Introducción a la recursividad

Diseño y Análisis de Algoritmos

Universidad
Rey Juan Carlos

Contenidos

Introducción

¿Qué es la recursividad?

- La mayoría de programadores piensa que **la recursividad**:

  "surge cuando una rutina se llama a sí misma"

- Es cierto, pero la recursividad **es mucho más** que eso
  - Debemos alejarnos de esa visión tan estrecha
- **Herramienta muy potente** para la resolución de problemas computacionales y matemáticos
  - No hay que evitarla porque "parezca" difícil
- Es fundamental para el diseño de algoritmos, y por tanto, en la asignatura

¿Qué es la recursividad?

- Definición de **descendientes**, según el diccionario de la Real Academia Española:
  - *Descendientes*: "hijos, nietos o personas que **descienden** de otra"
    - Es recursiva, pero no es muy clara...
  - *Descender*: "proceder, por natural propagación, de un mismo principio o persona común, que es la cabeza de la familia"
    - No es recursiva, pero requiere saber qué significa "natural propagación", o "cabeza de familia"...

- La siguiente es mucho mejor

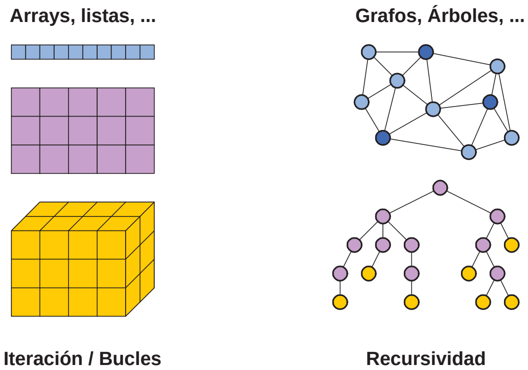
"los hijos y los descendientes de los hijos"

$$D(p) = \begin{cases} \emptyset & \text{si } H(p) = \emptyset \\ H(p) \cup D(H(p)) & \text{si } H(p) \neq \emptyset \end{cases}$$

H: hijos, D: descendientes, p: persona

¿Cuándo es útil aplicar la recursividad?

- Especialmente útil cuando la “estructura” del problema, algoritmo o los datos no es “lineal”:



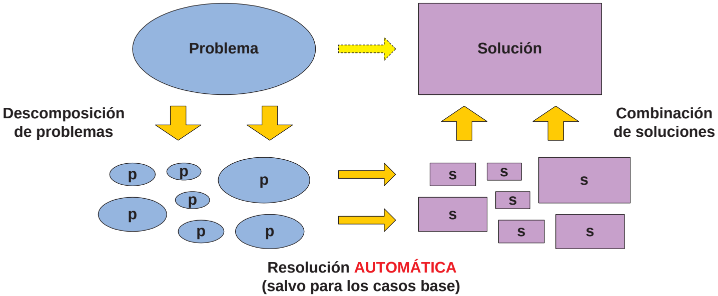
- Por supuesto, la recursividad también se puede emplear con “estructuras lineales”

Conceptos clave en la recursividad

- La **descomposición/simplificación** de problemas
  - Debemos ser capaces de reconocer que para resolver un problema primero podemos **resolver subproblemas idénticos al original, pero más sencillos o de menor tamaño**
- El concepto de **inducción**
  - Construimos nuestra solución **suponiendo que ya sabemos la solución a problemas más simples**
- El paradigma de **programación declarativa**
  - Hay que pensar en **qué** se va a hacer mucho más que en **cómo** se va a hacer

Descomposición/simplificación de problemas

- En general, **simplificar**, **transformar**, o **descomponer** un problema en otros más **sencillos** o de **menor tamaño** suele ser una buena idea
- La recursividad surge cuando este proceso genera problemas más simples, o de menor tamaño, idénticos al original



Descomposición/simplificación de problemas

- Problemas muy sencillos o triviales
  - Casos base
  - Suelen aparecer cuando el "tamaño" del problema es muy pequeño

- Problemas idénticos al original pero de menor tamaño
  - Casos recursivos
  - Aparecen cuando simplificamos el problema, de manera que los nuevos subproblemas se aproximen a los casos base
  - Generalmente, hay que "reducir" parámetros hacia los casos base
    $n \leftarrow n - 1$, $n \leftarrow n/2$, ...

