-1$
- Calculando de abajo arriba se obtiene el valor de cada variable: (sustitución hacia atrás)

$$x_n = d'_n / a'_{nn}$$

$$x_i = (d'_i - \sum_{j=i+1}^n a'_{ij} x_j) / a'_{ii}$$

Eliminación de Gauss

Input n, A, d

Algoritmo Elim_Gauss

for i = 1 **to** n - 1

for k = i + 1 **to** n

$m = a_{ki} / a_{ii}$

$a_{ki} = 0$

for j = i + 1 **to** n

$a_{kj} = a_{kj} - ma_{ij}$

endfor

$d_k = d_k - md_i$

endfor

endfor

for i = n **downto** 1

for j = i + 1 **to** n

$d_i = d_i - a_{ij}x_j$

endfor

$x_i = d_i / a_{ii}$

endfor

Output x

Método de Gauss-Seidel

- Se asume una solución $\mathbf{x}=(x_1, x_2, \dots, x_n)$ de un sistema de n ecuaciones. Se trata de obtener una solución mejor y usando:

$$y_k = -\left(\frac{1}{a_{kk}}\right)\left(\sum_{j=1}^{k-1} a_{kj}y_j + \sum_{j=k+1}^n a_{kj}x_j - d_k\right)$$

- El proceso continúa hasta que dos valores sucesivos de y sea menor a un ε

Método de Gauss-Seidel

Input $n, A, \mathbf{d}, \varepsilon$

Algoritmo Gauss_Seidel

do

$\text{dif} = 0$

for $k = 1$ **to** n

$c = 0; f = 0$

for $j = 1$ **to** $k - 1$

$c = c + a_{kj} y_j$

endfor

for $j = k + 1$ **to** n

$f = f + a_{kj} x_j$

endfor

$y_k = (1/a_{kk}) \cdot (c + f - d_k)$

$\text{dif} = \text{dif} + (x_k - y_k)^2$

endfor

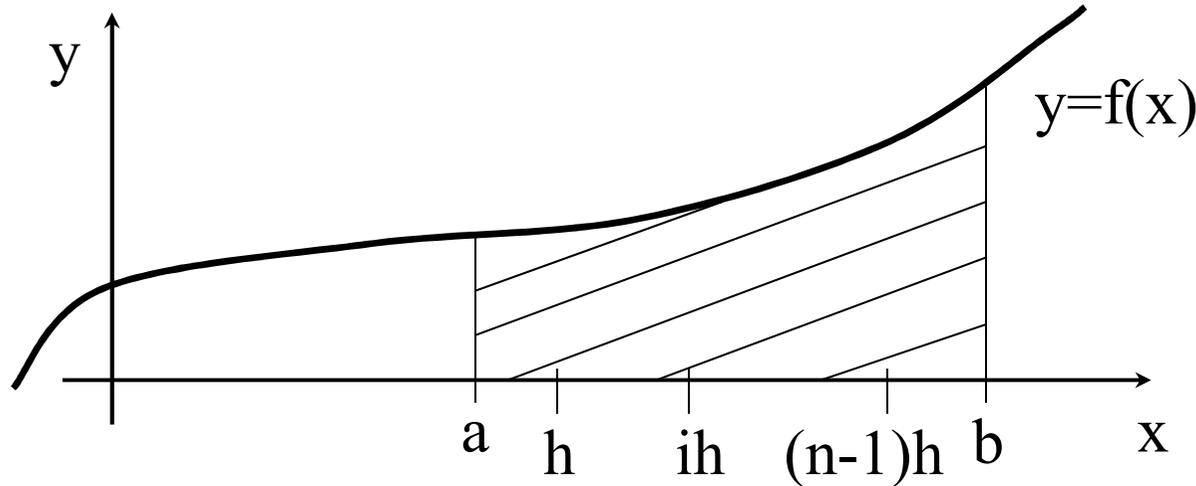
for $i = 1$ **to** n $x_i = y_i$

while $\text{dif} > \varepsilon$

Output \mathbf{x}

Integración: Método trapecio

$$\int_a^b f(x) dx \approx \frac{h}{2} f(a) + h \sum_{i=1}^{n-1} f(a + ih) + \frac{h}{2} f(b)$$



