



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Estructuras de datos y algoritmos I
Tema 3:
ESTRATEGIA PARA CONSTRUIR ALGORITMOS



3 Estrategia para construir algoritmos

Objetivo: Aplicar diversas técnicas como la recursividad para construir algoritmos.

3 Estrategia para construir algoritmos

3.1 Algoritmos de búsqueda exhaustiva y fuerza bruta.

3.2 Algoritmos ávidos.

3.3 Recursividad.

3.3.1 El concepto de recursividad.

3.3.2 Funciones matemáticas de recursividad.

3.3.3 Uso de relaciones de recurrencia para analizar algoritmos recursivos.

3.3.4 Retroceso recursivo.

3.3.5 Implementación de la recursividad.

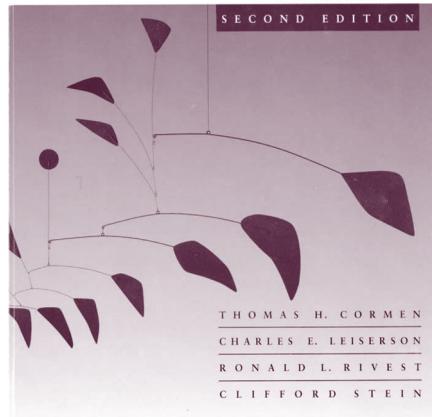
3.4 Top-down y bottom-up.

3.5 Divide y vencerás.

3.6 Backtrack.

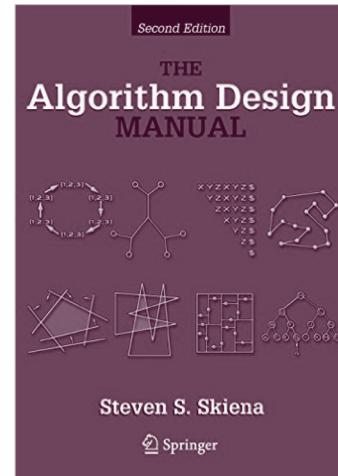
Bibliografía

INTRODUCTION TO ALGORITHMS



Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, McGraw-Hill.

Bibliografía



The Algorithm Design Manual. Steven S. Skiena, Springer.



3 Estrategias para construir algoritmos.

3 Estrategias para construir algoritmos.

Tener una base sólida de conocimientos y técnicas algorítmicas es una característica que separa a los diseñadores expertos de los novatos.

Con las capacidades tecnológicas actuales se puede cumplir con las tareas solicitadas sin saber mucho de algoritmos, pero con buenos conocimientos de algoritmos se puede hacer mucho más.



Un buen algoritmo, de manera general, debe cumplir con 3 propiedades fundamentales:

- 1) Debe ser correcto (cumplir con el objetivo).**
- 2) Debe ser eficiente (cumplir el objetivo en el menor tiempo posible).**
- 3) Debe ser fácil de implementar.**

El enfoque con el que se construye un algoritmo involucra, necesariamente, al método científico:

- **Observación:** algún problema del real.
- **Hipótesis:** un modelo consistente con la observación.
- **Predicción de eventos:** basados en la hipótesis.
- **Verificación:** comprobar la hipótesis ahondando en las predicciones.
- **Validación:** repetir los eventos hasta que la hipótesis concuerden.

Un usuario selecciona un número entre 1 y 100, se desea encontrar el número elegido por el usuario.

¿Cómo se puede adivinar el número seleccionado por el usuario?

¿Cuál es, en promedio, el número de intentos con el que se puede encontrar el número seleccionado?

Un candado posee una combinación de 4 dígitos pero no se tiene la clave numérica para abrirlo.

¿Cómo se puede obtener la combinación correcta?

¿Cuál es, en promedio, el número de intentos con el que se puede encontrar la combinación?



Los algoritmos se pueden clasificar con base en la manera en que optimizan el código, algunas clasificaciones de este tipo son:

- **Algoritmos de búsqueda exhaustiva y fuerza bruta.**
- **Top-down y bottom-up.**
- **Algoritmos ávidos.**
- **Divide y vencerás.**
- **Recursividad.**
- **Backtrack.**

DICTIONARY ATTACK!



3.1 Algoritmos de búsqueda exhaustiva y fuerza bruta.

3.1 Algoritmos de búsqueda exhaustiva y fuerza bruta.

La búsqueda exhaustiva es una técnica general para la resolución de problemas, realizando un proceso sistemático dentro de un espacio de soluciones.

Un algoritmo de fuerza bruta consiste en probar todas las posibles combinaciones dentro del espacio de soluciones, es decir, es un algoritmo de búsqueda exhaustiva.

Ataque de diccionario

El ataque por diccionario (dictionary attack) es un tipo de ataque informático que utiliza un diccionario de palabras para llevar a cabo su cometido.

El ataque consiste en probar las palabras que existen en un archivo (diccionario) hasta obtener la contraseña (si es que ésta se encuentra en el archivo). Por lo regular, los diccionarios contienen palabras que han sido utilizadas de manera recurrente como contraseñas.

Las 25 contraseñas más populares durante el año 2015 fueron:

- | | | |
|--------------|----------------|----------------|
| 1) 123456 | 10) baseball | 19) letmein |
| 2) password | 11) welcome | 20) login |
| 3) 12345678 | 12) 1234567890 | 21) princess |
| 4) qwerty | 13) abc123 | 22) qwertyuiop |
| 5) 12345 | 14) 111111 | 23) solo |
| 6) 123456789 | 15) 1qaz2wsx | 24) passw0rd |
| 7) football | 16) dragon | 25) starwars |
| 8) 1234 | 17) master | |
| 9) 1234567 | 18) monkey | |

Normalmente, un diccionario con las contraseñas más utilizadas es económicamente elevado. Sin embargo, es posible crear un diccionario, para ello solo se deben conocer los caracteres válidos (a-z, A-Z, 0-9, etc.) y el número de dígitos permitidos (longitud de la contraseña).

Por lo tanto, para generar todas las posibles combinaciones (con ello contraseñas) solo se deben conocer todas las instancias de entrada válidas para formar la contraseña.

Suponiendo que el conjunto de entrada válido es el abecedario en minúsculas $E = \{a, b, c, d, ..., w, x, y, z\}$ y la longitud de la contraseña es de 4 dígitos, un algoritmo para generar todas las posibles combinaciones es el siguiente:

```
1 FUNC crearDico (abecedario[]: CARACTER)
2   MIENTRAS d1< 27 HACER
3     MIENTRAS d2< 27 HACER
4       MIENTRAS d3< 27 HACER
5         MIENTRAS d4< 27 HACER
6           ESCRIBIR d1d2d3d4
7           FIN MIENTRAS
8         FIN MIENTRAS
9       FIN MIENTRAS
10      FIN MIENTRAS
11    FIN
```

El algoritmo anterior permite crear las ~~ene~~ posibilidades para el conjunto de entrada, generando así un diccionario de posibles contraseñas.

Una vez generado el diccionario con ~~ene~~ palabras, es posible realizar las comparaciones necesarias hasta encontrar la contraseña buscada. Esta búsqueda exhaustiva se conoce como ataque de diccionario por fuerza bruta.

The greedy algorithm used to give change.
Amount owed: 41 cents.

Subtract Quarter
 $41 - 25 = 16$



Subtract Dime
 $16 - 10 = 6$



Subtract Nickel
 $6 - 5 = 1$



Subtract Penny
 $1 - 1 = 0$



3.2 Algoritmos ávidos

3.2 Algoritmos ávidos

Los algoritmos para optimizar problemas generalmente siguen una secuencia de pasos con un conjunto de decisiones en cada paso.

Un algoritmo ávido o voraz (Greedy algorithm) siempre ejecuta la decisión que parece mejor en el momento, es decir, crea una decisión local óptima con la esperanza de que esa elección contribuya a la solución óptima general.

