

# 1. Algoritmos de fuerza bruta

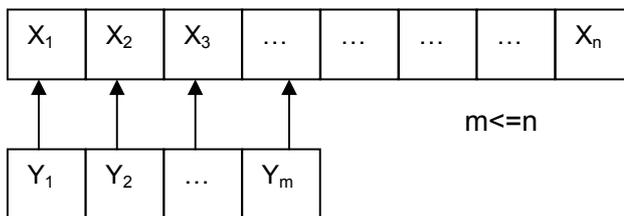
Los algoritmos de fuerza bruta se caracterizan por ser la aproximación más directa para resolver un problema. De forma general, esta técnica se basa en una interpretación del planteamiento del problema y de sus condiciones iniciales, y utiliza una estrategia de buscar la solución de forma sistemática en la que explora todas las posibilidades. En algunos casos la solución obtenida por fuerza bruta puede ser mejorada con relativa facilidad para obtener aproximaciones más elegantes en cuanto a la codificación y de menor costo en cuanto a la complejidad de los algoritmos.

A continuación se presentan algunos ejemplos en los cuales se emplean algoritmos de fuerza bruta para solucionar un problema.

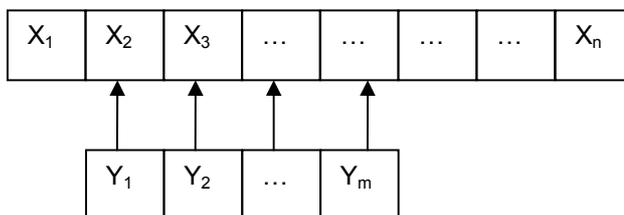
## 1.1. El problema de la comparación de cadenas

*Dadas dos cadenas de caracteres X e Y, determinar si Y está contenida en X.*

Una aproximación por fuerza bruta consiste en alinear las dos cadenas de caracteres de izquierda a derecha, y verificar si los caracteres coinciden. En caso de que los caracteres no coincidan, se avanza un carácter en la cadena X y se alinea nuevamente con la cadena Y para realizar la siguiente comparación. Gráficamente, esta estrategia puede ser representada de la siguiente forma:



1. Si todos los caracteres son iguales, Y está contenida en X
2. Si los caracteres no coinciden, avanzar un carácter en X y realizar de nuevo la comparación



El algoritmo basado en esta estrategia se muestra a continuación:

```

//Algoritmo para resolver el problema de búsqueda de cadenas
//Entrada:
//  X: Cadena de texto
//  Y: Cadena a buscar dentro de la cadena X
//  n: Longitud de la cadena X
//  m: Longitud de la cadena Y.
//Valor de retorno:
//  posición en la cual se encuentra Y dentro de X
//  -1 si Y no se encuentra dentro de X.

algoritmo subcadenaFB(X,Y,n,m)
  para i=1 hasta n - m
    j=0
    mientras j <= m and X[i+j] = Y[j]
      j=j+1
    fin_mientras
    si j = m retornar i
  fin_para
  retornar -1
fin algoritmo

```

En esta estrategia se recorre la cadena X de izquierda a derecha (ciclo externo), y a medida que se recorre se comparan los caracteres de la cadena X y de la cadena Y en forma secuencial (ciclo interno). Si el número de caracteres iguales (j) es igual a la longitud de la cadena Y (m), se retorna la posición a partir de la cual se encuentra Y dentro de X (i). Si la cadena Y no se encuentra dentro de la cadena X, se retorna -1.

## 1.2. El problema del laberinto

*Dado un laberinto, la posición de inicio y la posición de llegada, determinar si es posible llegar desde el inicio hasta la llegada.*

En una aproximación por fuerza bruta, a partir de la posición actual se deberán explorar todas las posibilidades de realizar un movimiento hacia la posición de llegada (todas las direcciones). Para mejorar un poco el algoritmo, se puede marcar cada casilla una vez se haya visitado (Esto evita además que el algoritmo pase dos o más veces por la misma casilla). Un algoritmo que resuelve el problema del laberinto es el siguiente:

```

//Algoritmo recursivo para resolver el problema del laberinto
//Entrada:
//  lab: Matriz del laberinto.lab[i][j] = 1 hay camino, 0 no hay camino
//  vis: Matriz para marcar las casillas visitadas
//  n, m: Orden de la matriz n = filas, m = columnas
//  fi, ci, ff, cf: Coordenadas inicial y final (fila y columna)
//Valor de retorno:
//  1: existe camino
//  0: no existe camino

algoritmo laberinto(lab[ ][ ], vis[ ][ ], n, m, fi, ci, ff, cf)

  si fi = ff and ci == cf //Si estamos en la llegada
    retornar 1;
  fin_si

  //Marcar la casilla como visitada, para evitar
  // regresar por el mismo camino
  vis[fi][ci] = 1;