Integración: Mtdo. trapecio

Input n, a, b,
f(x) [función que calcula f(x)]

Algoritmo Trapecio

h = (b - a)/n

sum = (f(a) + f(b)) / 2

for i = 1 **to** n - 1

sum = sum + f(a + ih)

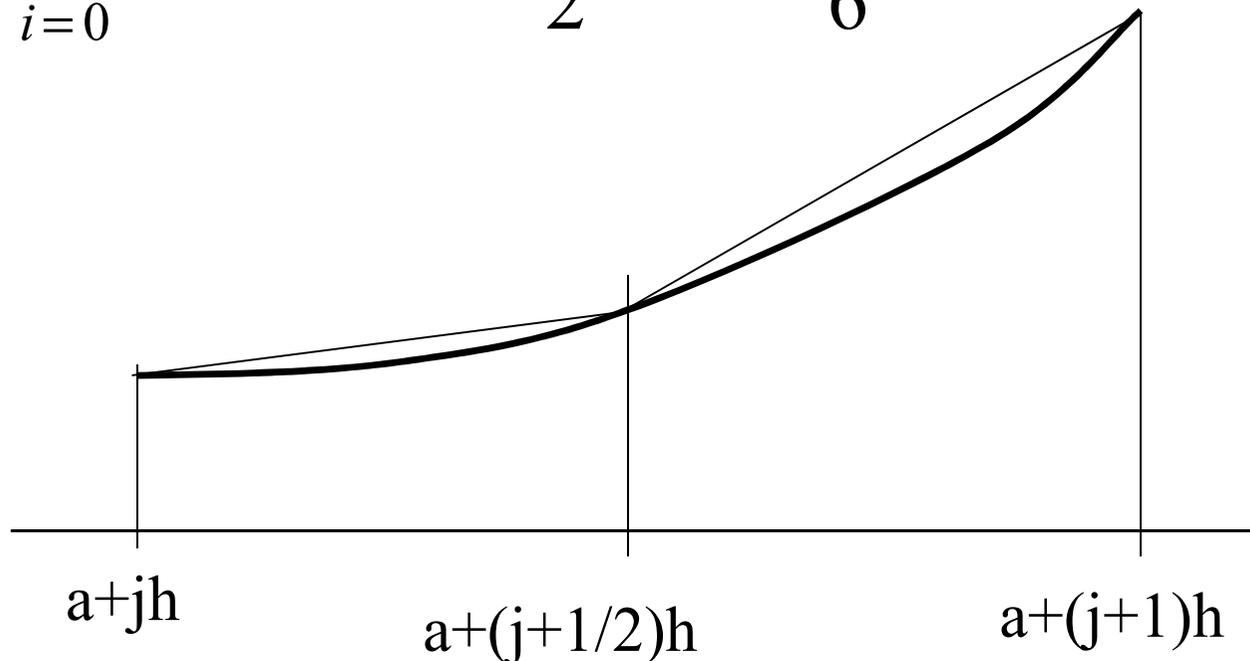
endfor

sum = h * sum

Output sum

Integración: Método Simpson

$$\int_a^b f(x) dx \approx \frac{h}{6} f(a) + \frac{h}{3} \sum_{i=1}^{n-1} f(a + ih) +$$
$$\frac{2h}{3} \sum_{i=0}^{n-1} f\left(a + \left(i + \frac{1}{2}\right)h\right) + \frac{h}{6} f(b)$$



Integración: Mtdo.Simpson

Input n, a, b,
f(x) [función que calcula f(x)]