Código Huffman

Se refieren a una técnica de compresión ampliamente usada y muy efectiva. Puede llegar a comprimir entre 20 y 90 % dependiendo de las características de los datos a comprimir.

El algoritmo ávido de Huffman utiliza una tabla de frecuencia de ocurrencia de caracteres para crear una manera óptima de representar cada carácter como una cadena binaria.

Ejemplo 2

Dado un archivo de datos de 100,000 caracteres (que en la memoria ocuparía 800,000 bits), se desea almacenar de manera compacta. Se puede observar que la frecuencia de caracteres en el archivo es la siguiente:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5

Ejemplo 2

De la tabla anterior se puede observar que solo aparecen 6 caracteres diferentes y que el carácter más repetitivo es ‘a’ (45,000 veces).

Para representar el archivo de información se considera el diseño de un código binario, donde cada carácter es representado por una cadena binaria única.

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud fija.	000	001	010	011	100	101

Ejemplo 2



Utilizando el código de longitud fija se observa que se necesitan 3 bits para representar los seis caracteres: a=000, b=001, c=010, d=011, e=100, f=101.

Dado que el archivo consta de 100,000 caracteres, este método requiere de 300,000 bits para codificar el archivo completo.

Ejemplo 2

También se puede considerar un código de longitud variable con la siguiente distribución:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud variable.	0	101	100	111	1101	1100

Un código de longitud variable hace un mejor trabajo que uno de longitud fija, debido a que a los caracteres frecuentes les asigna un código de palabra corto y a los caracteres que no tienen tanta frecuencia les asigna un código de palabra largo.

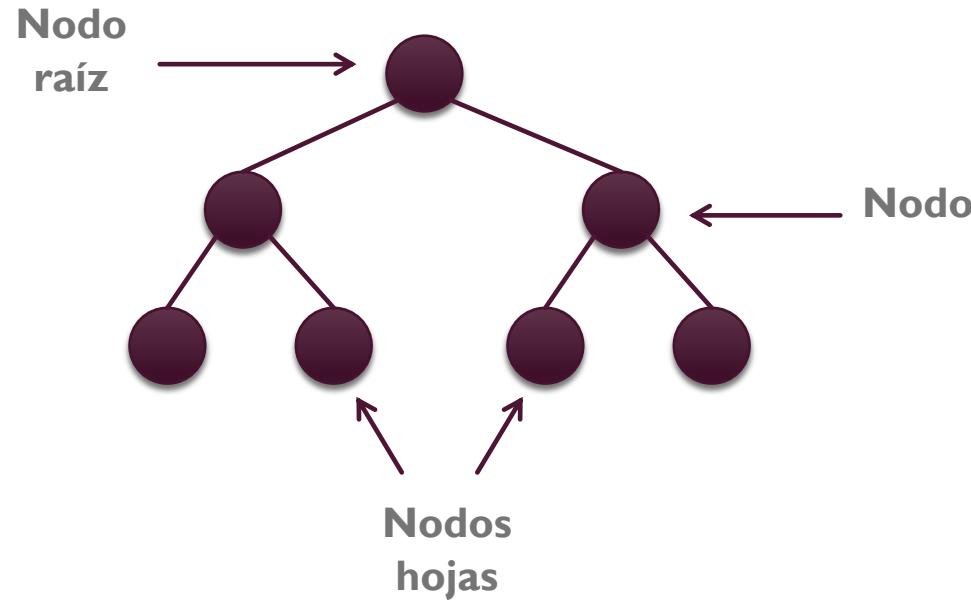
El código de longitud variable requiere $(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000 = 224,000$ bits para codificar el archivo, genera un ahorro de un poco más del 25%.

Ejemplo 2

La codificación variable se considera una codificación prefija, ya que ningún código de palabra es prefijo de algún otro código. La codificación prefija simplifica el proceso de decodificación.

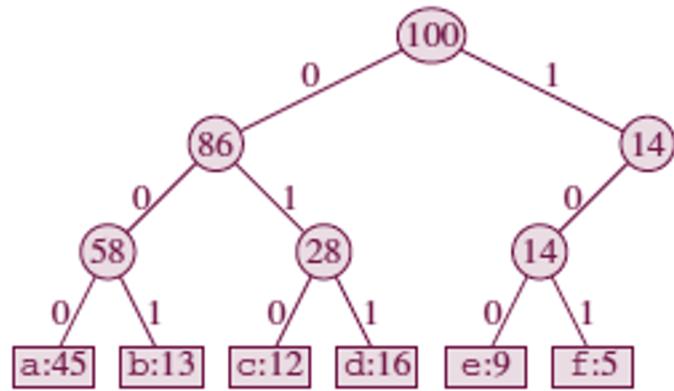
Por ejemplo, la cadena 001011101 se puede descodificar como 0.0.101.1101 que resulta en aabe.

La decodificación se puede representar mediante una estructura de árbol. Una estructura de árbol posee un nodo raíz (el primer nodo del árbol) y varios nodos hojas (los últimos nodos).

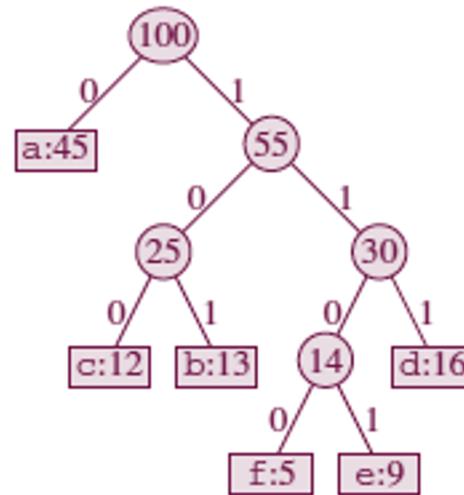


Ejemplo 2

A partir de un árbol se puede interpretar el código de palabra de un carácter como la ruta que existe desde el nodo raíz hasta la hoja que contiene el carácter, donde 0 significa ir a la izquierda y 1 significa ir a la derecha.



Longitud fija



Longitud
variable

Dado un árbol A que corresponde a un código prefijo, se puede calcular fácilmente el número de bits requeridos para codificar un archivo.

Para cada carácter c en el alfabeto C, se tiene $f(c)$ (frecuencia de aparición del carácter c en el archivo) y $p_A(c)$ (profundidad de la hoja que posee el carácter c dentro del árbol A).

Ejemplo 2

El número de bits que se requieren para condificar el archivo está dado por:

$$B(A) = \sum f(c) p_A(c)$$

con $c \in C$ que se define como el costo del árbol A.

Huffman inventó un algoritmo ávido que construye un código prefijo óptimo.

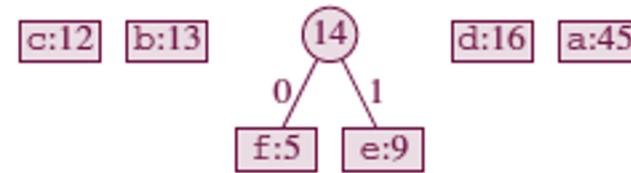
A continuación se presenta el pseudocódigo del algoritmo de Huffman. Se asume que C es un conjunto de n caracteres y que cada carácter $c \in C$ es un objeto con una frecuencia $f(c)$. El algoritmo crea un árbol A que corresponde a un código óptimo de arriba hacia abajo.

```
I FUNC HUFFMAN(C)
2     n ← |C|                                ** longitud de C
3     Q ← C                                  ** elementos de C
4     i ← 1                                   ** índice
5     MIENTRAS i < n-1 HACER
6         crearNodo(z)
7         left[z]← x ← extraerMen(Q)
8         right[z]← y ← extraerMen(Q)
9         f [z]← f [x] + f [y]
10        INSERT(Q, z)
11    FIN_MIENTRAS
12    return EXTRACT-MIN(Q)
13 FIN_FUNC
```

