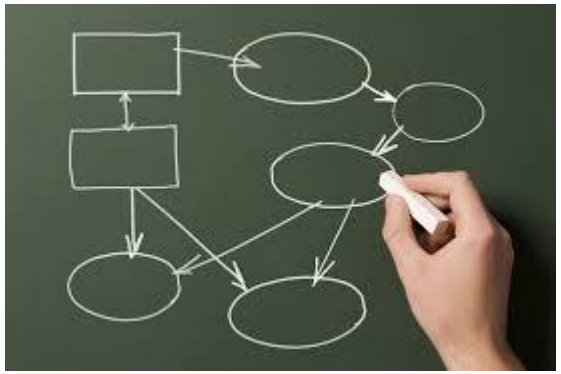
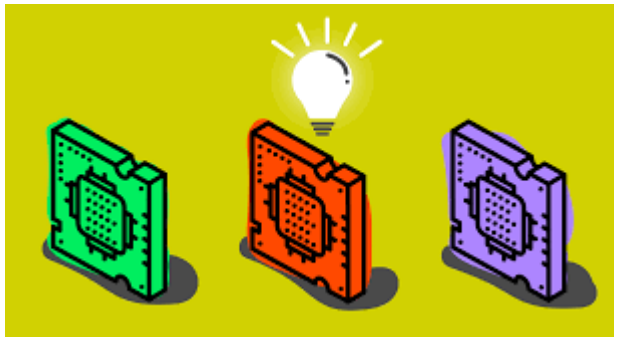
TEMA 3: Estrategias para construir algoritmos

OBJETIVO: El alumno conocerá y aplicará las principales técnicas para construir algoritmos, así como la recursividad como estrategia para desarrollar algoritmos eficientes.

3.1 Aspectos generales de la construcción de algoritmos





3.1 Estrategias de diseño

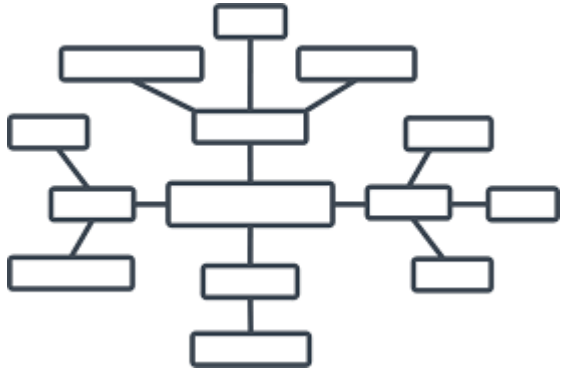
- Las estrategias de diseño establecen la forma en la que se modela el problema que se va a resolver
- Saber cual es la estrategia apropiada para cada problema depende de factores como los datos que se trabajan, las entradas y salidas esperadas

3.1.1.1 Top-down

Consiste en establecer una serie de etapas, niveles o jerarquías iniciando desde los aspectos más generales hasta llegar a los más específicos.

Se busca establecer una relación entre las etapas de la solución del problema.

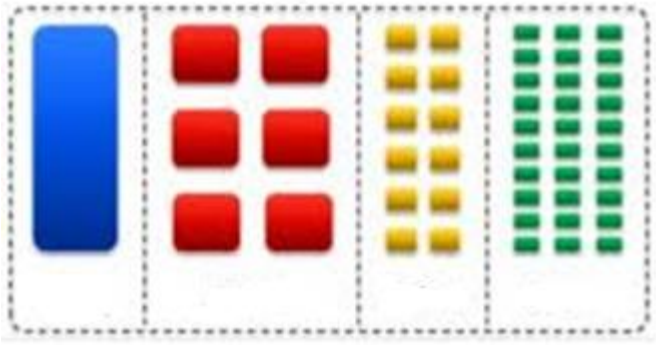
Puede ser por mediante módulos conectados entre si, de entradas y salidas de información o de datos abstractos que contienen otros.

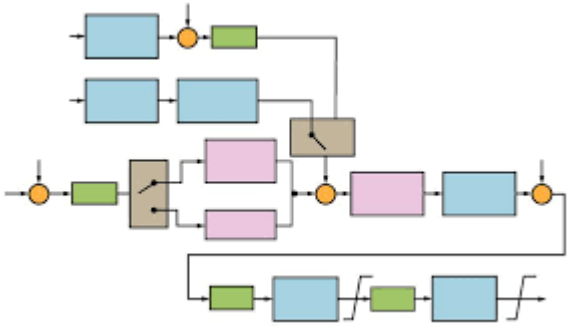


3.1.1.1 Top-down

Esta técnica tiene los siguientes aspectos:

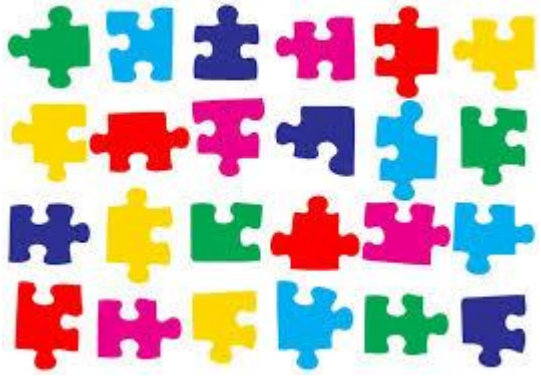
- Simplificación del problema y sus módulos haciendo una descomposición.
- Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
- El programa final queda estructurado en forma de bloque o módulos lo que hace más sencilla su lectura y mantenimiento.





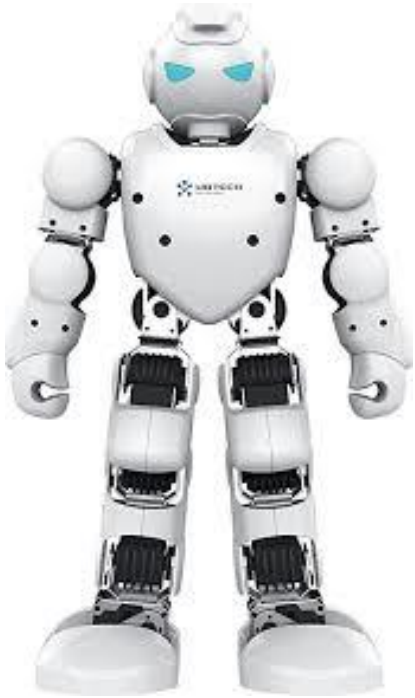
3.1.1.2 Bottom-Up

- Parte desde los módulos particulares o específicos que son parte de la solución general de un problema.
- Se suele utilizar cuando ya se elaboró el diseño de una solución y se va a realizar la implementación de la misma.



3.1.1.2 Bottom-Up

- Si el diseño de la solución se realiza siguiendo este modelo, puede ser complicado ya que no se tiene un panorama general del problema completo.
- Se utiliza cuando el punto de partida es simple y después se agregan nuevas funcionalidades a un sistema



3.1.2 Problemas de Optimización

- Un problema de optimización consiste en minimizar o maximizar el valor de una variable.
- La variable que se desea minimizar o maximizar debe ser expresada como función de otra de las variables relacionadas en el problema.

3.1.2 Problemas de Optimización

- En un **problema de optimización** no solo se busca una solución, sino que se busca "*la mejor*" de todas.
- En los problemas de optimización, la dinámica de los algoritmos busca tomar una serie de decisiones, cuyo efecto general es reducir o maximizar un recurso (una variable) especifico

(Mínimo costo - Máximo beneficio)

3.2 Búsqueda exhaustiva o fuerza bruta

3.2.1 Fuerza Bruta - Definición

- Consiste en verificar todas las posibles soluciones a un problema hasta encontrar aquella o aquellas que cumplan con el propósito de resolver el problema.
- Es la forma de aproximación más simple para resolver un problema.

Ventajas

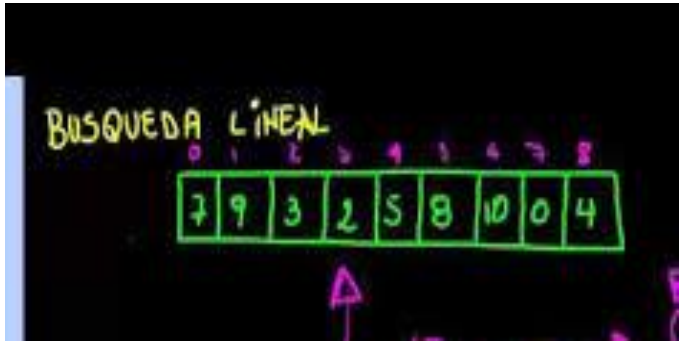
- ✓ Simplicidad en los algoritmos
- ✓ Aplicable a diferentes tipos de problemas

Desventajas

- ✓ Carece de eficiencia en muchos casos.
- ✓ Funcional para instancias relativamente pequeñas de los problemas que se resuelven

3.2.2 Fuerza Bruta – Ejemplos

- Buscar los divisores de un número
  - ✓ Verificar si un número es primo.
- Dado un conjunto de números, encontrar la suma máxima entre ellos
- Dada una cierta longitud y un conjunto de caracteres, encontrar un password.
- Búsqueda lineal



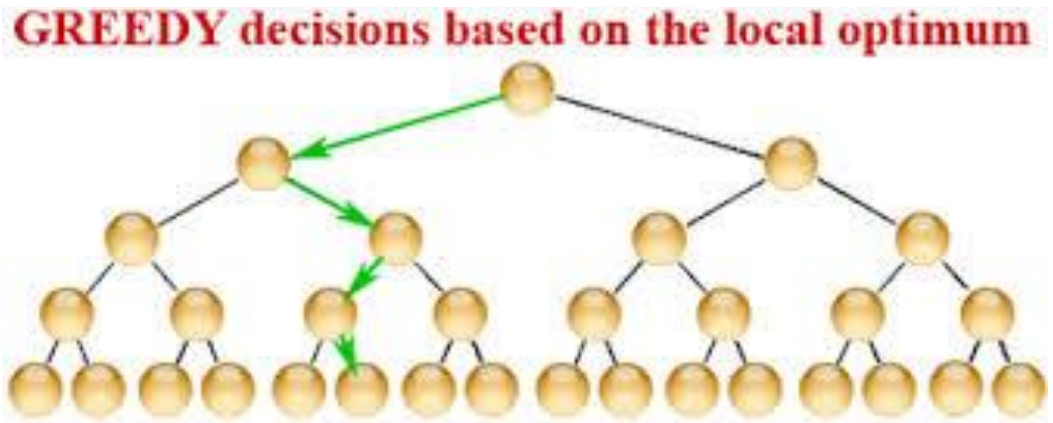
3.3 Algoritmos ávidos (Greedy)

3.3.1 Algoritmos Ávidos - Definición

- Un algoritmo ávido es aquel que, para resolver un determinado problema, elige la mejor opción en cada paso local (de manera directa) con la esperanza de llegar a una solución general óptima.
- Este esquema algorítmico plantea pocas dificultades al diseñar y comprobar su funcionamiento.