Algoritmo Simpson

h = (b - a)/n

sum = f(a) + f(b)

for i = 1 **to** n - 1

sum = sum + 2f(a + ih)

endfor

for i = 1 **to** n - 1

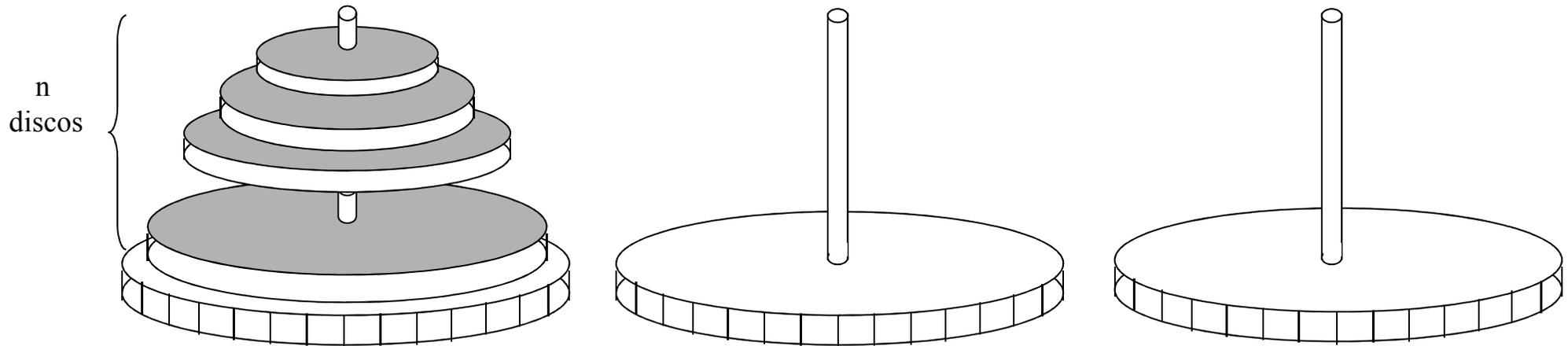
sum = sum + 4f(a + (i + 1/2)h)

endfor

sum = h * sum / 6

Output sum

Recursión: Torres de Hanoi



- Se trata de mover los discos desde un pivote, donde se encuentran inicialmente, a otro pivote, según las siguientes reglas:
 - Sólo se puede mover un disco cada vez
 - Un disco de mayor diámetro nunca puede estar encima de uno de menor diámetro

Recursión: Torres de Hanoi

Input n , [Número de discos]
 P_i , [Palo inicial; $1 \leq P_i \leq 3$]
 P_f [Palo final; $1 \leq P_f \leq 3, P_i \neq P_f$]

Algoritmo Torres_Hanoi

```
function H( $n, f, d$ )  
  if  $n = 1$  then  $f \rightarrow d$   
    else H( $n - 1, f, 6 - f - d$ )  
      Output “mover disco  $n$   
        de  $f$  a  $d$ ”  
      H( $n - 1, 6 - f - d, d$ )  
  endif  
  return  
endfunc
```

Invocación: H(num, P_i , P_f)

Torres de Hanoi: llamadas

H(3,1,3)

H(2,1,2)

H(1,1,3)

mover disco 1 de 1 a 3

mover disco 2 de 1 a 2

H(1,3,2)

mover disco 1 de 3 a 2

mover disco 3 de 1 a 3

Torres de Hanoi: llamadas

H(2,2,3)

H(1,2,1)

mover disco 1 de 2 a 1

mover disco 2 de 2 a 3

H(1,1,3)

mover disco 1 de 1 a 3

Recursión: Quicksort

Input $n, K_i, i = 1, 2, \dots, n$

Algoritmo Quicksort

function Qsort(i, d)

if $(d - i) > 0$ **then**

$p = \text{ParteLista}(i, d)$

Qsort($i, p - 1$)

Qsort($p + 1, d$)

endif

return

endfunc

Qsort($1, n$)

Output K_1, K_2, \dots, K_n en orden

Quicksort: ParteLista

Input $n, K_i, i = 1, 2, \dots, n$

function ParteLista(F, L)

