

# Algoritmos ávidos

Pedro O. Pérez M., MTI

Análisis y diseño de algoritmos  
Tecnológico de Monterrey

*pperezm@tec.mx*

02-2019

# Contenido

## Introducción

- Definición

- Forma general

## Ejemplos clásicos

- Cambio de monedas

- Búsqueda en profundidad sobre un grafo

- Programación de actividades

- Reservaciones de hotel

- Fracciones egipcias

- Subconjunto de producto máximo de un arreglo

- Subsecuencia lexicográficamente más grande

# Definición

Se conocen también como *algoritmos miopes, golosos, ávidos o avaros*, y caracterizan por decisiones basados en la información que tienen a primera mano, sin tener en cuenta lo que pueda pasar más adelante. Además, una vez que toman una decisión nunca reconsideran otras posibilidades, lo que ocasionalmente los lleva a caer en puntos muertos o sin salida.

Los algoritmos ávidos también se caracterizan por la rapidez con la que encuentran una solución (cuando la encuentran), que casi nunca es la mejor. Normalmente son utilizados para resolver problemas en los cuales la velocidad de respuesta debe ser muy alta o el espacio de búsqueda es muy grande.

Ejemplos típicos de problemas que se pueden resolver mediante este paradigma están las búsquedas en árboles o graos, solución de laberintos y algunos juegos entre otros. También muchos problemas que requieren obtener máximos o mínimos.

## Forma general

La estrategia general de este tipo de algoritmos se basa en la construcción de una solución que comienza sin elementos, y cada vez que debe tomar algún tipo de decisión lo hace con la información que tiene en ese momento, para, de alguna manera, agregar elementos y así avanzar hacia la solución final. Cada elemento se agrega al conjunto solución, y así hasta llegar a la solución completa o a un punto en el cual el algoritmo no puede seguir avanzando, lo cual no indica que no se encontró una solución al problema.

---

## Procedure 1 GREEDY\_ALGORITHM

---

**Input:**  $C : \text{Set}$

$S : \text{Set}$

**while**  $C \neq \emptyset$  **and**  $SOLUTION(S) = \text{false}$  **do**

$x \leftarrow SELECT(C)$

$S \leftarrow S + x$

$C \leftarrow C - x$

**end while**

**if**  $SOLUTION(S)$  **then**

**return**  $S$

**else**

**return**  $\emptyset$

**end if**

---

## Cambio de monedas

Dado un sistema monetario  $S$  con  $N$  monedas de diferentes denominaciones y una cantidad de cambio  $C$ , calcular el menor número de monedas del sistema monetario  $S$  equivalente a  $C$ .

Ejemplos:

► **Input** :  $s[] = 1, 3, 4$   $c = 6$

**Output** : 3

**Explanation** : The change will be  $(4 + 1 + 1) = 6$





# Búsqueda en profundidad

La búsqueda en profundidad (en inglés **DFS** o **Depth First Search**) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado <sup>1</sup>.

---

<sup>1</sup><https://goo.gl/ZsAAzP>



# Programación de actividades

Te dan  $N$  actividades con sus tiempos de inicio ( $S_i$ ) y finalización ( $F_i$ ).  
Selecciona el número máximo de actividades que puede realizar una sola persona, asumiendo que una persona solo puede trabajar en una sola actividad a la vez. <sup>2</sup>  
Ejemplo:

► **Input :**

$\text{start}[] = (1, 3, 0, 5, 8, 5)$

$\text{finish}[] = (2, 4, 6, 7, 9, 9)$

**Output : 4**

---

<sup>2</sup><https://goo.gl/1RG25M>

---

## Procedure 4 ACTIVITIES\_SELECTION

---

**Input:**  $A$  : *Activity\_Array*  
     $SORT\_ASC\_BY\_END(A)$   
     $i \leftarrow 1$   
     $S \leftarrow \emptyset + A[i]$   
    **for**  $j \leftarrow 2$  **to**  $A.length$  **do**  
        **if**  $A[j].start \geq A[i].end$  **then**  
             $S \leftarrow S + A[j]$   
             $i \leftarrow j$   
        **end if**  
    **end for**  
    **return**  $S$

---

# Reservaciones de hotel

Un gerente de hotel debe procesar  $N$  reservas anticipadas de habitaciones para la próxima temporada. Su hotel tiene  $K$  habitaciones. Las reservas contienen una fecha de llegada y una fecha de salida. Quiere saber si hay suficientes habitaciones en el hotel para satisfacer la demanda. <sup>3</sup>