3.3.1 Algoritmos Ávidos - Definición

- Consiste en tomar las decisiones sucesivamente, de modo que cada decisión individual sea la mejor de acuerdo con algún criterio “a corto plazo” cuya evaluación no sea demasiado costosa.
- Una vez tomada una decisión, no se podrá revertir, ni siquiera si más adelante se hace obvio que no fue una buena.



3.3.2 Algoritmos Ávidos – Forma general

1. Establecer el problema en función de decisiones directas de tal forma que cada decisión resulte en un nuevo subproblema a resolver.
2. Mostrar que siempre hay una solución al problema original a partir de la decisión que tome la etapa actual.
3. Demostrar que el subproblema obtenido a partir de la primer solución proporcionada con el algoritmo “greedy” aún tiene una solución al problema.

3.3.3 Algoritmos Ávidos – Ejemplos

- La estrategia en la que trabaja un algoritmo greedy es top-down.
- Los algoritmos de tipo “GREEDY” no necesariamente obtienen la solución óptima.
- Algunos ejemplos típicos de problemas conocidos que se resuelven utilizando un algoritmo ávido son:
  - Problema del cambio de monedas.
  - Problema de la mochila.

3.3.3 Algoritmos Ávidos – Ejemplos

- **Problema del cambio de monedas**

Consiste en descomponer una cantidad ***n*** en el menor numero de monedas.

El enfoque de resolución de este problema utilizando un algoritmo de tipo greedy es tomar siempre la moneda de más alta denominación que no rebase la cantidad ***n***.

3.3.3 Algoritmos Ávidos – Ejemplos

Ejemplo: Pesos Mexicanos {10,5,2,1}

a) $24: {10,10,2,2}
b) $15: {10,5}

Ejemplo: Otra moneda {11,5,1}

$15: { 11, 1, 1, 1, 1}
$18:{11,5,1,1,1}

3.3.3 Algoritmos Ávidos – Ejemplos

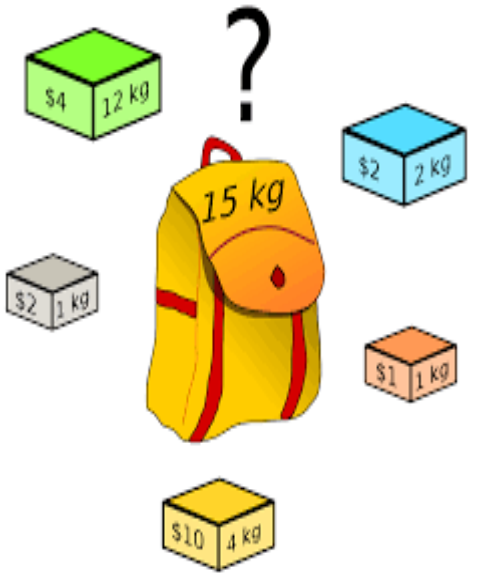
- **Problema de la mochila**

Se tiene una mochila con una capacidad de ***n*** kilogramos y se tiene un conjunto de objetos con un peso y un valor económico cada uno.

El objetivo es llevar en la mochila el mayor valor posible.

Peso	**10**	**20**	**30**	**40**	**50**
Costo	20	10	65	40	60

Mochila de 100 kg … ¿Qué objetos seleccionar?



3.3.3 Algoritmos Ávidos – Ejemplos

Peso	**10**	**20**	**30**	**40**	**50**
Costo	20	10	65	40	60

- **Primero el más ligero**

  Peso = 10 + 20 + 30 + 40 = 100 kg

  Valor = 20 + 10 + 65 + 40 = 135

- **Primero el más valioso**

  Peso = 30 + 50 + 10 = 90kg

  Valor = 65 + 60 + 20 = 145

- **Primero el que tenga mas valor por unidad de peso**

  Peso = 30 + 10 + 50 = 90kg

  Valor = 65 + 20 + 60 = 145

3.3.3 Algoritmos Ávidos – Ejemplos

Problema de la mochila fraccional

En esta variante del problema, se puede tomar “fracciones” de los objetos de tal manera que siempre se llena la mochila a su máxima capacidad

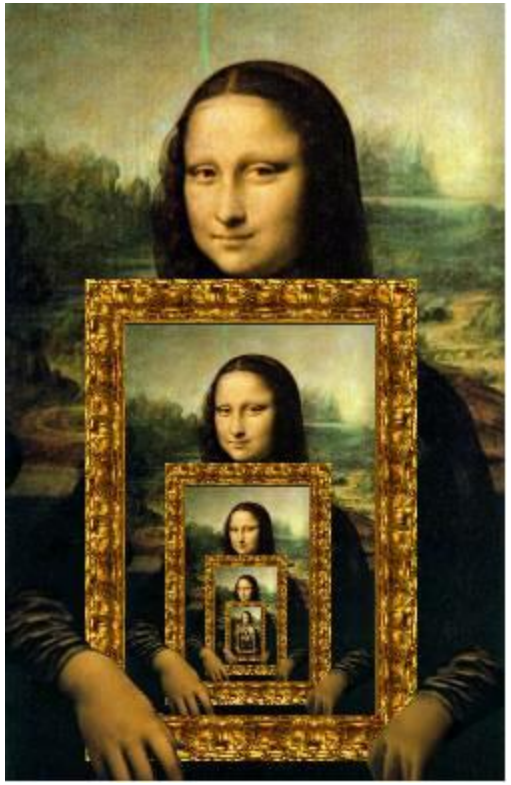
Peso	**10**	**20**	**30**	**40**	**50**
Costo	20	10	65	40	60
Valor x unidad	2	0.5	2.1666	1	1.2

3.3.3 Algoritmos Ávidos – Ejemplos

Peso	10	20	30	40	50
Costo	20	10	65	40	60
Valor x unidad	2	0.5	2.1666	1	1.2

- **Primero el más ligero**
  Peso = 10 + 20 + 30 + 40 +10 (de 50)= 110 kg
  Valor = 20 + 10 + 65 + 40 + 12 = 147
- **Primero el más valioso**
  Peso = 30 + 50 + 30 (de 40) = 110kg
  Valor = 65 + 60 + 30 = 155
- **Primero el que tenga mas valor por unidad de peso**
  Peso = 30 + 10 + 50 + 20(de 40)= 110kg
  Valor = 65 + 20 + 60 + 20= 165

3.4 Recursividad

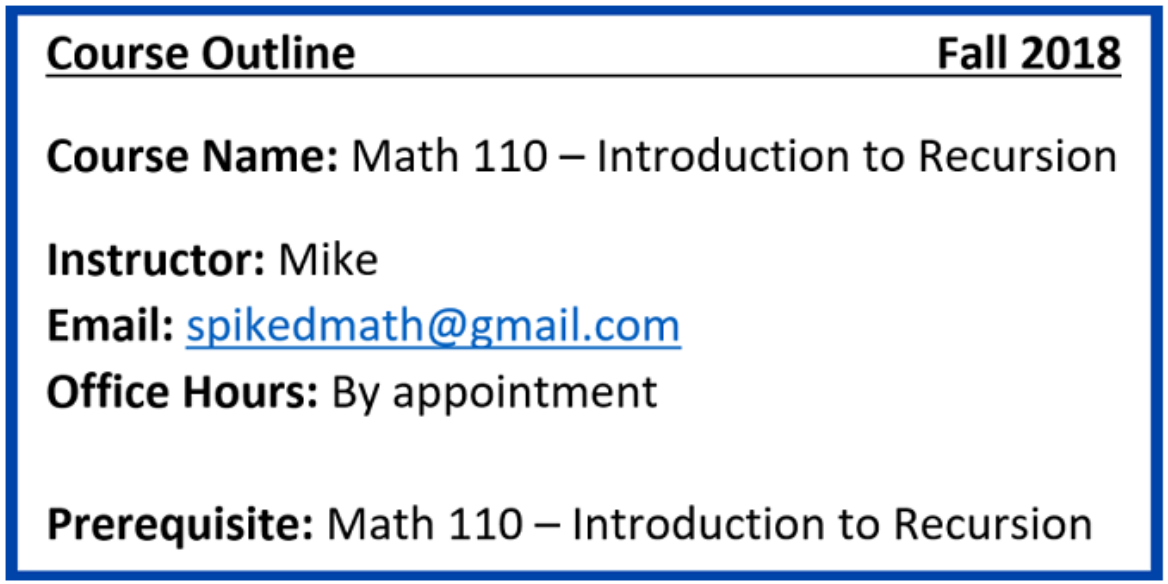


3.4.1 El concepto de Recursividad

- Una función recursiva es aquella que hace una llamada a sí misma.
- ¿Cómo puede un método resolver un problema llamándose a sí mismo?
- La clave está en que esa llamada se hace en un contexto diferente, que generalmente es más simple que el anterior.