$p = F$

$i = F + 1; j = L$

forever

while ($K_p < K_j$ and $j \neq p$) $j = j - 1$

if $p < j$ **then** $K_p \leftrightarrow K_j; p = j; j = j - 1$

else exit

while ($K_p > K_i$ and $i \neq p$) $i = i + 1$

if $p > i$ **then** $K_p \leftrightarrow K_i; p = i; i = i + 1$

else exit

endforever

return p

endfunction

Output Lista particionada

Búsqueda en cadenas

- Cadena a investigar: $S = s_1 s_2 \dots s_n$
- Cadena a buscar: $P = p_1 p_2 \dots p_m$
- s_i y p_j son caracteres de un alfabeto y usualmente $m < n$

Input n, S, m, P

Algoritmo Busca_fbruta

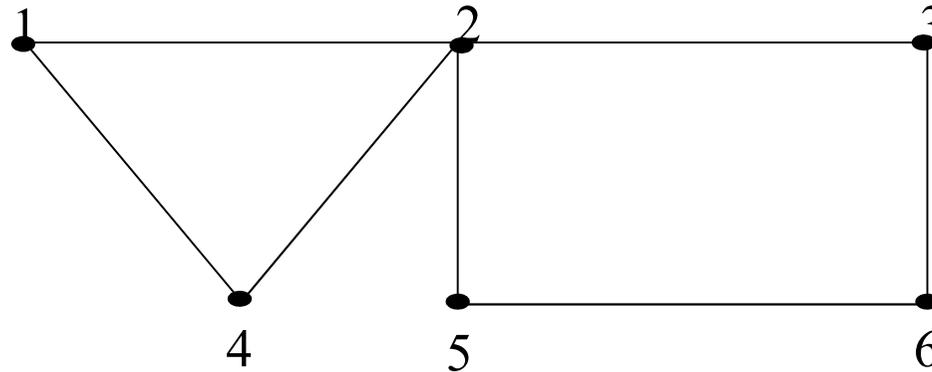
```
for i = 1 to n - m + 1
    j = 1
    while j <= m and p_j = S_{i+j-1}
        j = j + 1
    endwhile
    if j = m + 1 then
        Output i
    endif
endfor
Output 'No encontrado'
```

Grafos

- Un grafo es una estructura que consiste de un conjunto de vértices y un conjunto de aristas
- Las aristas pueden ser no dirigidas (grafo) o dirigidas (digrafo)
- Las estructuras de datos para representar un grafo son:
 - Matriz de adyacencia
 - Lista de adyacencia
- Dos vértices conectados por una arista son adyacentes

Matriz de adyacencia

- Matriz de $n \times n$ (n =número de vértices). Si hay una arista entre los nodos $(i, j) \rightarrow 1$, sino 0. En grafos es simétrica



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Caminos en grafos

- Sea A = matriz de adyacencia de un grafo. En la matriz A^m el elemento i, j contiene el número de caminos de longitud m desde el vértice i al j .

Input A [Matriz de adyacencia]
 m [longit. caminos deseados]

Algoritmo Num_caminos
 $B = A$
 for $k = 2$ **to** m
 Mult_matr(A, B, C)
 if $k < m$ **then** $B = C$
 endfor

Output C

Búsqueda en profundidad

Input G [Grafo con conj.vértices V]

Algoritmo Busca_en_profund

function Bprof(u)

 Visitar u

 Marcar u ‘visitado’

for w en A(u)

if marca(w) ≠ ‘visitado’ **then**

 Bprof(w)

endfor

return

endfunc

 Marcar todos vertices ‘no visitado’

 Seleccionar un vertice v

 Bprof(v)

Output Resultado procesar info. en cada vertice

Arbol de expansión mínimo

Input G [Grafo con conj.vértices V]
W [Conjunto pesos cada arista]

Algoritmo ArbolExpMin

Hallar arista $\{l, m\}$ de menor peso

$T = \{l, m, \{l, m\}\}$

$U = V - \{l\} - \{m\}$

$V' = V - \{v\}$

while $U \neq \emptyset$

Hallar arista $\{q, k\}$ de menor
peso desde un vertice q en T
a un vertice k no en T

$T = T \cup \{k, \{q, k\}\}$

$U = U - \{k\}$

endwhile

Output T