Concepto de inducción

- Pensemos en los pasos que tomamos para realizar una demostración por inducción

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

1. Partimos de un **caso base**, para el que se cumple la definición

$$S(1) = \sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$$

2. **Suponiendo que se verifica para** $n-1$, demostrar que se cumple para n

$$S(n) = \sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i =$$

por la hipótesis de inducción, suponemos que se verifica para $n-1$:

$$= \frac{n(n-1)}{2} + n = \frac{n(n-1)}{2} + \frac{2n}{2} = \frac{n^2+n}{2} = \frac{n(n+1)}{2} \quad \checkmark$$

Concepto de inducción

- Pensemos en los pasos que tomamos para implementar de manera recursiva:

$$S(n) = \sum_{i=1}^{n} i$$

1. Establecer el caso base **caso base**: $S(1) = 1$ (también vale $S(0) = 0$)
2. **Suponiendo que conocemos la solución a** $S(n-1)$, construimos la solución para $S(n)$:

$$S(n) = n + S(n-1)$$

- Lo cual origina el siguiente algoritmo:

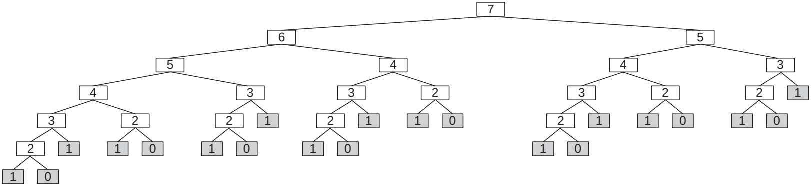
```
int sumatorio(int n){
  if(n==1)
    return 1;
  else
    return n + sumatorio(n-1);
}
```

El paradigma de programación declarativa

- En el ejemplo anterior, no nos preocupa **cómo** se calculará $S(n-1)$, nos vale con saber **qué** se calcula
- En general, hay que pensar en **qué** se va a hacer mucho más que en **cómo** se va a hacer
- Suponemos que sabemos **qué** se resuelve (el subproblema), pero no nos interesa saber **cómo**
- A diferencia del paradigma imperativo, evitaremos pensar en cómo se modifican los parámetros y variables a medida que se ejecuta un programa paso a paso

El paradigma de programación declarativa

- No perdemos tiempo en pensar **cómo** se resolverán los subproblemas
- Si podemos, evitaremos pensar en el *árbol de recursión*
- Por ejemplo, para números de Fibonacci saber que $F(n) = F(n-1) + F(n-2)$ es suficiente
- Pensar en el árbol de recursión generalmente no aclara nada



Plantilla para diseñar algoritmos recursivos

1. Reconocer los **casos base**
2. **Simplificar** o reducir el problema original hacia los casos base
3. **Completar** la solución, suponiendo que ya sabemos resolver los subproblemas simplificados

Tipos de recursividad

- Lineal (no final)

$$f(n, A) = \begin{cases} I & \text{si } n = 1 \\ A \cdot f(n-1, A) & \text{si } n > 1 \end{cases} \qquad g(n) = f\left(n, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\right)_{1,2}$$

- Lineal final (por cola)

$$f(n, a, b) = \begin{cases} a & \text{si } n = 0 \\ f(n-1, a+b, a) & \text{si } n \geq 1 \end{cases} \qquad g(n) = f(n, 0, 1)$$

- Múltiple

$$f(n) = \begin{cases} 1 & \text{si } n = 1, 2 \\ 1 + \sum_{i=1}^{n-2} f(i) & \text{si } n \geq 3 \end{cases} \qquad g(n) = f(n)$$

- Mutua

$$B(i) = \begin{cases} 1 & \text{si } i = 1 \\ A(i-1) & \text{si } i \geq 2 \end{cases} \quad A(i) = \begin{cases} 0 & \text{si } i = 1 \\ A(i-1) + B(i-1) & \text{si } i \geq 2 \end{cases} \quad g(n) = B(n) + A(n)$$

- Anidada

$$f(n, y) = \begin{cases} 1 + y & \text{si } n = 1, 2 \\ f(n-1, y + f(n-2, 0)) & \text{si } n \geq 3 \end{cases} \qquad g(n) = f(n, 0)$$