3.4.1 El concepto de Recursividad

- La función puede invocarse a sí misma (una o más veces) hasta llegar a una versión que no requiere una nueva llamada.
- Para que una definición recursiva sea válida, la referencia a sí misma debe ser *más simple* que el caso previo.



3.4.2 Definición de una función recursiva

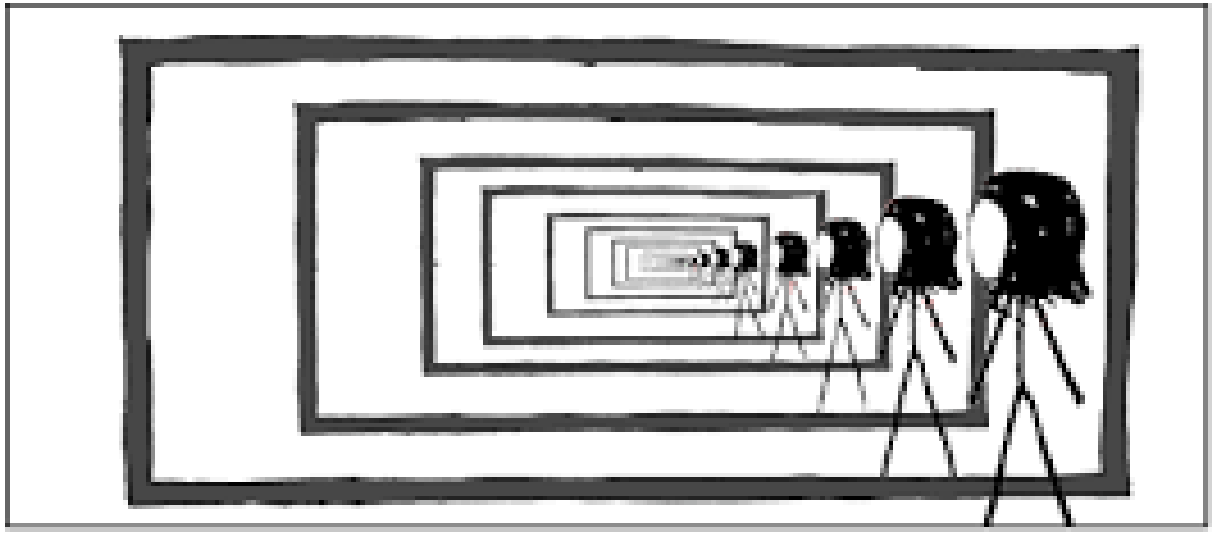
Una función recursiva requiere de 2 partes:

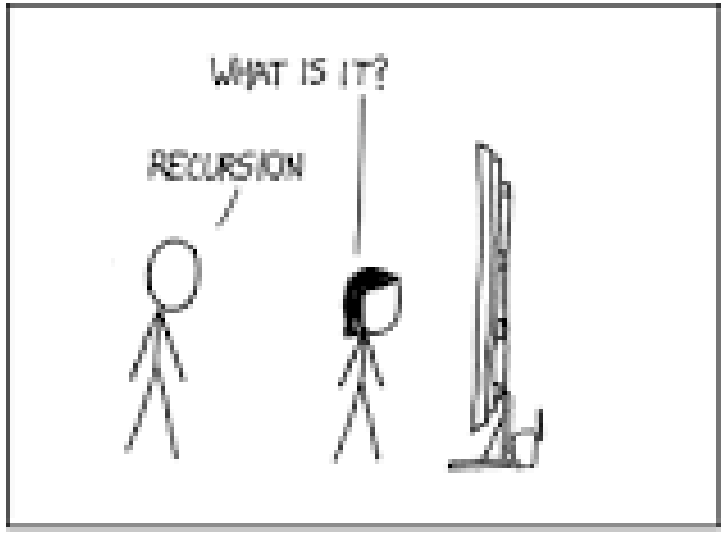
1.- Llamada recursiva (a sí misma): En alguna de sus instrucciones se invoca a sí misma, con un parámetro de entrada diferente, que implica un caso más simple. Se hacen nuevas llamadas a la función, cada vez más próximas al caso base.

2.- Caso Base: Trivial o fin de recursión. Es la parte donde el problema se resuelve directamente sin una nueva llamada a sí mismo. Evita la continuación indefinida de la ejecución.

3.4.3 Otros aspectos de Recursividad

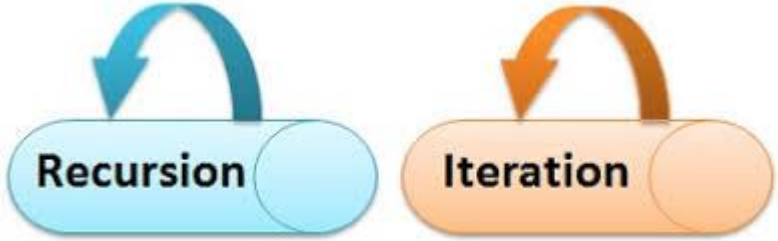
- La recursión es una potente herramienta que se aplica en la resolución de problemas.
- Las estructuras autoreferenciales son un buen ejemplo de aplicación de naturaleza recursiva.





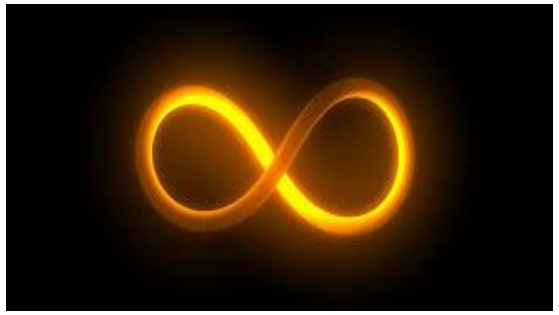
3.4.3 Otros aspectos de Recursividad

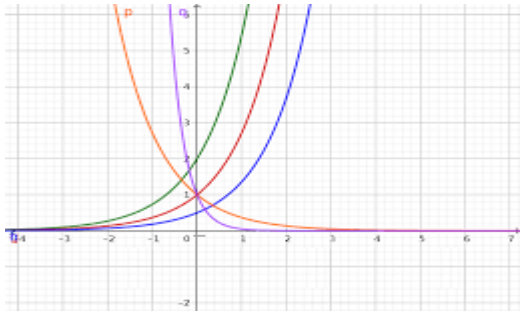
- La contraparte de la recursividad es la “iteración” o la resolución a través de una fórmula directa.
- En las funciones iterativas se resuelve un problema aplicando procedimientos de solución repetidamente y que no se considera la posible estructura interna del problema.
- En algunos casos la función recursiva es más eficiente, en otros la iterativa es mejor.



3.4.3 Otros aspectos de Recursividad

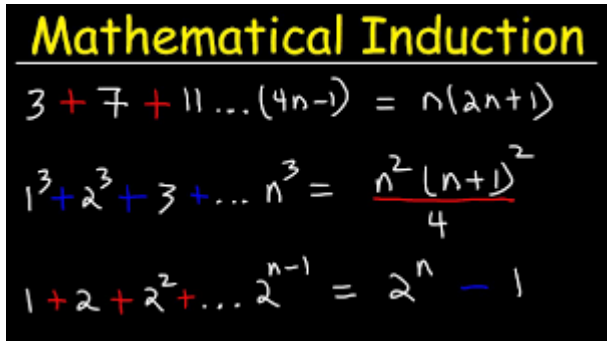
- El diseño de una función recursiva suele ser complicado y algunos de los problemas que ocurren frecuentemente son:
  - Creación de una lógica circular que ocasiona que la función no termine su ejecución.
  - Error en la definición del caso base.
  - El número de operaciones crece excesivamente