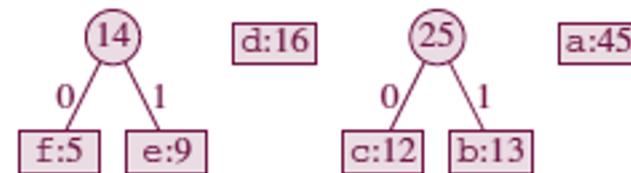
Ejemplo 2

1 f:5 e:9 c:12 b:13 d:16 a:45

2

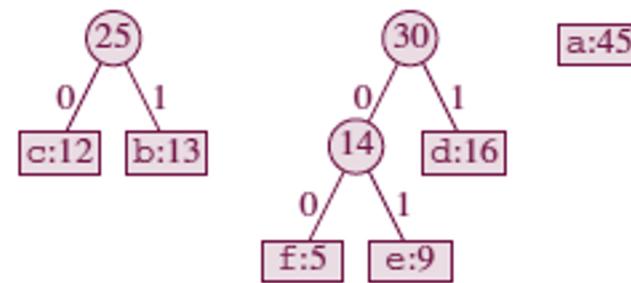


3

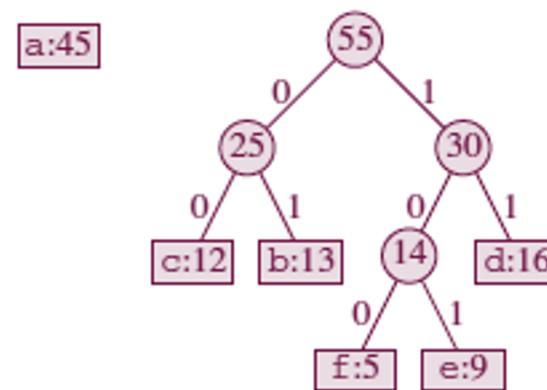


Ejemplo 2

4

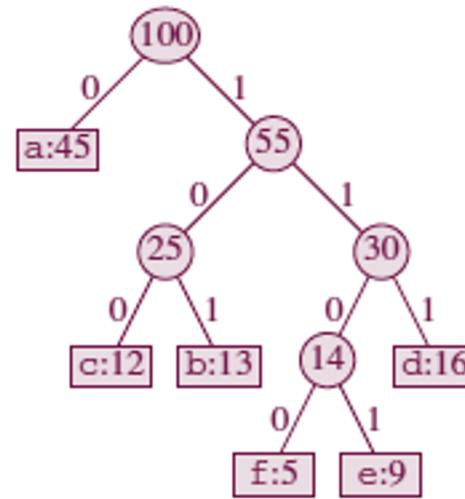


5



Ejemplo 2

6



Así como lo hizo el algoritmo de Huffman, un algoritmo ávido va buscando la decisión que parece mejor en el momento, esperando con ello optimizar la solución general.



3.3 Recursividad

3.3 Recursividad

Cuando un algoritmo realiza llamadas a sí mismo se dice que éste es un algoritmo recursivo.

La recursividad se puede describir como una función en términos de un valor de entrada diferente en cada llamada.



Un algoritmo que en vez de llamarse a si mismo repite en una serie de código de manera cíclica se conoce como algoritmo iterativo.

Generalmente, un algoritmo recursivo resulta más corto que un algoritmo iterativo para solventar el mismo problema. Por otro lado, un algoritmo recursivo puede ser convertido a un algoritmo iterativo, aunque esto puede afectar la eficiencia del mismo.

3.3.1 El concepto de recursividad.

El término recursividad se refiere al hecho de que una función se mande llamar a sí misma, es decir, dentro del cuerpo de la función se invoca a la misma función. Los algoritmos o funciones recursivas siguen 3 reglas básicas:

- **Hacen llamadas a sí mismas con instancias de entrada cada vez más pequeñas.**
- **No poseen ciclos de repetición.**
- **Poseen un número finito de casos base y un caso general.**

Un algoritmo iterativo se caracteriza porque utiliza estructuras de repetición que se ejecutan mientras la condición se mantenga afirmativa:

```
1  FUNC dec2bin (x: Entero) DEV vacío
2      MIENTRAS y > 0 HACER
3          ESCRIBIR y%2
4          y ← y/2;
5      FIN_MIENTRAS
6  FIN_FUNC
```

Un algoritmo recursivo se caracteriza porque utiliza estructuras de selección, además de que se manda llamar a sí mismo dentro del cuerpo del programa con instancias cada vez menores:

```
1      FUNC dec2bin (x: Entero) DEV vacío
2          SI x > 0 HACER
3              rec (x/2);
4              ESCRIBIR x%2
5          FIN_SI
6      FIN_FUNC
```

Realizar el pseudocódigo de un algoritmo iterativo. Después, transformar el algoritmo iterativo en un algoritmo recursivo.

NOTA. No todos los algoritmos pueden ser recursivos, pensar en alguno que sí lo sea.

3.3.2 Funciones matemáticas de recursividad

La recursividad constituye un concepto fundamental en las matemáticas y en las ciencias de la computación.

Los razonamientos recursivos encuentran su base en las matemáticas, ya que son necesarias para definir la esencia del algoritmo (ecuación).

Sucesión de Fibonacci (recursiva)

Como ya se había mencionado, la sucesión de Fibonacci consiste en una serie de números que, iniciando en cero y uno, el número siguiente es la suma de los números dos anteriores. Por lo tanto, se puede representar en forma matemática de la siguiente manera:

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

Normalmente, los algoritmos recursivos son más sencillos que los algoritmos iterativos, ya que éstos se componen de un caso base y de la fórmula matemática del problema a resolver, sin embargo pueden tener problemas de eficiencia para instancias grandes.

Para el caso de Fibonacci, la sucesión se puede calcular siempre y cuando el número dado (n) sea mayor a 1 ó 0 (casos base).

El algoritmo que permite calcular el enésimo número de la serie de Fibonacci de manera recursiva es el siguiente:

```
1      FUNC Fibonacci (ene:ENTERO) DEV ENTERO
2          SI (ene = 0):
3              DEV 0
4          SI (ene = 1):
5              DEV 1
6          DEV fibonacci(ene - 1) + fibonacci(ene - 2)
7      FIN_FUNC
```

3.3.3 Uso de relaciones de recurrencia para analizar algoritmos recursivos

Una ecuación de recurrencia es una expresión que define una sucesión, en la cual cada elemento se determina por otros elementos más sencillos (los anteriores), que incluyen condiciones iniciales.

Una ecuación de recurrencia sirve para medir la complejidad del algoritmo en cuestión y, con ello, estimar los recursos utilizados en tiempo de ejecución.

Ecuación de recurrencia lineal de primer orden

Sea la ecuación de recurrencia $a_n = 3a_{n-1}$ con $n \geq 1$ y el primer término de la serie $a_0 = 5$, obtener la solución para la ecuación de recurrencia descrita.

Ejemplo 6

Dada la ecuación $x_n = 3x_{n-1}$, con $n \geq 1$ y $x_0 = 5$:

$$a_0 = 5$$

$$a_1 = 3 * a_0 = 3(5)$$

$$a_2 = 3 * a_1 = 3 * (3a_0) = 3 * 3 * 5 = 3^2 * 5$$

$$a_3 = 3 * a_2 = 3 * (3a_1) = 3 * 3 * 3 * 5 = 3^3 * 5$$

...