Ejemplos

Suma de los primeros n números naturales

$$S(n) = \sum_{i=1}^{n} i$$

1. Reconocer los **casos base** sencillos o triviales
   - $S(1) = 1$ y $S(0) = 0$
2. **Simplificar** o reducir el problema original hacia los casos base
   - El datos de entrada es $n \geq 0$. Como los casos base aparecen para $n = 1$ y $n = 0$, lo lógico es reducir n, ¿pero en cuánto?
     - $n \leftarrow n - 1$ origina un algoritmo sencillo
     - $n \leftarrow n/2$, suponiendo n par, presenta dificultades (aunque conduciría a un algoritmo más eficiente ya que disminuye n más rápidamente)

Suma de los primeros n números naturales

3. **Completar** la solución, suponiendo que ya sabemos resolver los subproblemas simplificados
   - Supongo que conozco $S(n-1)$. ¿Qué operación necesitamos para conseguir $S(n)$?

$$S(n) = S(n-1) + n \qquad \text{es lo más sencillo}$$

   - Supongo que conozco $S(n/2)$. ¿Qué operación necesitamos para conseguir $S(n)$?

$$S(n) = \sum_{i=1}^{n/2} i + \sum_{i=n/2+1}^{n} i = S(n/2) + ??? =$$

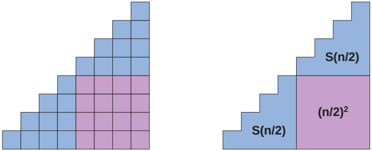
     - Se puede demostrar que:

$$S(n) = S(n/2) + \frac{3n^2}{8} + \frac{n}{4} \quad \text{pero esto no siempre se ve}$$

Suma de los primeros n números naturales

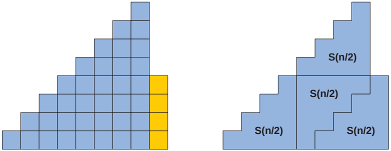
- En este caso podemos simplificar la expresión:

$$S(n) = 2S(n/2) + \frac{n^2}{4} \qquad \text{esto es un poco más fácil}$$



o

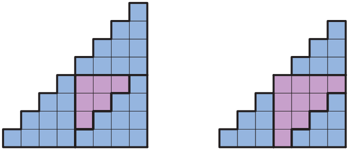
$$S(n) = 4S(n/2) - \frac{n}{2} \qquad \text{ahorras una multiplicación}$$



Suma de los primeros n números naturales

- Se pueden plantear funciones más sofisticadas (pero no necesariamente más eficientes):

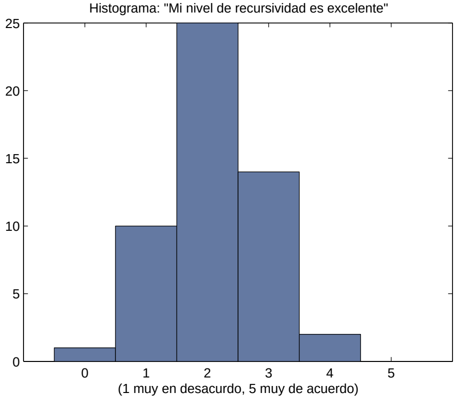
$$S(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 3S(\frac{n}{2}) + S(\frac{n}{2} - 1) & \text{si } n > 1 \text{ y } n \text{ par} \\ 3S(\frac{n-1}{2}) + S(\frac{n+1}{2}) & \text{si } n > 1 \text{ y } n \text{ impar} \end{cases}$$



- Esta función vale tanto para valores de n pares como impares

Nivel de recursividad

- Apreciación subjetiva de alumnos de segundo curso de Grado en Ingeniería Informática sobre su nivel de recursividad



- ¡Vamos a darle la vuelta a estos resultados!

Ejercicios que debéis saber hacer:

- Producto lento (usando sumas)
- Suma lenta (usando incrementos y decrementos de uno en uno)
- Sumar los dígitos de un número
- Contar los dígitos de un número
- Factorial
- Potencia
- Coeficientes binomiales
- Números de Fibonacci
- Sumatorio general
- Escribir un número invertido
- Torres de Hanoi

Ejemplos básicos

- Producto lento para números naturales

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

- Suma lenta para números naturales

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

- Sumar los dígitos de un número

$$f(n) = \begin{cases} n & \text{si } n < 10 \\ n\%10 + f(n/10) & \text{si } n \geq 10 \end{cases}$$