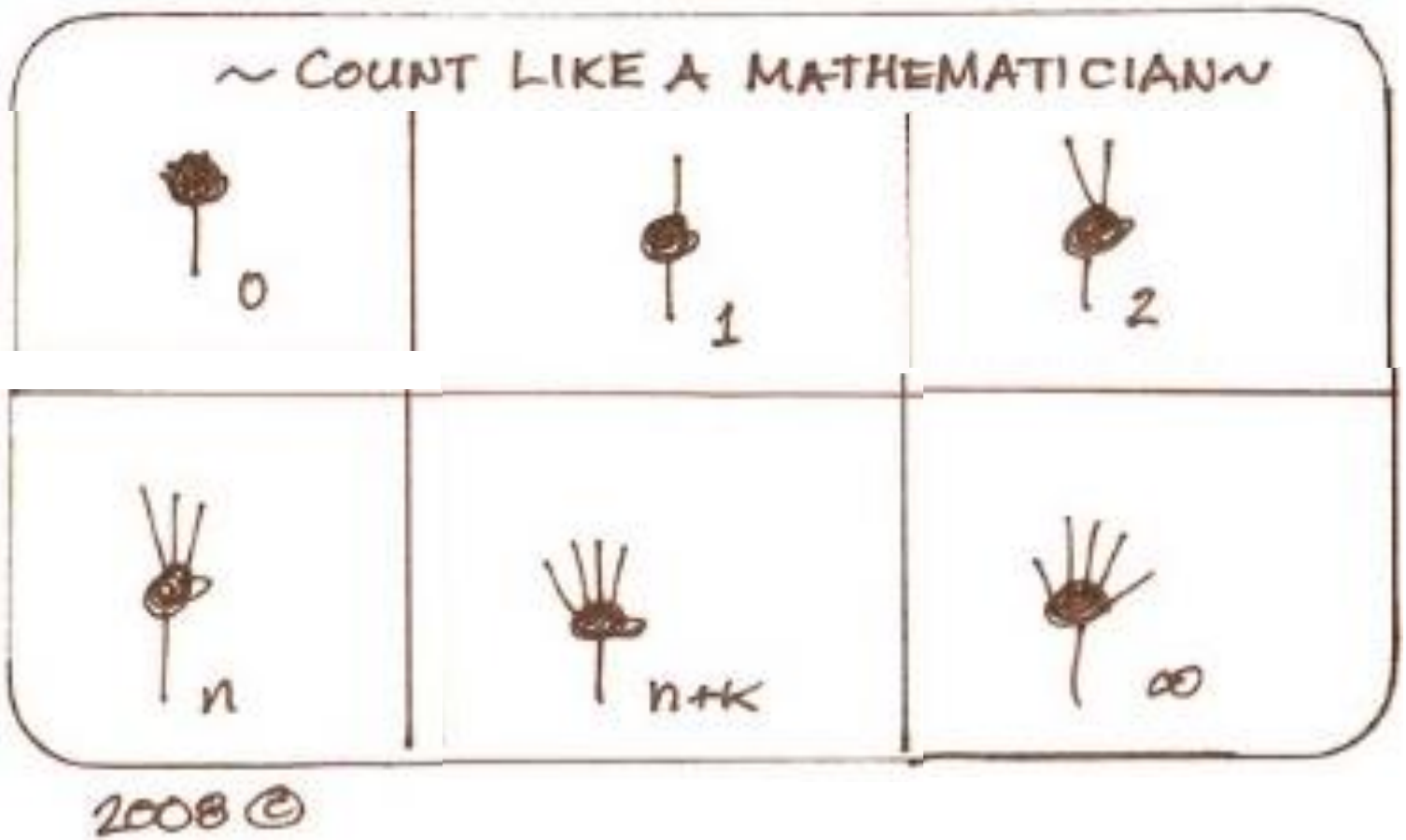




3.4.4 Recursividad e Inducción matemática

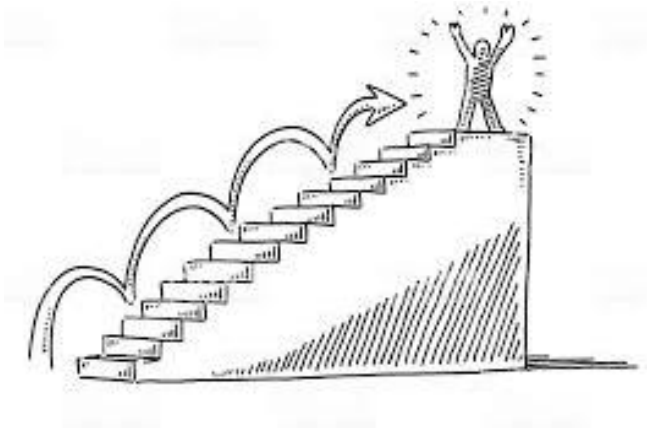
- La inducción matemática es la base de la recursividad.
- Las demostraciones por inducción, muestran que si se sabe que una afirmación es cierta para el caso mas pequeño y se puede demostrar que un caso implica el siguiente, entonces se sabe que la afirmación es cierta para todos los casos.
- Para probar que una definición recursiva es correcta se debe demostrar por inducción

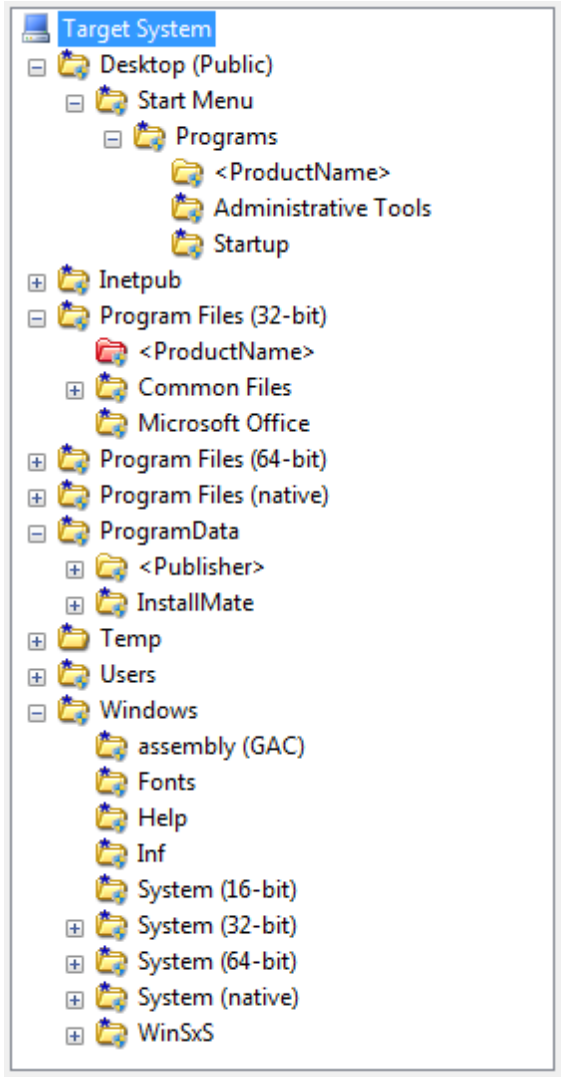



~ COUNT LIKE A MATHEMATICIAN ~
0
1
2
n
n+k
∞
2008 ©

3.4.5 Ejemplos cotidianos de recursividad

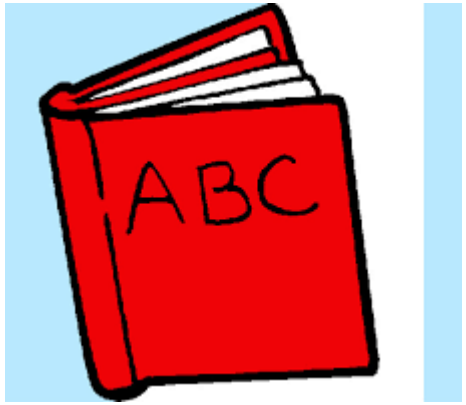
- Los archivos de una computadora suelen estar almacenados en directorios. Los usuarios pueden crear subdirectorios que almacenan archivos y otros directorios.
- Al descender o ascender por una escalera se realiza un recorrido recursivo ya que se realiza la misma acción hasta llegar a un caso base (no más escaleras)





3.4.5 Ejemplos cotidianos de recursividad

- Las palabras de los diccionarios se definen utilizando otras palabras.
- Al buscar, es posible que no se comprenda la definición y se tenga que buscar el significado de las palabras que contiene dicha definición.
- Como el diccionario es finito, llega un momento en el que se comprenden todas las palabras, que se caiga en una definición circular, o que alguna palabra no exista.



3.4.6 Funciones matemáticas recursivas

- **La suma de los N primeros enteros.**

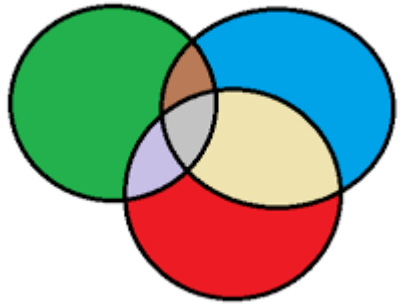
Definición Directa

$$S(N) = \frac{N(N+1)}{2}$$

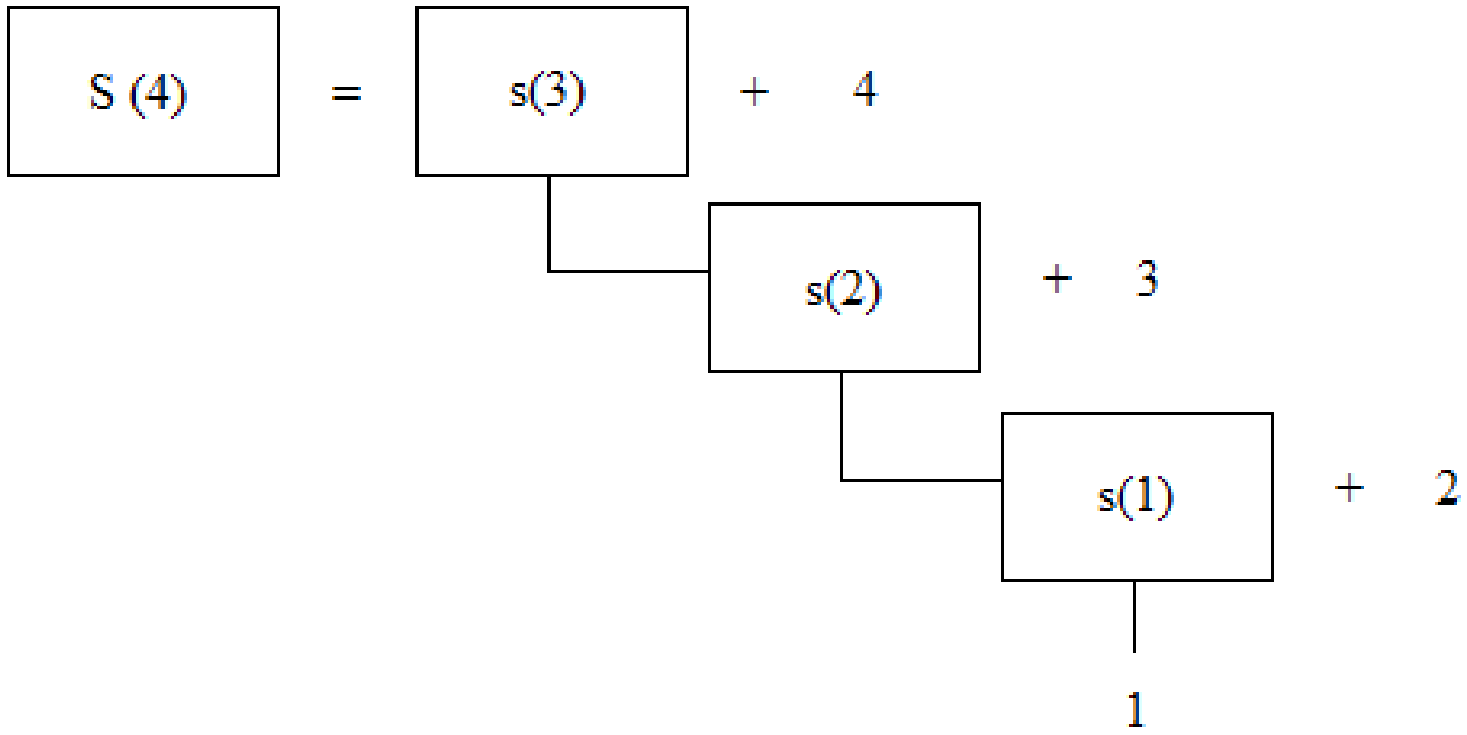
Definición Recursiva

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

Caso Base: S(1) = 1



para s(4)...