Por lo tanto, la solución de la ecuación de recurrencia es:

$$x_n = 3^n(5).$$

Ejemplo 6

En general, la solución de una ecuación de recurrencia de primer orden con $n \geq 1$ y $x_0 = A$ es:

$$a_n = A * r^n$$

Ecuación de recurrencia lineal de segundo orden

Dada la ecuación de recurrencia lineal de segundo orden

$$c_n a_n + c_{n-1} a_{n-1} + c_{n-2} a_{n-2} = 0 \quad (1)$$

con $n \geq 2$, se busca una solución de la forma

$$a_n = A * r^n \quad (2)$$

Sustituyendo en (2) en (1) se tiene:

$$c_n A r^n + c_{n-1} A r^{n-1} + c_{n-2} A r^{n-2} = 0 \quad (3)$$

Si se factoriza (3) se obtiene:

$$Ar^{n-2} (c_n r^n + c_{n-1} r^{n-1} + c_{n-2} r^{n-2}) = 0 \quad (4)$$

$$c_n r^2 + c_{n-1} r + c_{n-2} = 0 \quad (5)$$

A la expresión (5) se le conoce como polinomio característico y las raíces (soluciones) de esta ecuación de recurrencia son dos raíces distintas r_1 y r_2 (reales o complejas).

Por lo tanto, la solución de la ecuación de recurrencia de segundo grado es

$$a_n = \alpha r_1^n + \beta r_2^n \quad (6)$$

Ejemplo 7

Resolver la siguiente ecuación de recurrencia con los valores iniciales dados:

$$\begin{cases} a_n + a_{n-1} - 6a_{n-2} = 0 & n \geq 2 \\ a_0 = 1, a_1 = 2 \end{cases}$$

Resolviendo el polinomio característico $x^2 + x - 6 = 0$ se tienen como resultado dos raíces distintas $x_1 = 2, x_2 = -3$.

Ejemplo 7

Se conoce que la solución general de la ecuación de recurrencia es:

$$a_n = \alpha r_1^n + \beta r_2^n$$

Sustituyendo con las raíces encontradas se tiene:

$$a_n = \alpha 2^n + \beta (-3)^n$$

Ejemplo 7

Para determinar los valores de α y β se utilizan los valores en la frontera $a_0=1$ y $a_1=2$ y con ello se tiene el sistema:

$$\begin{aligned}\alpha + \beta &= 1 \\ 2\alpha - 3\beta &= 2\end{aligned}$$

Resolviendo el sistema se tiene que $\alpha = 1$ y $\beta = 0$. Con esto se puede establecer que la solución de la ecuación de recurrencia es:

$$a_n = 2^n$$



El problema de los algoritmos iterativos o recursivos es que se necesita saber los términos anteriores para calcular el término enésimo.

El objetivo de las ecuaciones de recurrencia es obtener el resultado del término enésimo sin necesidad de calcular los anteriores.

Sucesión de Fibonacci

El enésimo término de la sucesión de Fibonacci se obtiene sumando los dos anteriores, es decir:

$$a_n = a_{n-1} + a_{n-2}, \text{ con } n > 2$$

La ecuación anterior define la sucesión de Fibonacci en función de los términos anteriores y, por tanto, recibe el nombre de relación de recurrencia, ecuación de recurrencia o ecuación en diferencias.

Por lo tanto, la sucesión de Fibonacci está definida por la siguiente relación de recurrencia:

$$\begin{aligned}f_{n+2} &= f_{n+1} + f_n, \quad \text{con } n > 1 \\f_0 &= 0, f_1 = 1\end{aligned}$$

Obtener la solución general de la ecuación de recurrencia para la serie de Fibonacci.

3.3.4 Retroceso recursivo

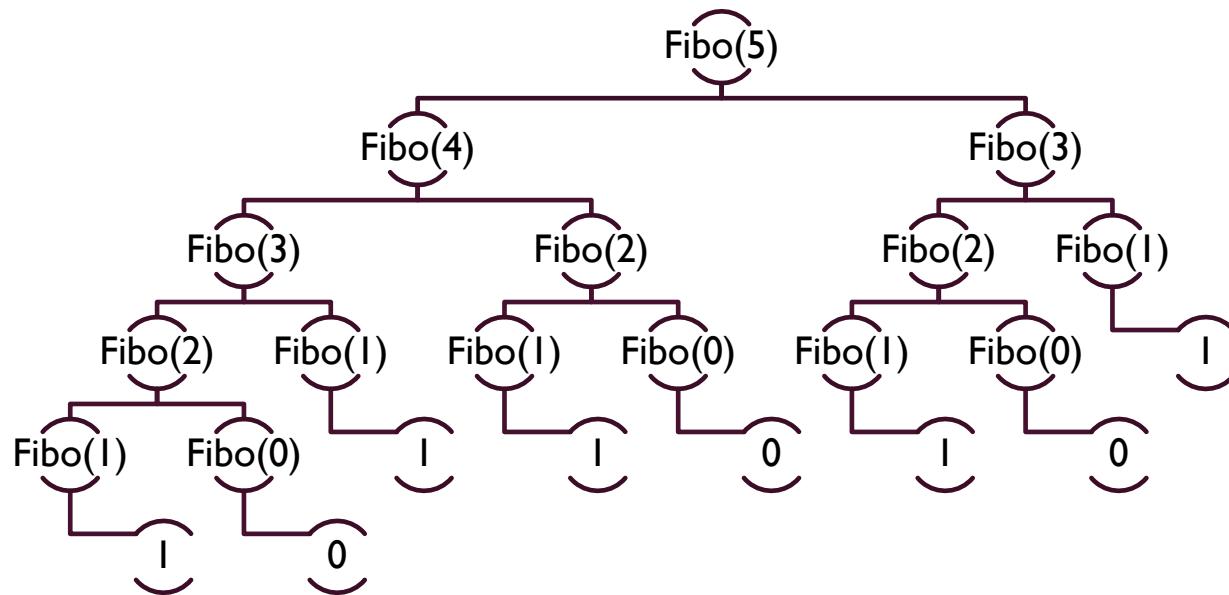
Las funciones recursivas generan llamadas recurrentes sobre una misma función, pero ¿cómo se comportan estas llamadas internamente?

El análisis del proceso recursivo es importante porque permite analizar el comportamiento en memoria del algoritmos y, con ello, la viabilidad del mismo.

Como ya se analizó, el algoritmo recursivo que permite calcular el enésimo número de la serie de Fibonacci es el siguiente:

```
1  FUNC Fibonacci (ene:ENTERO) DEV ENTERO
2      SI (ene == 0):
3          return 0
4      SI (ene == 1):
5          DEV I
6          DEV fibonacci(ene - 1) + fibonacci(ene - 2)
7  FIN_FUNC
```

A nivel de memoria, el método en la línea 6 hace dos llamadas a la función de manera recursiva, dando como resultado el siguiente comportamiento:





Por lo tanto, no es conveniente implementar algoritmos recursivos en los que distintas llamadas recursivas produzcan los mismos cálculos, lo mejor para estos casos es manejarlos como algoritmos iterativos o, si es posible, obtener el resultado a través de una ecuación de recurrencia.

3.3.5 Implementación de la recursividad

Como ya se mencionó, para poder llevar a cabo la implementación de un algoritmo recursivo es necesario considerar tres aspectos fundamentales:

- Analizar el o los caso(s) base del (de los) algoritmo(s) (¿cuándo se va a detener?)
- Analizar el caso general el cual realizará llamadas dentro de la función hacia la misma función, con instancias de entrada diferentes hasta caer en un caso base.
- No utilizar ciclos de repetición.