```

```

dir = 1;
mientras dir <=4
    //funciones auxiliares que retornan la siguiente fila y columna
    sf = siguiente_fila(fi, dir);
    sc = siguiente_columna(ci, dir);
    // Primero verificar si [sf,sc] es un movimiento valido
    si movimiento_valido(lab, vis, n, m, sf, sc)
        //Llamada recursiva a laberinto, con la
        //nueva posicion valida
        si laberinto(lab, vis, n, m, sf, sc, ff, cf)
            retornar 1;
        fin_si
    fin_si
    dir = dir + 1;
fin_mientras
retornar 0;
fin_algoritmo

```

Este algoritmo hace uso de las funciones auxiliares `siguiente_fila`, `siguiente_columna` y `movimiento_valido`. Estas funciones calculan la siguiente posición a verificar. A continuación se presenta el pseudocódigo de las funciones mencionadas.

```

funcion siguiente_fila(fila, direccion)
    si direccion = 1 //arriba
        retornar fila - 1
    si direccion = 2 or direccion = 4 //derecha e izquierda
        retornar fila
    si direccion = 3 //abajo
        retornar fila + 1
fin_funcion

funcion siguiente_columna(columna, direccion)
    si direccion = 1 or direccion = 3 //arriba y abajo
        retornar columna
    si direccion = 2 //derecha
        retornar columna + 1
    si direccion = 4 //izquierda
        retornar columna - 1
fin_funcion

funcion movimiento_valido(lab[[[], vis[[[], n, m, i, j)
    //Retornar verdadero si la casilla [i,j] esta en rango
    // y no ha sido visitada
    si i < n and j < m and i >=0 and j >= 0
        and vis[i][j]=0 and lab[i][j]=1
        retornar 1
    sino
        retornar 0
    fin_si
fin_funcion

```

### 1.3. El problema del cambio de monedas

Dada una cantidad de dinero  $M$  y un conjunto  $d$  de denominaciones de billetes, desarrollar el algoritmo que calcule el mínimo número de billetes requeridos para completar la cantidad  $M$ . Suponga que existe un número infinito de billetes de cada denominación y que existe cambio exacto para  $M$ .

Este problema presenta un nivel de complejidad mayor, ya que un algoritmo de fuerza bruta deberá explorar todas las posibles combinaciones que suman  $M$ . Por ejemplo, si  $M=90$  y  $d = \{50, 100, 10, 20\}$ , se puede determinar que el mínimo de billetes es 3. Sin embargo, se deben explorar todas las combinaciones:

$$90 = 50 + 20 + 20 \text{ (3 billetes)}$$

$$90 = 50 + 20 + 10 + 10 \text{ (4 billetes)}$$

$$90 = 50 + 10 + 10 + 10 + 10 \text{ (5 billetes)}$$

$$90 = 20 + 20 + 20 + 20 + 10 \text{ (5 billetes)}$$

.. Y así sucesivamente.

Un algoritmo para resolver el problema del cambio de monedas de forma recursiva es el siguiente:

```
//Algoritmo recursivo para resolver el problema del cambio
//Entrada:
//   m: cantidad de dinero
//   d: arreglo de denominaciones
//   n: numero de denominaciones
//   k: subíndice de la denominación actual (para la recursión).
//     En la llamada inicial toma el valor de 1.
//   total: número de billetes entregados. En la llamada inicial
//     toma el valor de 0.
//Valor de retorno:
//   número mínimo de billetes, o cero (0) si no existe una
//   combinación válida de billetes.

algoritmo cambio(m, d[], n, k, total)

    minimo = 0 //Se asume que no hay solucion

    si m = 0 //Caso base
        minimo = total
    sino
        si k <= n //Solo continuar si no se han agotado las
            // denominaciones
            si m >= d[k] //Se debe restar el valor de la denominacion
                //actual y continuar probando
                min = INFINITO; //Se asume que no hay solucion
                para i=k hasta n
                    total = cambio(m-d[i],d,n,i, total + 1)
                    //Se encontro una alternativa mejor

                    si total > 0 and total < min
                        min = result
                fin_si
            fin_para
        si min != INFINITO //Se encontro una solucion!
```

```
        minimo = min
        fin_si
    sino //Pasar a la siguiente denominacion
        total = cambio(m,d,n,k+1, total)
        si total > 0
            minimo = total
        fin_si
    fin_si
fin_si
retornar minimo
fin_algoritmo
```