- Contar los dígitos de un número

$$f(n) = \begin{cases} 1 & \text{si } n < 10 \\ 1 + f(n/10) & \text{si } n \geq 10 \end{cases}$$

Ejemplos básicos

- Factorial

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

- Potencia

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

- Coeficiente Binomial

$$\binom{n}{p} = \begin{cases} 1 & \text{si } (n = p) \text{ o } (p = 0) \\ \binom{n-1}{p} + \binom{n-1}{p-1} & \text{en caso contrario} \end{cases}$$

- Números de Fibonacci

$$F_n = \begin{cases} 1 & \text{si } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 3 \end{cases}$$

Ejemplos básicos

- Sumatorio general

$$g(f, m, n) = \sum_{i=m}^{n} f(i) = \begin{cases} f(n) & \text{si } n = m \\ f(m) + g(f, m+1, n) & \text{si } m < n \end{cases}$$

- Escribir un número invertido

$$p(n) = \begin{cases} \text{write}(n) & \text{si } n < 10 \\ \text{write}(n\,\%10);\ \text{p(n/10);} & \text{si } n \geq 10 \end{cases}$$

- Torres de Hanoi

```
hanoi(int n, int destino, int origen, int auxiliar)
  if (n > 0)
     hanoi(n-1, auxiliar, origen, destino);
     << Mover disco n desde origen a destino >>
     hanoi(n-1, destino, auxiliar, origen);
```

Problemas Variados

Algoritmo de Horner

- Un método famoso para la evaluación de un polinomio es el **algoritmo de Horner**. Evalúa un polinomio de grado n utilizando solamente n multiplicaciones. La clave detrás del algoritmo de Horner es la descomposición de un polinomio $p(x)$ de grado n de la siguiente manera:

$$p(x,d) = d_0 + x(d_1 + x(d_2 + \cdots + x(d_{n-1} + d_n x)\cdots))$$

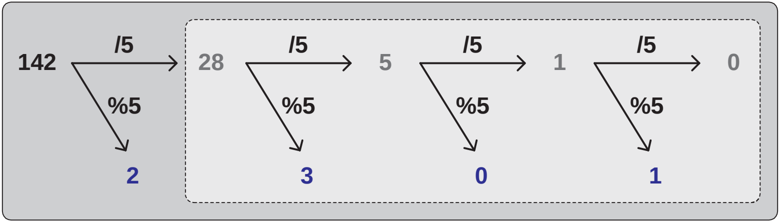
- Suponiendo que los coeficientes del polinomio de grado n se almacenan en un array d de longitud $n+1$, una solución es:

$$f(d, inic, fin, x) = \begin{cases} d[fin] & \text{si } inic = fin \\ d[inic] + x \cdot f(d, inic+1, fin, x) & \text{si } inic < fin \end{cases}$$

$$p(x,d) = f(d, 0, n, x)$$

Cambio de base

- $142_{10} = 1032_5$
- Algoritmo y descomposición del problema ($28_{10} = 103_5$):

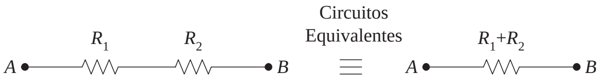


$$f(n,b) = \begin{cases} n & \text{si } n < b \\ n\%b + 10f(n/b, b) & \text{si } n \geq b \end{cases}$$

- Para 142_{10} genera $2 + 10(3 + 10(0 + 10(1))) = 1032$

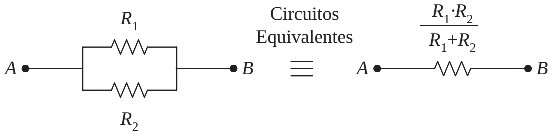
Escalera de resistencias

- **Dos resistencias en serie**



La resistencia entre A y B, $R_{AB} = R_1 + R_2$. Esto quiere decir que podemos construir un circuito equivalente con una resistencia de valor $R_1 + R_2$.

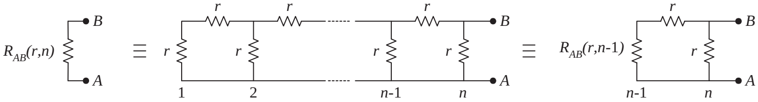
- **Dos resistencias en paralelo**



La resistencia entre A y B, R_{AB} se obtiene usando la siguiente fórmula $1/R_{AB} = 1/R_1 + 1/R_2$. Es decir, podemos construir un circuito equivalente con una resistencia de valor $R_1 R_2/(R_1 + R_2)$.