Ejemplo 8

El factorial de un número natural está dado por una serie de productos desde el número dado y hasta llegar a uno, disminuyendo una unidad en cada iteración, es decir:

$$n! = n * n-1 * n-2 * n-3 * \dots * 3 * 2 * 1$$

Además posee dos propiedades importantes:

$$\begin{aligned}1! &= 1: \text{el factorial de } 1 \text{ es } 1. \\0! &= 1: \text{el factorial de } 0 \text{ es } 1.\end{aligned}$$

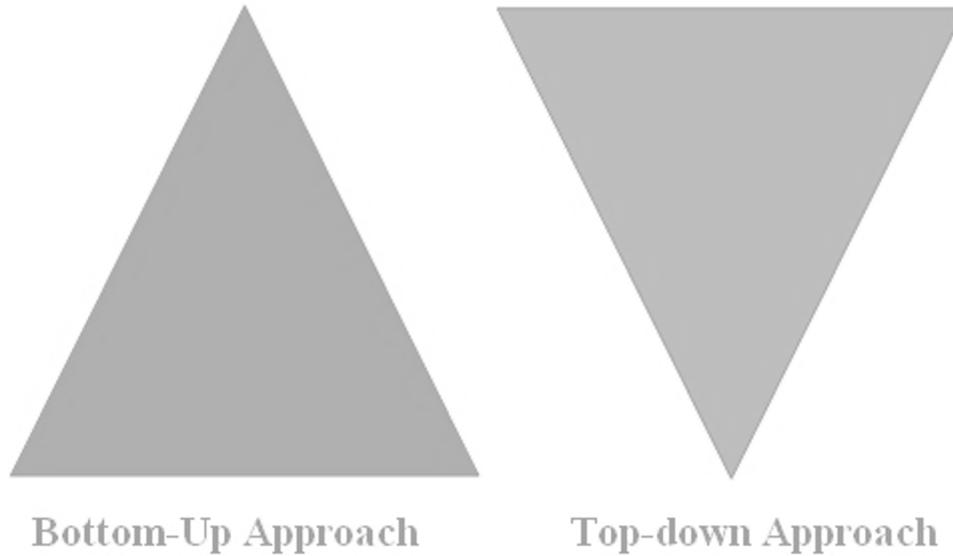
Con base en la definición anterior se puede afirmar que para implementar un algoritmo que resuelva el factorial de un número se tiene que:

- El factorial posee dos casos base, cuando el número es 1 o cuando el número es 0 el resultado es uno.
- Para el caso general, cuando el número es mayor a 1, la ecuación del factorial es:

$$n! = n * (n-1)!$$

El algoritmo recursivo para obtener el factorial de un número queda de la siguiente manera:

```
1  FUNC factorial(ene):
2      SI ene = 0 ENTONCES
3          return 1
4      SI ene == 1 ENTONCES
5          return 1
6      DEV ene * (factorial(ene - 1))
7  FIN_FUNC
```



3.4 Top-down y bottom-up

3.4 Top-down y bottom-up

Top-down es un diseño de programación descendente mediante el cual un problema se puede descomponer en varios niveles o subproblemas (pasos sucesivos de refinamiento).

El diseño descendente permite diseñar una solución del problema modularizando o segmentando de arriba (lo más general) hacia abajo (lo más específico). Esto se conoce como diseño Top-Down.

Autómatas

Un autómata es una máquina formal (mecanismo matemático) que acepta información de entrada, la procesa (sometiendo la información recibida a transformaciones simbólicas que pueden adoptar la forma de un cálculo o cómputo) y genera un resultado o salida.

Existen diversos tipos de autómatas, cada uno se asocia a una potencia computacional determinada, es decir, a una capacidad de resolución de problemas.

Los autómatas se pueden clasificar de manera algorítmica dependiendo del tipo de problema que puedan resolver. La jerarquía de autómatas es la siguiente:

- **Autómatas finitos**
- **Autómatas intermedios**
 - **Autómatas de memoria de pila**
 - **Autómatas de memoria linealmente limitada**
- **Máquinas de Turing**

Un autómata finito determinístico es una estructura del tipo:

$$A_F = \langle \Sigma, Q, q_0, T, F \rangle$$

Σ es el alfabeto de entrada

Q es el conjunto de estados

q_0 es el estado inicial

T es la función de transición de estados

F es el conjunto de los estados iniciales

Por lo tanto, cualquier máquina que permita realizar un proceso sin supervisión humana se puede considerar un autómata: una máquina de refrescos, una máquina para cobrar, un cajero automático, etc.

Se desea modelar el autómata teléfono, para lo cual se debe encontrar su estructura de la forma:

$$A_F = \langle \Sigma, Q, q_0, T, F \rangle$$

Σ : el alfabeto de entrada son los dígitos del 0 al 9.

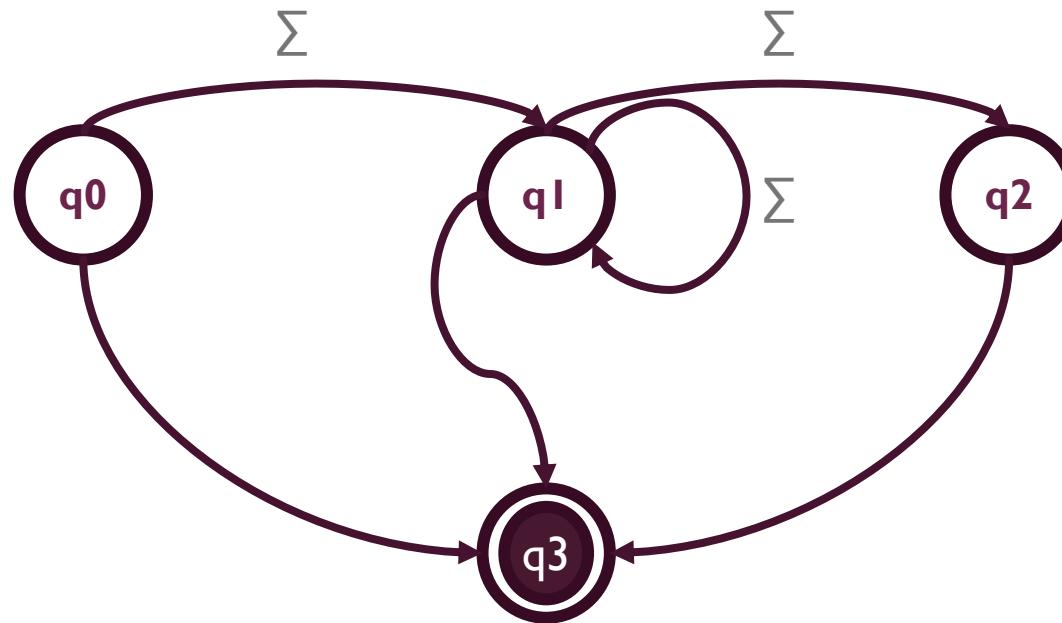
Q : el autómata consta de 4 estados: descolgar (q_0), insertar dígito (q_1), realizar llamada (q_2) y colgar (q_3).

q_0 : el estado inicial es descolgar (q_0).

T : al descolgar se puede insertar un dígito. Al insertar un dígito se pueden insertar más dígitos hasta que el número sea válido. Si el número es válido se realiza la llamada. En cualquier momento se puede colgar.

F : el único estado inicial es descolgar.

Autómata teléfono



Si la información proporcionada fuese un estado, por ejemplo q_1 , no se podría modelar con certeza el autómata, ya que el estado q_1 permite insertar un dígito de un abecedario dado, sin embargo, un cajero, una máquina de refrescos o una terminal de tarjeta bancaria (entre otros) poseen un estado similar para realizar su proceso.

Por lo tanto, el análisis de un autómata utiliza un diseño algorítmico top-down, ya que se inicia conociendo el ente a modelar (más general), para analizar todos sus subprocesos y así generar los estados del mismo (más específico).



Bottom-up es un diseño de programación ascendente que requiere identificar los procesos que se necesitan computar según vayan apareciendo, de tal manera que se pueda satisfacer el problema inmediato.

Este tipo de diseño permite generar soluciones particulares para pequeños bloques del problema. Debido a que las soluciones son particulares, se puede empezar con la etapa de pruebas de manera temprana.

Sucesión de Fibonacci

La sucesión de Fibonacci consiste en una serie de números de tal manera que, iniciando en cero y uno, el número siguiente es la suma de los números dos anteriores.