3.4.6 Funciones matemáticas recursivas

- **En código**

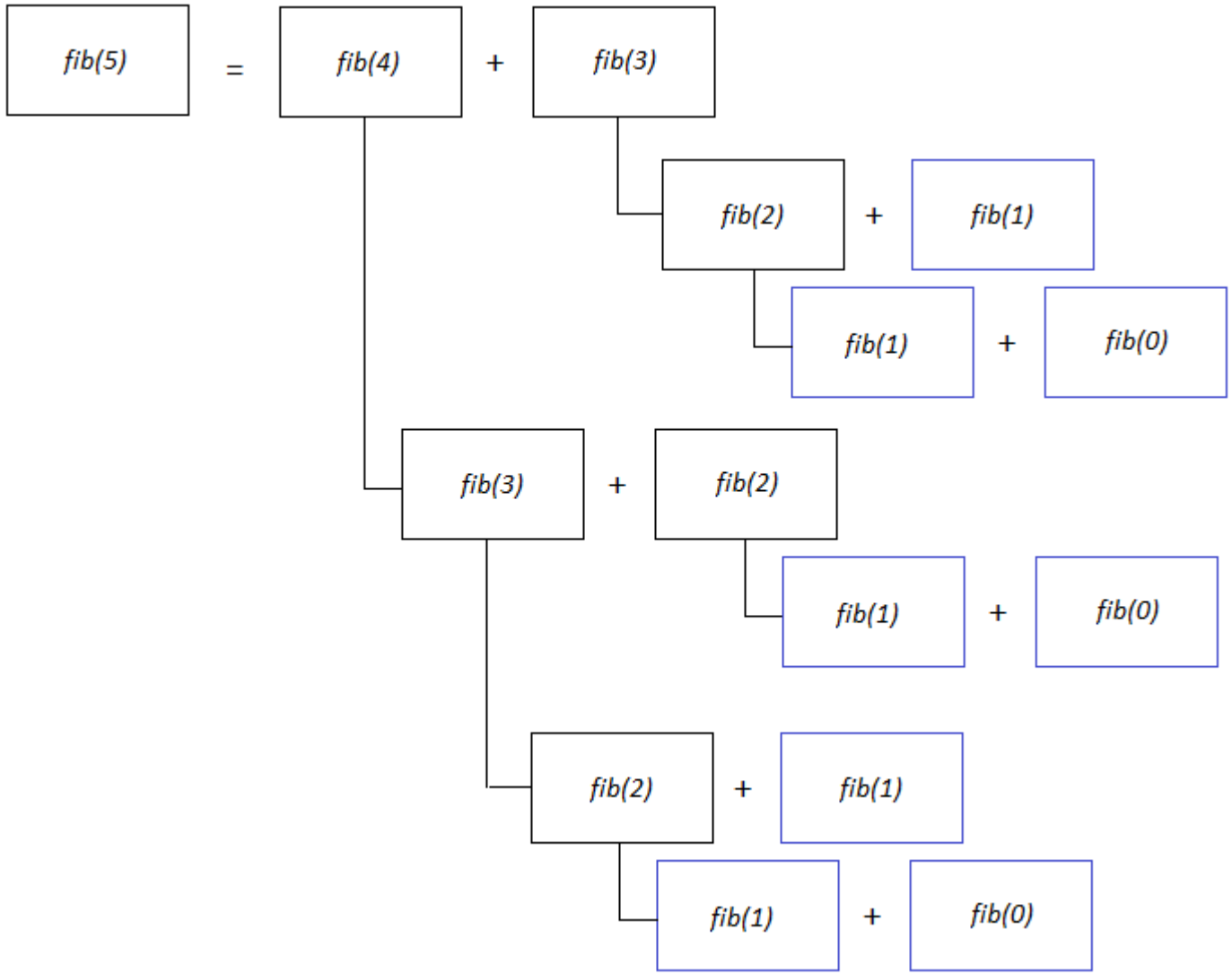
```
int sumaEnteros(int n){
    if (n==1)
        return 1;
    else
        return sumaEnteros(n-1) + n
}
```

3.4.6 Funciones matemáticas recursivas

- **Números de Fibonacci: 1,1,2,3,5,8,13,21,34…**

$$fib(n) = fib(n-1) + fib(n-2)$$

Caso Base: fib(0) = 0, fib(1) = 1


fib(5)
=
fib(4)
+
fib(3)
fib(2)
+
fib(1)
fib(1)
+
fib(0)
fib(3)
+
fib(2)
fib(1)
+
fib(0)
fib(2)
+
fib(1)
fib(1)
+
fib(0)

3.4.6 Funciones matemáticas recursivas

- **En código**

```
int fibonacci(int n){
    if (n<=2)
      return 1;
   else
      return fibonacci(n-1)+ fibonacci(n-2)
```

3.4.6 Funciones matemáticas recursivas

- **La suma de 2 números.**

$$suma(a,b) = \begin{cases} a & b = 0 \\ suma(a-1, b+1) & b < 0 \\ suma(a+1, b-1), & b > 0 \end{cases}$$

3.4.6 Funciones matemáticas recursivas

- **En código**

```
int sumaRec (int num1, int num2){
   if(num2 == 0)
        return num1;
   else if(num2 < 0)
        return sumaRec(num1-1, num2+1);
   else
        return sumaRec(num1+1, num2-1);
}
```

3.4.6 Funciones matemáticas recursivas

- **La potencia de un numero (exponente positivo)**

$$potencia(b,n) = \begin{cases} 1 & n = 0 \\ b \cdot\ potencia(b, n-1), & n > 0 \end{cases}$$

3.4.6 Funciones matemáticas recursivas

- **Factorial de un número**

Iterativo:

$$fact(n) = n(n-1)(n-2)...(n-(n-1))$$

Recursivo:

$$fact(n) = n \cdot fact(n-1)$$

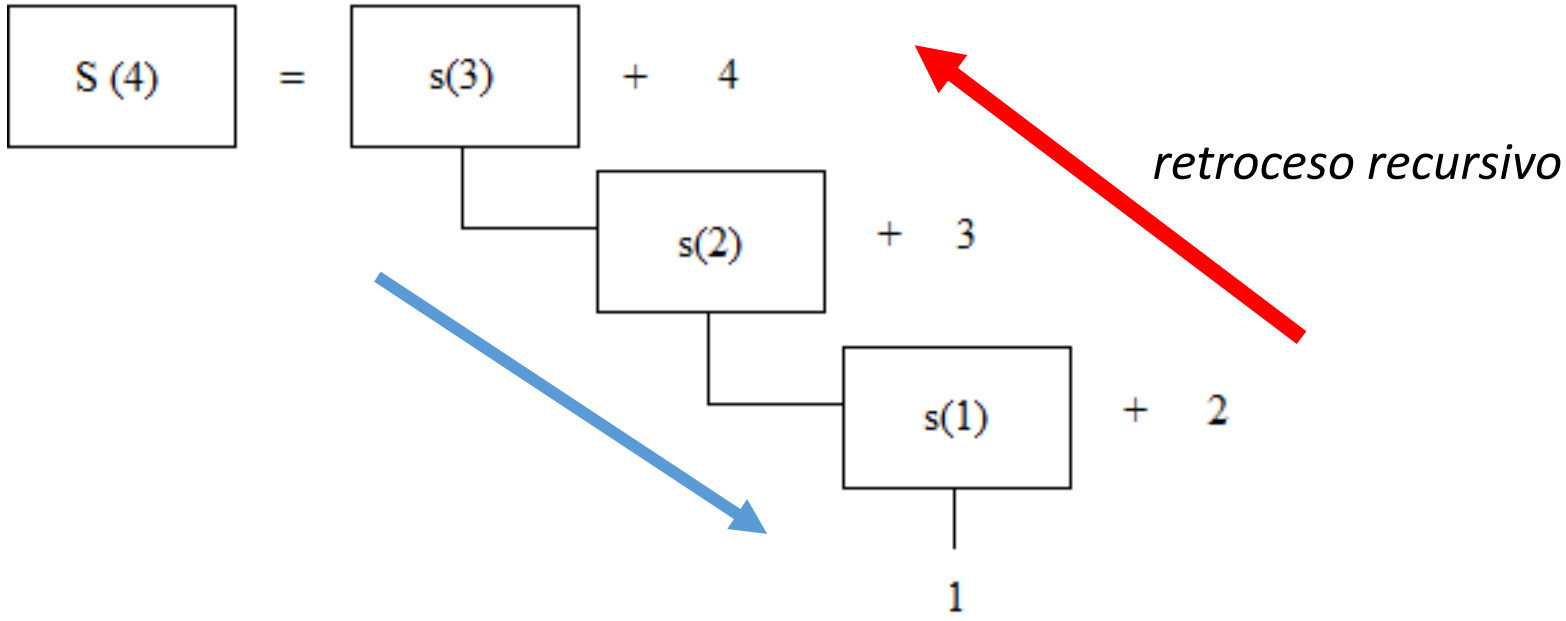
Caso Base: fact(0)=1

3.4.7 Retroceso Recursivo

- El retroceso recursivo implica realizar las operaciones en orden inverso siguiendo la forma **“bottom-up”.**
- Algunos problemas que se resuelven de manera recursiva únicamente realizan el seguimiento de los problemas de arriba hacia abajo.