Escalera de resistencias

- Hallar una expresión recursiva $R_{AB}(r,n)$ para la resistencia entre A y B, dado el circuito mostrado con exactamente n resistencias verticales cuyo valor es r ohmios

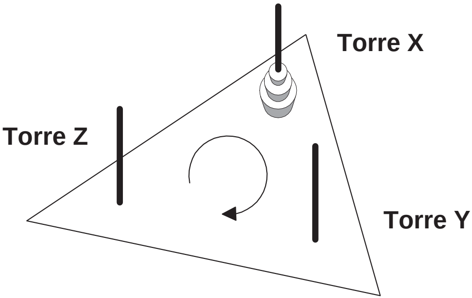


- Un circuito con una sola resistencia de valor $R_{AB}(r,n)$ entre dos extremos A y B sería equivalente al circuito en escalera completo con n peldaños

$$R_{AB}(r,1) = r \qquad \frac{1}{R_{AB}(r,n)} = \frac{1}{r} + \frac{1}{r + R_{AB}(r,n-1)}$$

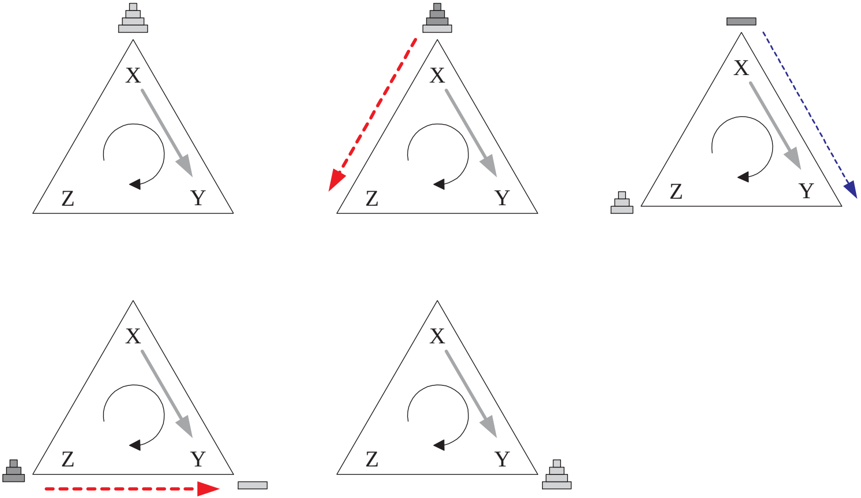
Torres de Hanoi Cíclicas

- Variante de la Torres de Hanoi, en la que los discos sólo pueden moverse en una “dirección”