Esta construcción matemática aparece recurrentemente en la naturaleza: la distribución de las hojas alrededor del tallo, la reproducción de los conejos, la disposición de las semillas en flores y frutos, etc.

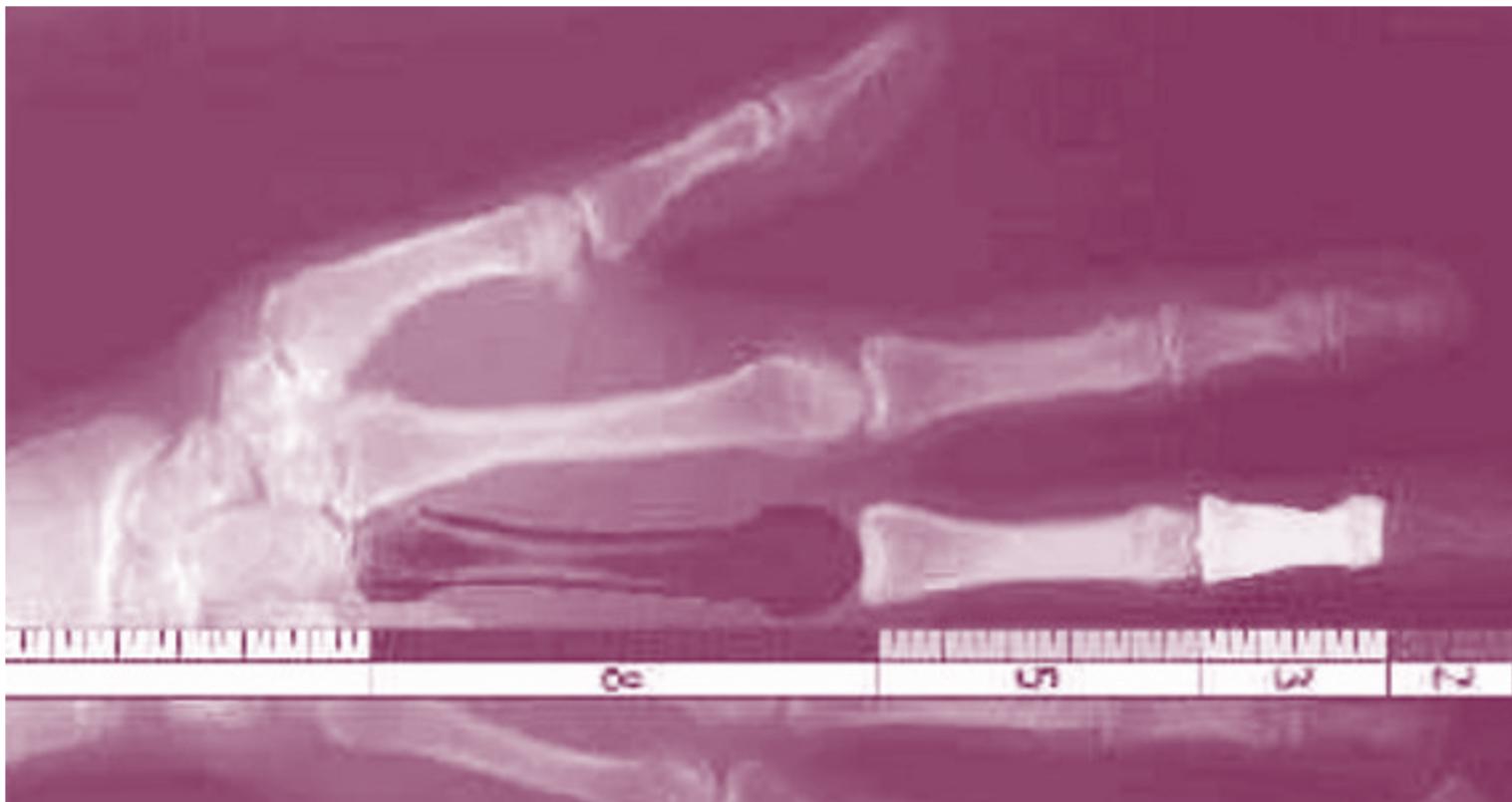
El inicio de la sucesión de Fibonacci está formada por los siguientes términos:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584...

Los números de Fibonacci tienen la característica de que el cociente de dos números consecutivos de la serie se aproxima a la denominada *razón dorada, sección áurea o divina proporción*. Este número tiene un valor de $\phi = (1+\sqrt{5})/2 = 1.61803398\dots$

Los griegos y renacentistas consideraban el número áureo como el ideal de la belleza. Un objeto que tuviese una proporción (por ejemplo, entre alto y ancho) que se ajuste a la sección áurea era estéticamente más agradable que uno que no lo hiciese.

Ejemplo 10



La sucesión de iterativa Fibonacci es un ejemplo de un diseño algorítmico bottom-up, esto debido a que para obtener el enésimo elemento de la serie es necesario calcular todos los elementos anteriores, partiendo desde el fondo (bottom) y hasta llegar al final de la solución (up).

A continuación se muestra un algoritmo que permite calcular el enésimo número de la serie de Fibonacci de manera iterativa:

```
1 FUNC fibonacci(ene: ENTERO) DEV ENTERO
2     a ← 0
3     b ← 1
4     contador ← 1
5     fibo ← 0
6     MIENTRAS contador < ene HACER
7         fibo ← b + a
8         a ← b
9         b ← fibo
10        contador ← contador + 1
11    DEV fibo
12 FIN_FUNC
```

En cada iteración del algoritmos se va generando el valor enésimo de la serie de Fibonacci. Se inicia en los primeros valores de la serie (bottom) para terminar en valor enésimo que se desea conocer (up).



3.5 Divide y vencerás

3.5 Divide y vencerás

En la práctica, muchos algoritmos poseen una estructura recursiva, es decir, para resolver un problema una función se manda llamar dentro de su cuerpo a sí misma una o más veces para resolver sub-problemas con instancias de entrada cada vez más pequeñas.



La mayoría de los algoritmos recursivos siguen el enfoque *divide y vencerás*, dividiendo el problema inicial en varios sub-problemas similares al inicial pero con instancias de entrada menores, combinando al final las soluciones para generar una solución general.



El paradigma divide y vencerás desarrolla tres pasos en cada nivel recursivo:

- **Divide el problema en un cierto número de subproblemas.**
- **Vence los subproblemas resolviéndolos de manera recursiva.**
Cuando la instancia del subproblema es lo suficientemente pequeña, simplemente los resuelve de una manera sencilla.
- **Combina la solución de los problemas hasta generar una solución general.**

Unión ordenada

El algoritmo de unión ordenada permite unir dos conjuntos de elementos previamente ordenados, siguiendo el paradigma divide y vencerás:

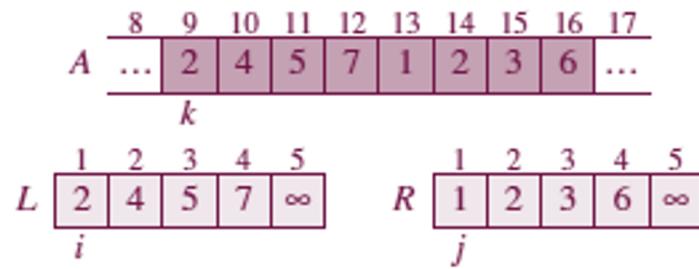
- **Divide la secuencia de n elementos a ser ordenados en dos subsecuencias de $n/2$ elementos cada una.**
- **Vence el reto ordenando recursivamente las dos subsecuencias de elementos utilizando unión ordenada.**
- **Combina (une) las dos subsecuencias ordenadas para producir la salida ordenada.**

La operación clave en el algoritmo consiste en unir las dos secuencias ordenadas en un paso combinado.