3.4.7 Retroceso Recursivo

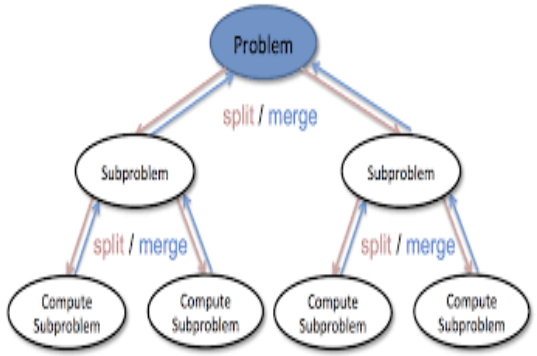
- En contraparte, las otras soluciones recusivas requieren un "retroceso" para realizar las operaciones o ejecutar las instrucciones de las funciones que se quedaron "en espera".

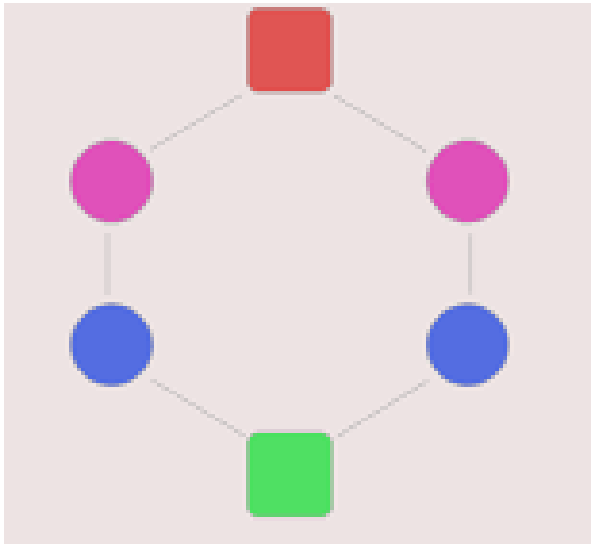


3.4.8 Ecuaciones de recurrencia

- Son expresiones en las que el término general de una sucesión se escribe en función de algunos términos anteriores.
- Reciben el nombre de ecuaciones de recurrencia o relaciones de recurrencia.
- El objetivo de estas relaciones es encontrar una forma de calcular cualquier término de una función recursiva sin necesidad de depender de valores anteriores.

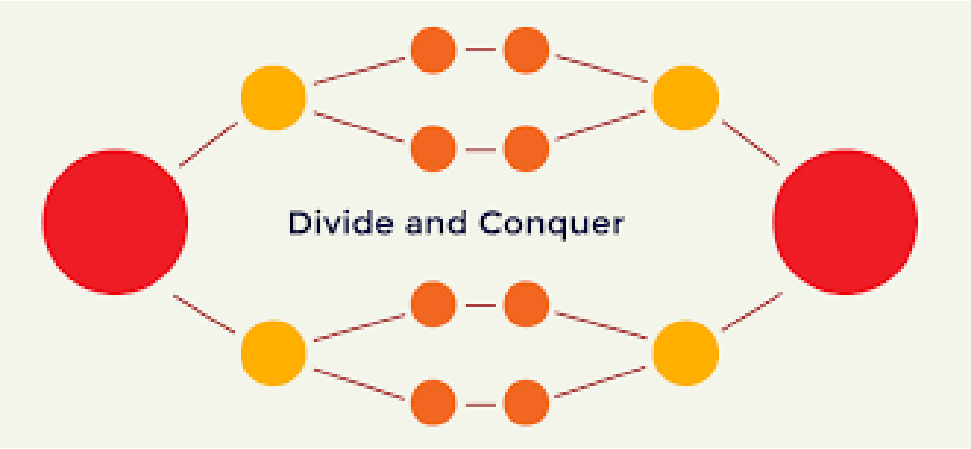
3.5 Divide y vencerás





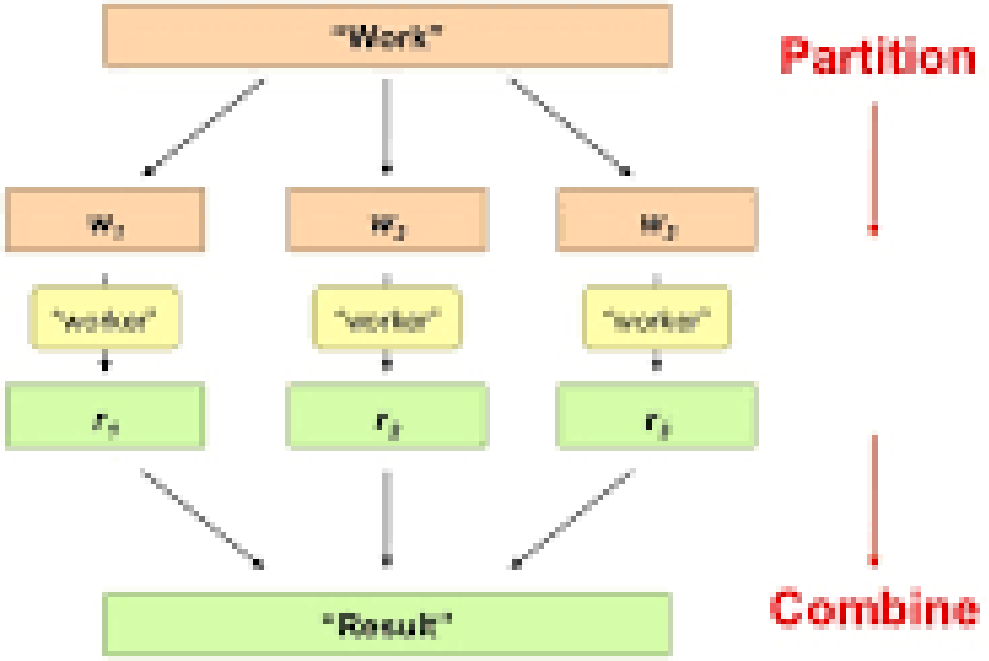
3.5.1 Divide y vencerás - Generalidades

- Además de ser una estrategia para construir algoritmos, es una técnica que suele ser considerada para resolver problemas en general.
- En el campo de los algoritmos consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo pero de menor tamaño.



3.5.2 Divide y vencerás - Descripción

- La idea es hacer una división del problema general en sub-problemas del mismo tipo pero de menor tamaño.
- Si después de hacer la “división”, los sub-problemas son relativamente grandes, se aplica de nuevo la técnica tantas veces como sean necesarias hasta suficientemente pequeños para resolverlos directamente.



3.5.2 Divide y vencerás - Descripción

- El resultado del diseño de estos algoritmos suele ser simple (pocas instrucciones), lo que da lugar a una mejor legibilidad, mayor facilidad en la depuración y análisis del algoritmo.
- La técnica de divide y vencerás suele ir dela mano con la recursividad.

3.5.2 Divide y vencerás - Descripción

- Otra consideración importante a la hora de diseñar algoritmos divide y vencerás, es el reparto de la carga entre los sub-problemas, puesto que es importante que la división en sub-problemas se haga de la forma más equilibrada posible.

3.5.3 Divide y vencerás – Forma general

La técnica consiste a grandes rasgos en los siguientes pasos:

1.- Se debe plantear el problema de tal forma que se pueda descomponer en *k* sub-problemas del mismo tipo pero de menor tamaño. A esta parte del procedimiento se le conoce como división.

2.- Se deben resolver los *k* problemas de manera independiente.

3.- Combinar las soluciones de cada uno de los k problemas para construir la solución al problema original.

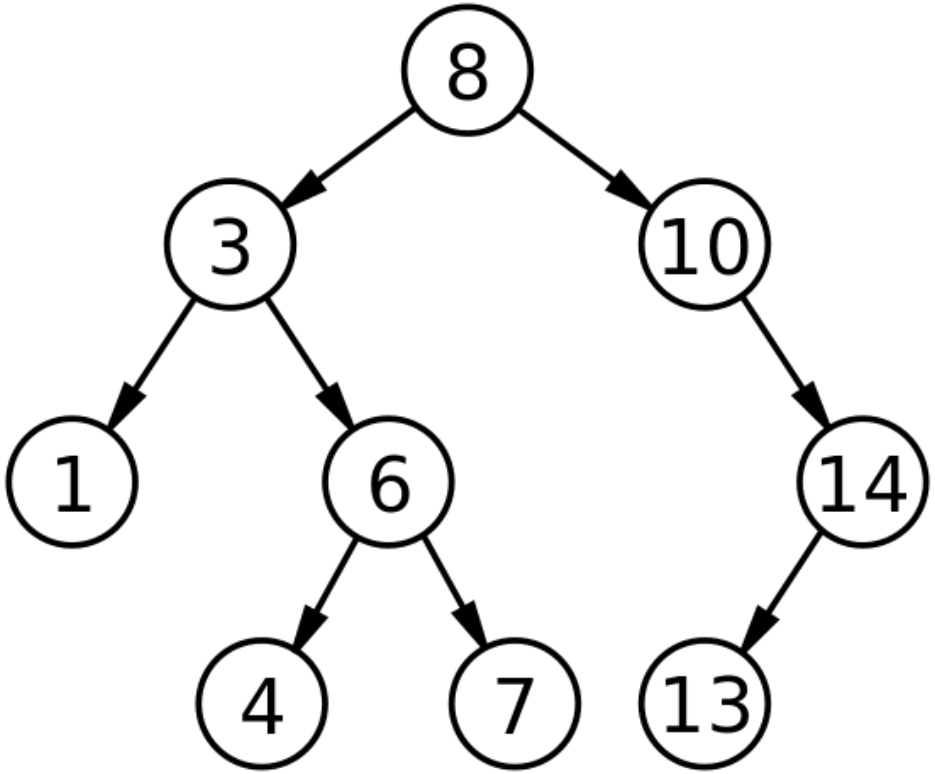
3.5.3 Divide y vencerás – Forma general

```
DV(x)
if (x es suficientemente pequeño) {
    return algoritmo_específico(x);
} else {
    descomponer x en {x_1,…, x_k}
    for i = 1 to k
        y_i ← DV(x_i)
    y ← recombinar (y_1, …, y_k)
    return y;
}
```