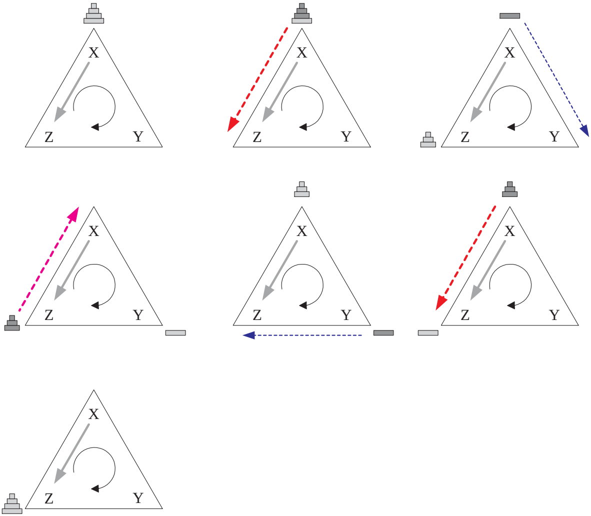
Torres de Hanoi Cíclicas

Movimiento en el sentido de las agujas del reloj



Torres de Hanoi Cíclicas

Movimiento en el sentido contrario al de las agujas del reloj



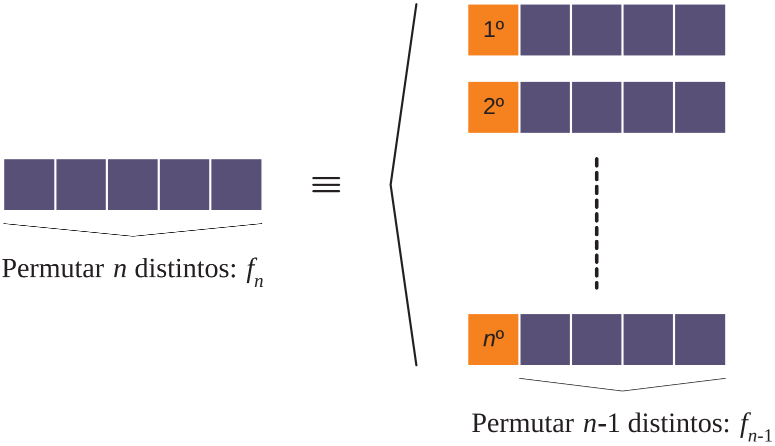
Torres de Hanoi Cíclicas

```
Reloj(n,X,Y,Z)

    si n>0

        AntiReloj(n-1,X,Z,Y)
        Mueve el disco n de X a Y
        AntiReloj(n-1,Z,Y,X)


AntiReloj(n,X,Z,Y)

    si n>0

        AntiReloj(n-1,X,Z,Y)
        Mueve el disco n de X a Y
        Reloj(n-1,Z,X,Y)
        Mueve el disco n de Y a Z
        AntiReloj(n-1,X,Z,Y)
```

Problemas Combinatorios

Ordenar *n* elementos distintos

- Hallar de cuántas maneras pueden ordenarse *n* elementos distintos.



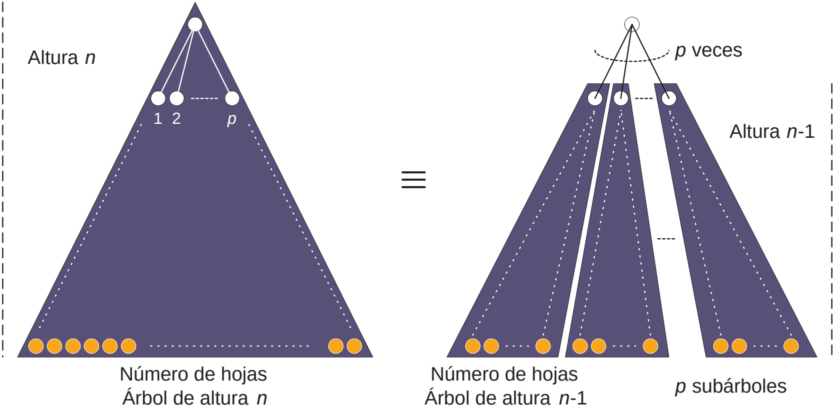
Ordenar n elementos distintos

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \\ n \cdot f(n-1) & \text{si } n > 1 \end{cases}$$

- El caso base es trivial (para que coincida con la función factorial $f(0)$ debe ser interpretado como 1).
- Fijando un elemento en la primera posición, habrá $f(n-1)$ formas de ordenar $n-1$ elementos. Como hay n formas de escoger el primer elemento, es necesario sumar todas, por tanto: $f(n) = n \cdot f(n-1)$.
- Son permutaciones (factorial)

Hojas de un árbol

- Hallar cuántas hojas tiene un árbol de altura n cuyos nodos padre siempre tienen p hijos.



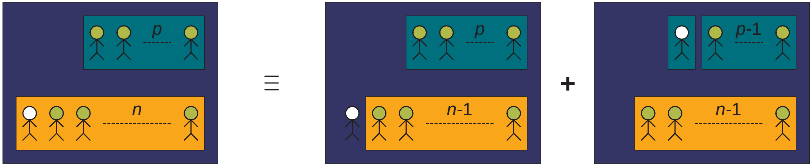
Hojas de un árbol

$$f(p,n) = \begin{cases} 1 & \text{si } n = 0 \\ p \cdot f(p, n-1) & \text{si } n > 0 \end{cases}$$

- El caso base es trivial.
- Si el nodo raíz tiene p hijos, esto significa que tiene p subárboles de altura $n-1$. Como las hojas del árbol de altura n es igual a la suma de las hojas de los subárboles, tenemos: $f(p,n) = p \cdot f(p, n-1)$.
- Son variaciones con repetición (potencia)

Grupos de alumnos

- En una clase con n alumnos, p alumnos van a salir a la pizarra a resolver un ejercicio. Calcular cuántas maneras diferentes existen de escoger a esos p alumnos, sin importar el orden.