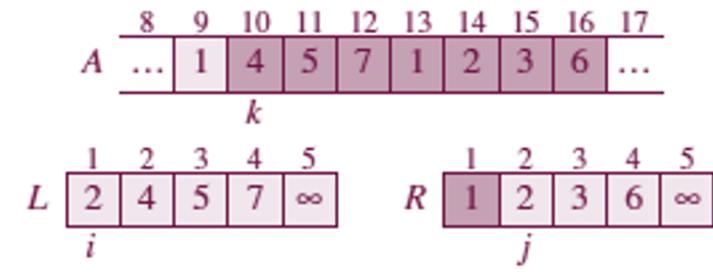
Para realizar dicha unión se utiliza una función auxiliar $\text{unir}(A, p, q, r)$, donde A es un arreglo, p, q y r son índices del arreglo tales que $p \leq q < r$. La función asume que los subarreglos $A[p, q]$ y $A[q+1, r]$ están previamente ordenados. El proceso une los dos arreglos para formar un subarreglo ordenado que reemplaza a $A[p, r]$.

```
1  FUNC unir(A[]:ENT, p:ENT, q:ENT, r:ENT)
2      n1 ← q - p + 1
3      n2 ← r - q
4      create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
5      MIENTRAS i ← 1 <= n1
6          L[i] ← A[p + i - 1]
7      MIENTRAS j ← 1 <= n2
8          R[j] ← A[q + j]
9      L[n1 + 1] ← ∞
10     R[n2 + 1] ← ∞
11     i ← 1
12     j ← 1
13     MIENTRAS k ← p <= r
14         SI L[i] ≤ R[j]
15             A[k] ← L[i]
16             i ← i + 1
17         DE LO CONTRARIO
18             A[k] ← R[j]
19             j ← j + 1
20 FIN_FUNC
```

A continuación se muestran posibles transiciones para un arreglo dado:



(a)



(b)

Ejemplo 11

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	5	7	1	2	3	6	...	
				k							

L	1	2	3	4	5	2	4	5	7	∞	
						i					

R	1	2	3	4	5	1	2	3	6	∞	
						j					

(c)

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	2	7	1	2	3	6	...	
				k							

L	1	2	3	4	5	2	4	5	7	∞	
						i					

R	1	2	3	4	5	1	2	3	6	∞	
						j					

(d)

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	2	3	1	2	3	6	...	
				k							

L	1	2	3	4	5	2	4	5	7	∞	
						i					

R	1	2	3	4	5	1	2	3	6	∞	
						j					

(e)

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	2	3	4	2	3	6	...	
				k							

L	1	2	3	4	5	2	4	5	7	∞	
						i					

R	1	2	3	4	5	1	2	3	6	∞	
						j					

(f)

Ejemplo 11

	8	9	10	11	12	13	14	15	16	17	
<i>A</i>	...	1	2	2	3	4	5	3	6	...	
					<i>k</i>						

<i>L</i>	1	2	3	4	5					
	2	4	5	7	∞					
				<i>i</i>						

<i>R</i>	1	2	3	4	5					
	1	2	3	6	∞					
			<i>j</i>							

(g)

<i>A</i>	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	3	4	5	6	6	...	
					<i>k</i>						

<i>L</i>	1	2	3	4	5					
	2	4	5	7	∞					
				<i>i</i>						

<i>R</i>	1	2	3	4	5					
	1	2	3	6	∞					
			<i>j</i>							

(h)

<i>A</i>	8	9	10	11	12	13	14	15	16	17	
	...	1	2	2	3	4	5	6	7	...	
					<i>k</i>						

<i>L</i>	1	2	3	4	5					
	2	4	5	7	∞					
				<i>i</i>						

<i>R</i>	1	2	3	4	5					
	1	2	3	6	∞					
			<i>j</i>							

(i)

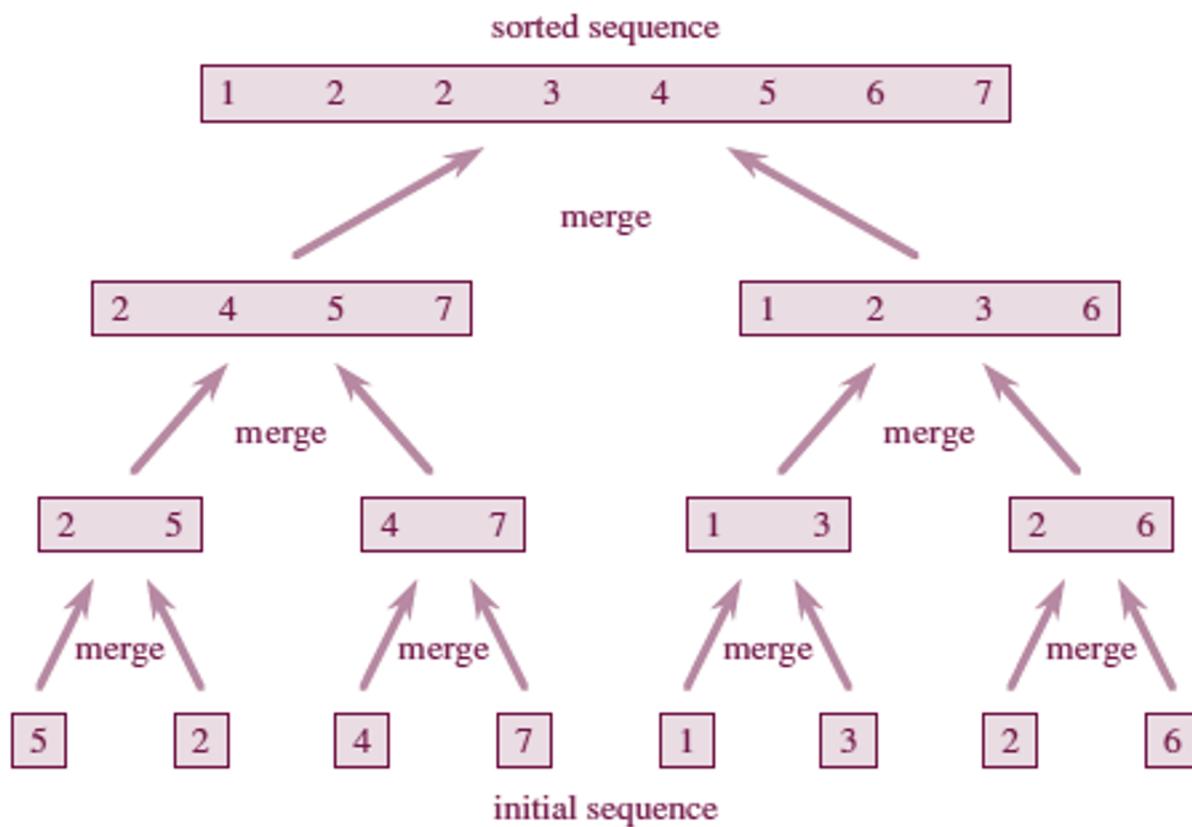
Regresando al algoritmo de `unionOrdenada(A, p, r)`, éste recibe como parámetros un arreglo de elementos, un índice inferior y un índice superior, y acomoda los elementos en un subarreglo `A[p, r]`.