3.5.4 Divide y vencerás – Ejemplos

```
busquedaBinaria (lista, x)
    izq ← 1
    der ← longitud (lista)
    mientras izq ≤ der
        medio ← (der + izq) /2
        si x = lista [medio]
            regresar medio
        otro si x < lista [medio]
            der ← medio − 1
        otro si x > lista [medio]
            izq ← medio + 1
    regresar −1
```

3.5.4 Divide y vencerás – Ejemplos



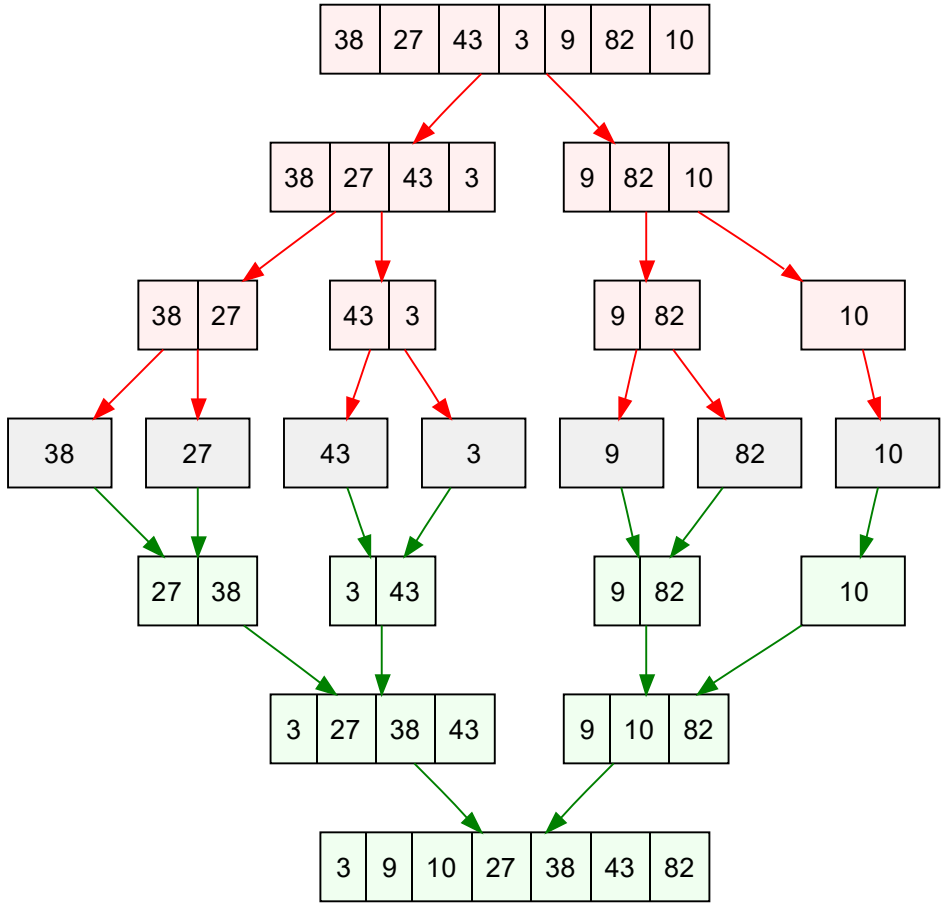
3.5.4 Divide y vencerás – Ejemplos

Ordenamiento por Intercalación (Mergesort)

Dada una lista desordenada, el procedimiento para ordenarla es el siguiente:

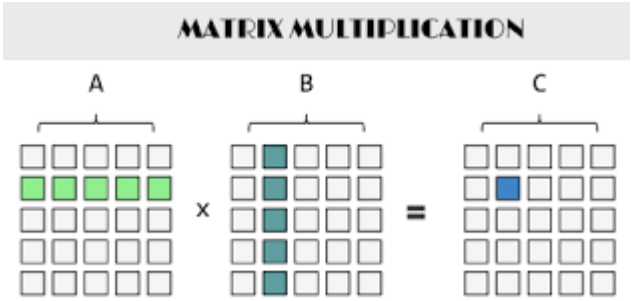
- ✓ Dividir la lista de n elementos en dos sub-listas de n/2 elementos cada una
- ✓ Ordenar las dos sub-listas utilizando mergesort
- ✓ Combinar (intercalar) las dos sub-listas ordenadas para obtener la lista ordenada final.

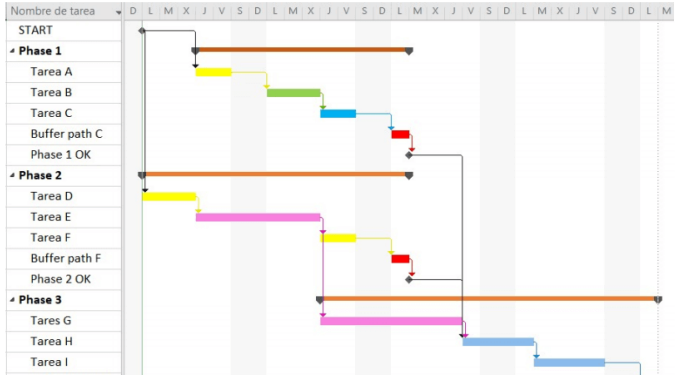
3.5.4 Divide y vencerás – Ejemplos



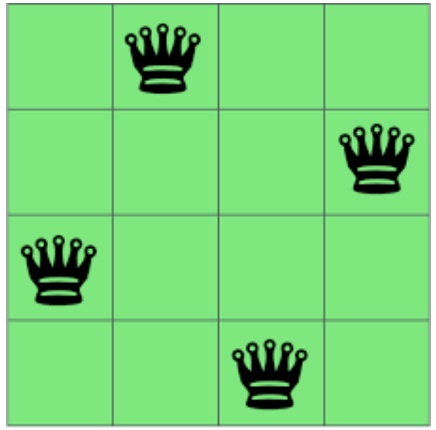
3.5.4 Divide y vencerás – Ejemplos

- Algoritmos de ordenamiento
- Multiplicación de matrices
- Par de puntos más cercano
- FFT (Transformada Rápida de Fourier)
- Calendarización.



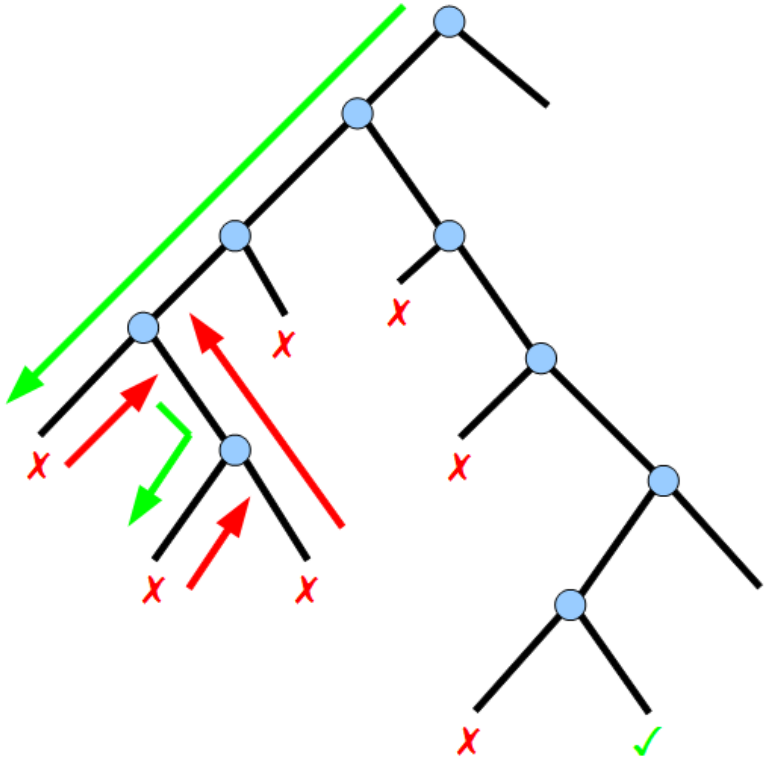


3.6 BACKTRACK



3.6.1 Backtrack - Descripción

- Backtracking (o búsqueda atrás) es una estrategia de construcción de algoritmos que permite realizar una búsqueda sistemática a través de un espacio de soluciones.
- El objetivo es encontrar soluciones parciales a medida que progresa el recorrido, estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa.