Grupos de alumnos

$$f(n,p) = \begin{cases} 1 & \text{si } n = p \text{ ó } p = 0 \\ f(n-1,p) + f(n-1,p-1) & \text{en caso contrario} \end{cases}$$

- Los casos base son triviales (si consideramos que $f(n,0) = 1$)
- Para hallar $f(n,p)$ podemos pensar en dos situaciones. Suponiendo que elegimos a un alumno al azar, si no sale a la pizarra habría $f(n-1,p)$ formas diferentes de elegir al resto de alumnos. Si sale a la pizarra, habrá $f(n-1,p-1)$ formas distintas de elegirlos. El resultado, naturalmente es la suma de estas cantidades.
- Son combinaciones (coeficientes binomiales)

Subir escaleras

- ¿De cuántas maneras se puede subir unas escaleras de n peldaños, dando pasos de uno o dos escalones?

Subir escaleras

$$f(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2 & \text{si } n = 2 \\ f(n-1) + f(n-2) & \text{si } n \geq 3 \end{cases}$$

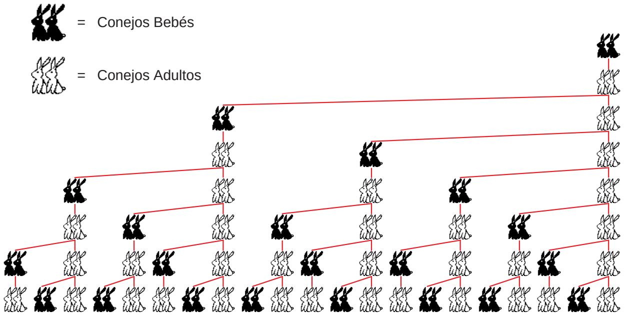
- Los casos base son triviales
- Para subir las escaleras tenemos dos posibilidades al dar el primer paso:
  - Subir un escalón, con lo que quedarán $n-1$ escalones
  - Subir dos escalón, con lo que quedarán $n-2$ escalones
- Por tanto, el resultado es el número de maneras de subir $n-1$ peldaños ($f(n-1)$), más el número de maneras de subir $n-2$ peldaños ($f(n-2)$)
- Son números de Fibonacci: $f(n) = F_{n+1}$

Problema de la población de conejos de Fibonacci

- Resolver el problema de la población de conejos de Fibonacci, que consiste en calcular el tamaño de una población de conejos después de n meses en condiciones ideales, bajo las siguientes reglas:
  - Inicialmente, una pareja de conejos recién nacidos, un macho y una hembra, son colocados en un prado
  - Los conejos tardan un mes en madurar
  - Los conejos maduros tardan otro mes en producir una nueva pareja de conejos recién nacidos
  - Los conejos nunca mueren
  - La hembra siembre da a luz a un macho y una hembra, desde el segundo mes en adelante

Problema de la población de conejos de Fibonacci

- El proceso se puede ver, por ejemplo, mediante un árbol o una tabla



Mes	Pares Bebés	Pares Adultos	Pares Totales
1	1	0	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5
6	3	5	8
7	5	8	13
8	8	13	21

Problema de la población de conejos de Fibonacci

- Modelado:

$$B_i = \begin{cases} 1 & \text{si } i = 1 \\ A_{i-1} & \text{si } i \geq 2 \end{cases} \qquad A_i = \begin{cases} 0 & \text{si } i = 1 \\ A_{i-1} + B_{i-1} & \text{si } i \geq 2 \end{cases}$$

- B_i y A_i son el número de parejas de conejos "Bebés" y "Adultos" en el mes i-ésimo, respectivamente

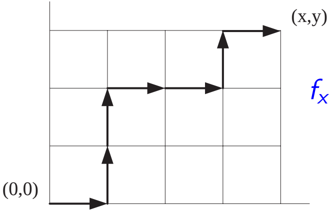
- Se puede demostrar que:

$$F_i = A_i + B_i = A_{i+1} = B_{i+2}$$

donde F_i es el i-ésimo número de Fibonacci.