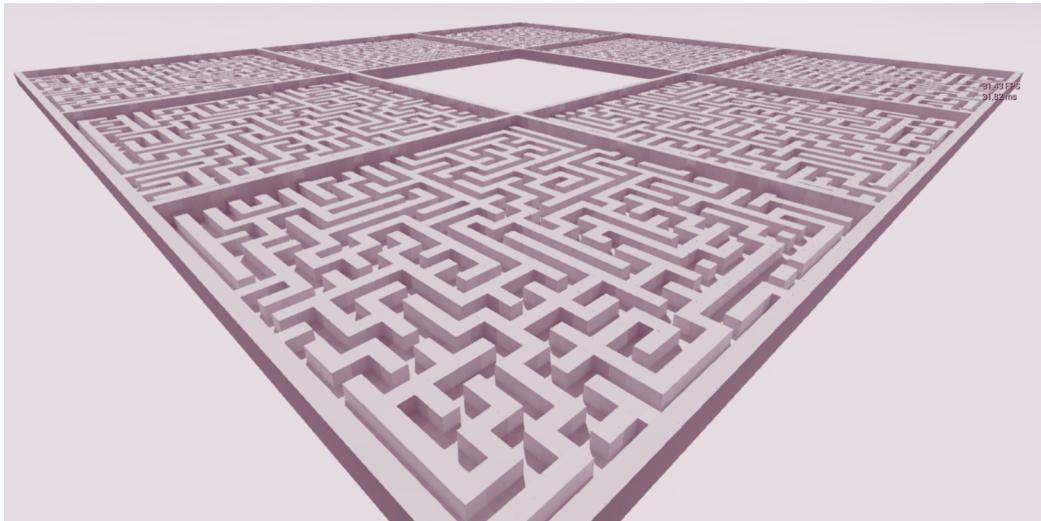
Si $p \geq r$, el arreglo tendrá como máximo un elemento y, por tanto, ya se encuentra ordenado. De lo contrario, se divide el arreglo $A[p, r]$ en $A[p, q]$ y $A[q+1, r]$ que contienen, respectivamente, la mitad de los elementos del arreglo ($n/2$).

El algoritmo que permite realizar la unión ordenada de elementos es el siguiente:

```
1 FUNC unionOrdenada (A, p, r)
2     SI p < r ENTONCES
3         q ← (p + r)/2
4         unionOrdenada (A, p, q)
5         unionOrdenada (A, q + 1, r)
6         unir (A, p, q, r)
7     FIN_SI
8 FIN_FUNC
```

Ejemplo 12





3.6 Backtrack

3.6 Backtrack

El **backtracking** o búsqueda hacia atrás es una técnica de programación para hacer búsquedas sistemáticas a través de todas las configuraciones posibles dentro de un espacio de búsqueda.

Las configuraciones pueden representar todos los posibles arreglos de objetos (permutaciones) o todos las posibles maneras para crear una colección de objetos (subconjuntos).

Robot de laberinto

Al tratar de resolver un laberinto a través de un robot seguidor de línea, se debe almacenar el conocimiento adquirido y analizar los resultados para aprender la ruta más corta (búsqueda hacia atrás).

Suponiendo que se tiene un robot seguidor de línea que utiliza la lógica izquierda, esto es, en cualquier intersección que el robot encuentre va a tomar el rumbo, preferentemente, hacia la izquierda, se tienen diversas posibilidades, todas las cuales se deben analizar.

Ejemplo 13



B



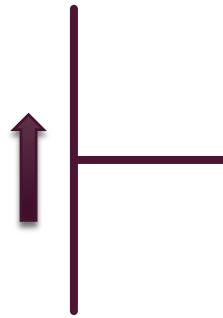
D



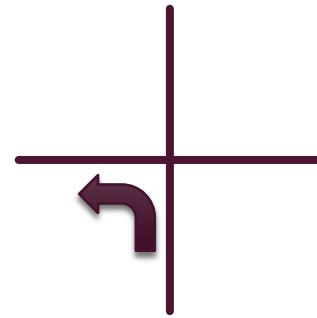
I



T



L

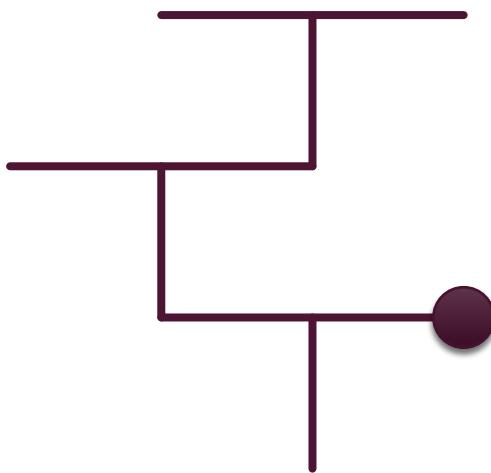


X



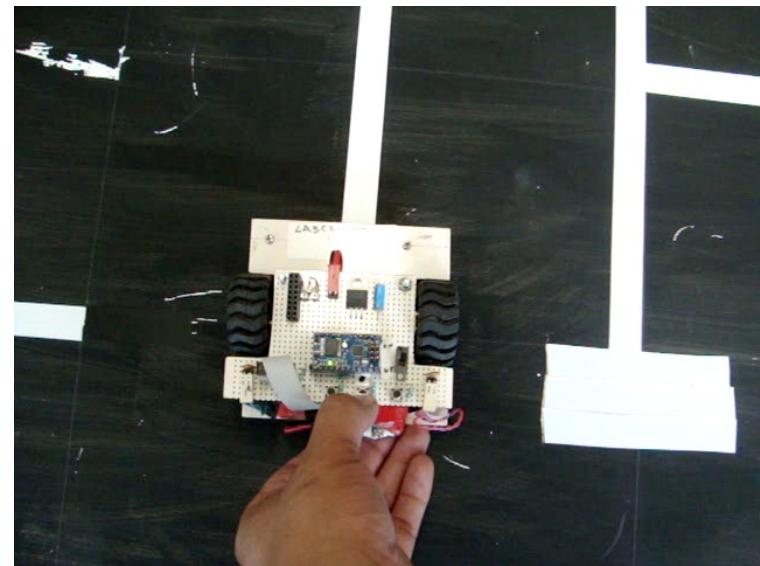
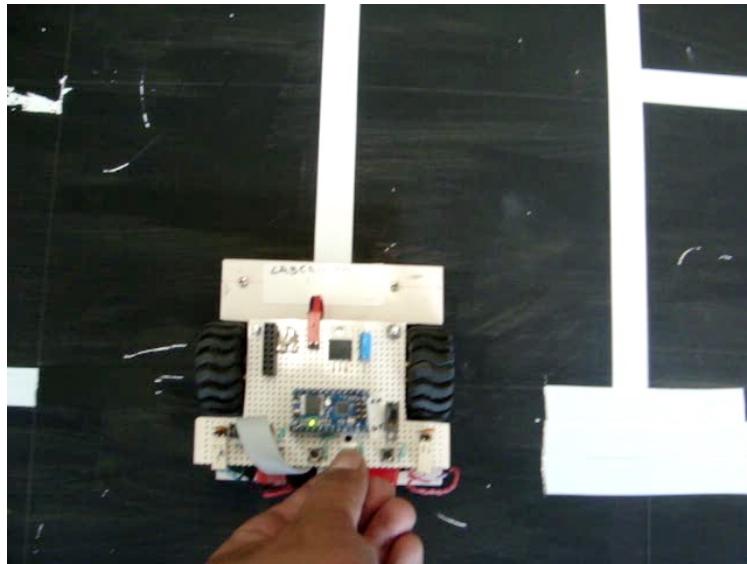
7

Ejemplo 13



T
T D
T D T
T D T B
T D T B L
T D D
T D D I
T D D I T
T D D I T B
T D D I T B L
T D D I D
T D D I D B
T D D I D B 7
T D D I B
T D D I B D
T D D B
T D D B 7
T D B
T D B I
T B L
D

Ejemplo 13



3 Estrategia para construir algoritmos

Objetivo: Aplicar diversas técnicas como la recursividad para construir algoritmos.

3.1 Algoritmos de búsqueda exhaustiva y fuerza bruta.

3.2 Algoritmos ávidos

3.3 Recursividad.

3.3.1 El concepto de recursividad.

3.3.2 Funciones matemáticas de recursividad.

3.3.3 Uso de relaciones de recurrencia para analizar algoritmos recursivos.

3.3.4 Retroceso recursivo.

3.3.5 Implementación de la recursividad.

3.4 Top-down y bottom-up.

3.5 Divide y vencerás.

3.6 Backtrack.