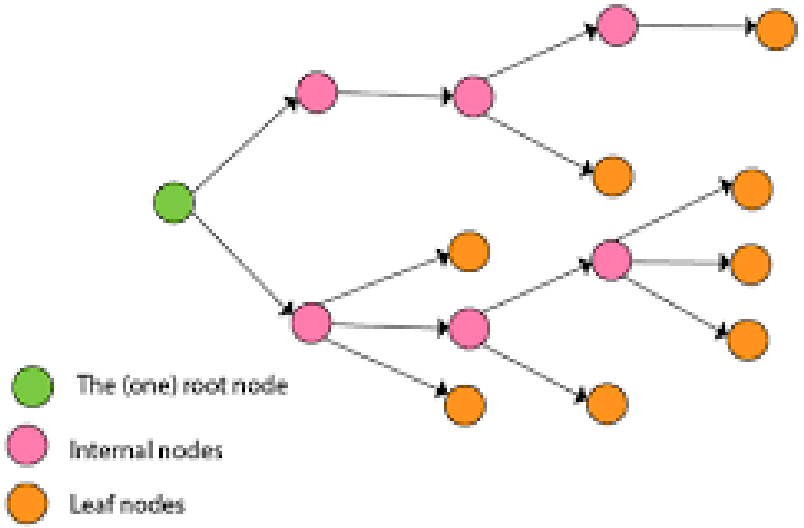


3.6.1 Backtrack - Descripción

- La búsqueda de solución a un problema no tiene éxito si en alguna etapa, la solución parcial construida hasta el momento no puede continuar.
- En tal caso, la búsqueda de la solución *regresa hacia "atrás"* e intenta otra opción de camino.

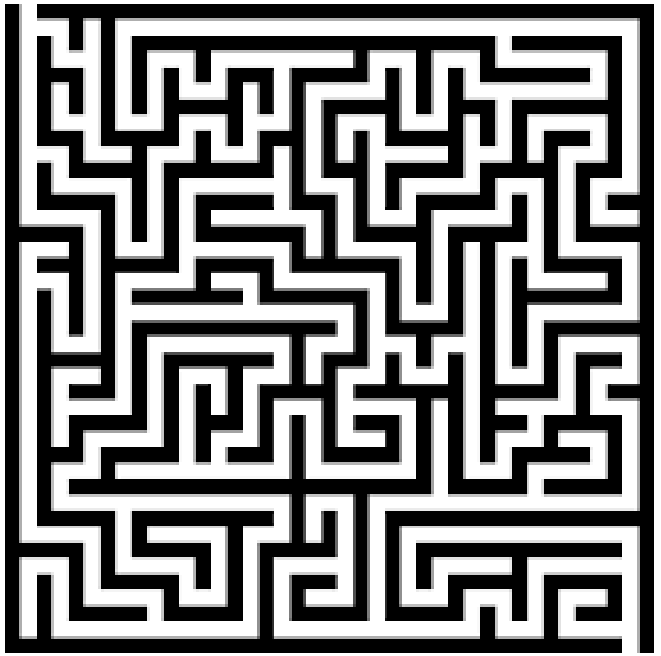
3.6.1 Backtrack - Descripción

- La construcción de estrategias backtrack es similar a “greedy” ya que en cada paso se busca un posible camino que conduce a una solución.
- La diferencia radica en que en este caso si es posible regresar y buscar un camino diferente



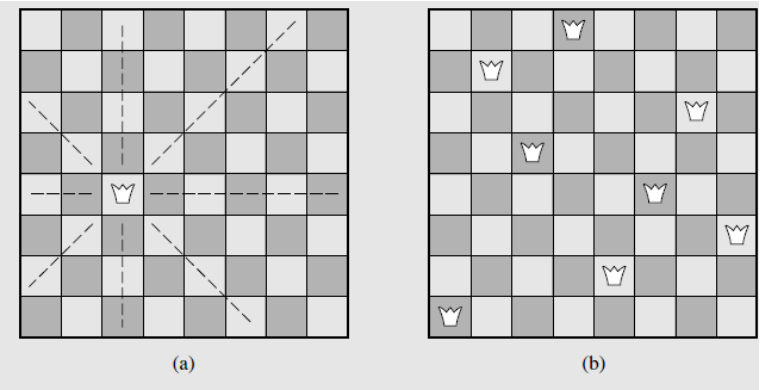
3.6.2 Backtrack - Ejemplos

- **Recorrer un laberinto hasta encontrar la salida.**
  - ✓ No se puede “dividir el problema
  - ✓ No hay mucha información al inicio
  - ✓ Se requiere tomar decisiones conforme se avanza o progresa en el recorrido.
  - ✓ Si esas decisiones resultan en un camino equivocado, se puede regresar e intentar otra solución

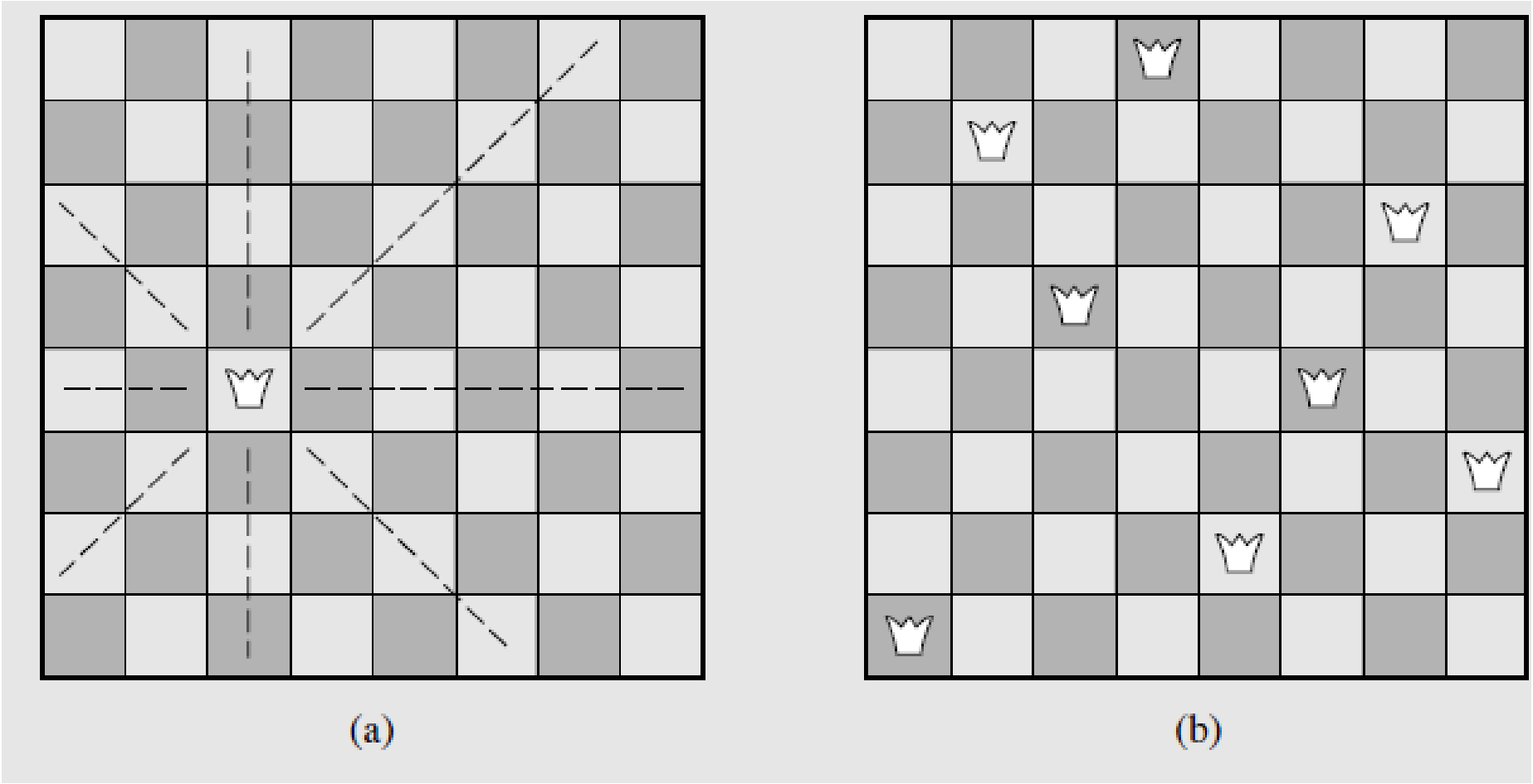


3.6.2 Backtrack - Ejemplos

- **Problema de la “n” reinas**
  - ✓ Dado un tablero de ajedrez de n x n, colocar n reinas de tal manera que no se puedan ver (que no se maten) entre ellas
  - ✓ No hay mucha información al inicio
  - ✓ Se requiere tomar decisiones sobre la colocación de las reinas.
  - ✓ Si esas decisiones resultan en una solución equivocada, se puede regresar e intentar otra solución



3.6.2 Backtrack - Ejemplos



(a)

(b)

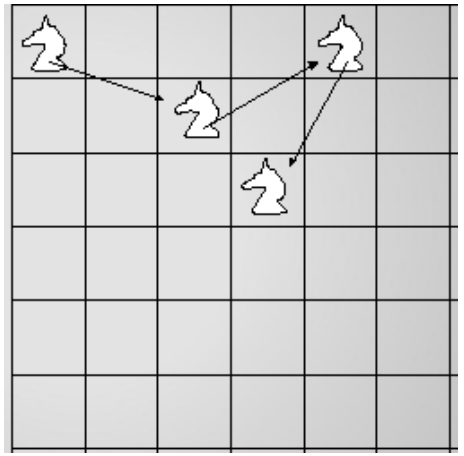
3.6.2 Backtrack - Ejemplos

- **Sudoku**

7	8		4			1	2	
6				7	5			9
			6		1		7	8
		7		4		2	6	
		1		5		9	3	
9		4		6				5
	7		3				1	2
1	2				7	4		
	4	9	2		6			7

3.6 Backtrack - Ventajas y desventajas

- La mayor ventaja de la estrategia Backtrack es la flexibilidad que proporciona, ya que dependiendo el enfoque del problema puede encontrar una o todas las soluciones al problema.
- La desventaja es que dependiendo de la dificultad del problema, puede tomar mucho tiempo en resolverse.



3.6.4 Backtrack - Aplicaciones

- Se usa en la implementación de los lenguajes de programación, principalmente de inteligencia artificial
- Se utiliza en análisis sintácticos de los compiladores
- Algunos algoritmos de teoría de grafos también se pueden analizar con este enfoque