Contar caminos

- ¿Cuántos posibles caminos existen para llegar desde el origen (0,0) hasta la coordenada (x,y), $x, y \in \mathbb{N}$, si únicamente se pueden dar pasos de longitud 1 hacia la derecha o hacia arriba?



$$f_{x,y} = \begin{cases} 0 & \text{si } (x < 0) \text{ ó } (y < 0) \\ 1 & \text{si } (x = 0) \text{ y } (y = 0) \\ f_{x,y-1} + f_{x-1,y} & \text{en caso contrario} \end{cases}$$

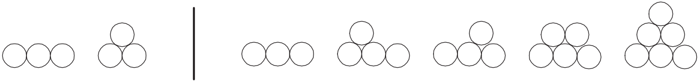
- Se puede demostrar que:

$$f_{x,y} \equiv \frac{(x+y)!}{x! \cdot y!} = \binom{n}{n-x} = \binom{n}{n-y}$$

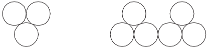
donde $n = x + y$

Configuraciones de monedas

- El problema consiste en determinar el número de configuraciones de monedas que podemos formar, en filas horizontales, de acuerdo a las siguientes reglas:
  1. Todas las monedas en una fila se están tocando (no hay huecos entre ellas)
  2. Cada moneda situada en una fila que no se la inferior, está tocando a dos monedas por debajo de ella
- La siguientes configuraciones no serían válidas:

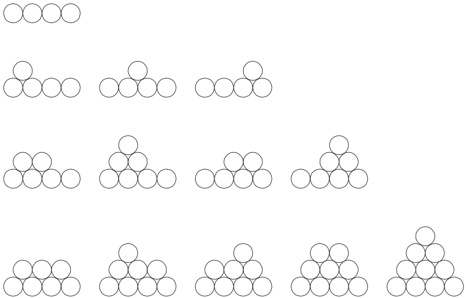


- La siguientes configuraciones no serían válidas:



Configuraciones de monedas, primera solución

- Solución tradicional (agrupamos según el número de monedas en el segundo nivel):

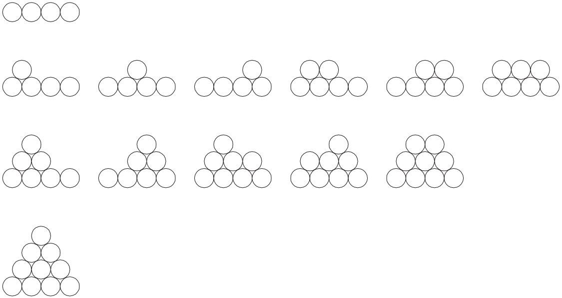


$$A_n = 1 + 1 \cdot A_{n-1} + 2 \cdot A_{n-2} + 3 \cdot A_{n-3} + \cdots + (n-1) \cdot A_1$$

A_n es el número de configuraciones con n monedas en la fila de abajo

Configuraciones de monedas, segunda solución

- Solución alternativa (agrupamos según las configuraciones de una determinada altura):



Configuraciones de monedas, segunda solución

- Sea $P_{h,n}$ el número total de configuraciones de altura h, que tienen n monedas en la fila de abajo, entonces para $n, h > 0$:

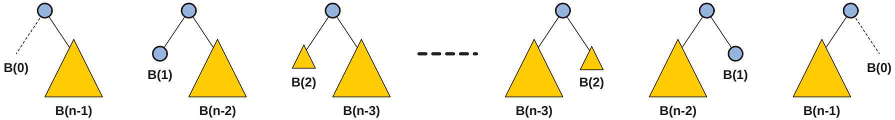
$$P_{h,n} = \begin{cases} 0 & \text{si } h > n \\ 1 & \text{si } (h = n) \text{ o } (h = 1) \\ \displaystyle\sum_{i=1}^{n-h+1} i \cdot P_{h-1,n-i} & \text{en otro caso} \end{cases}$$

por lo que la solución es:

$$A_n = \sum_{h=1}^{n} P_{h,n} = F_{2n-1} \text{ (número de Fibonacci)}$$

Número de árboles binarios con n nodos

- Cada nodo en un árbol binario tiene 0, 1 o 2 hijos
- Si $B(n)$ representa el número de árboles binarios con n nodos



$$B(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \displaystyle\sum_{i=0}^{n-1} B(i) \cdot B(n-i-1) & \text{si } n > 1 \end{cases}$$