---

<sup>3</sup><https://goo.gl/7e6idL>

**Input:** *Arrival* : Array, *Departure* : Array, *n* : Integer, *k* : Integer

*SORT ASC( Departure)*

$$i \leftarrow 1$$
$$j \leftarrow 1$$

```
current ← 0
```

$$required \leftarrow 0$$
**while**  $i < n$  **and**  $j < n$  **do**

**if**  $Arrival[i] < Departure[j]$  **then**

$$current \leftarrow current + 1$$

```
required = MAX(current, required)
```

$$i \leftarrow i + 1$$

else

$$current \leftarrow current - 1$$
$$j \leftarrow j + 1$$

---

```
while  $i < n$  do
     $current \leftarrow current + 1$ 
     $required = MAX(current, required)$ 
     $i \leftarrow i + 1$ 
end while
while  $j < n$  do
     $current \leftarrow current - 1$ 
     $j \leftarrow j + 1$ 
end while
return  $k \geq required$ 
```

---



# Fracciones egipcias

Cada fracción positiva puede representarse como la suma de fracciones unitarias únicas. Una fracción es una fracción unitaria si el numerador es 1 y el denominador es un entero positivo, por ejemplo,  $1/3$  es una fracción unitaria. Dicha representación se llama fracción egipcia, ya que fue utilizada por los antiguos egipcios.<sup>4</sup>

---

<sup>4</sup><https://goo.gl/sUzBPd>

---

## Procedure 6 EGYPTIAN

---

**Input:** *num* : integer, *dem* : Integer

```
if num = 0 or dem = 0 then
    return
end if
if dem mód num = 0 then
    print "1/" + (dem/num)
    return
end if
if num mód dem = 0 then
    print (num/dem)
    return
end if
```

---

```
if  $num > dem$  then
    print  $(num/dem) + " + "$ 
    EGYPTIAN( $num \bmod dem, dem$ )
    return
end if
 $n \leftarrow (dem/num) + 1$ 
print "1/" +  $n$  + " + "
EGYPTIAN( $(num * n) - dem, dem * n$ )
```

---

# Subconjunto de producto máximo de un arreglo

Dado un arreglo  $A$ , tenemos que encontrar el producto máximo posible con el subconjunto de elementos presentes en el arreglo. El producto máximo puede ser solo uno de los elementos del arreglo.<sup>5</sup>

Ejemplos:

► **Input** :  $a[] = -1, -1, -2, 4, 3$

**Output** : 24

**Explanation** : Maximum product will be  $(-2 * -1 * 4 * 3) = 24$

► **Input** :  $a[] = -1, 0$

**Output** : 0

**Explanation** : 0 (single element) is maximum product possible

► **Input** :  $a[] = 0, 0, 0$

**Output** : 0

---

<sup>5</sup><https://goo.gl/spb5Ka>

Una solución simple sería generar todos los subconjuntos, encontrar el producto de cada subconjunto y regresa el máximo. Sin embargo, existe una mejor solución si tomamos en cuenta los siguiente factores:

- ▶ Si el número de elementos negativos es par, el resultado es, sencillamente, el producto de todos los elementos.
- ▶ Si el número de elementos negativos es impar, el resultado es la multiplicación de todos los elementos excepto el número negativo más grande.
- ▶ Si hay ceros, el resultado el producto de todos los números, excepto los ceros con una excepción. La excepción es cuadn hay un número negativo y todos los otros números son ceros. En este caso, el resultado es 0.

---

## Procedure 7 MAXIMUM\_PRODUCT

---

Input:  $A$  : Array

if  $n = 1$  then

return  $A[1]$

end if

$max\_neg \leftarrow INT\_MIN$

$count\_neg \leftarrow 0$

$count\_zero \leftarrow 0$

$product \leftarrow 1$

---

---

---

```
for  $i \leftarrow 1$  to  $A.length$  do
  if  $A[i] = 0$  then
     $count\_zero \leftarrow count\_zero + 1$ 
  else
    if  $A[i] < 0$  then
       $count\_neg \leftarrow count\_neg + 1$ 
       $max\_neg \leftarrow MAX(max\_neg, count\_neg)$ 
    end if
     $product \leftarrow product * A[i]$ 
  end if
end for
```

---

---

```
if count_zero == n then
    return 0
end if
if count_neg mód 2 = 1 then
    if count_neg = 1 and count_zero > 0 and (count_neg + count_zero) =
        n then
        return 0
    end if
    product ← product max_neg
end if
return product
```

---



# Subsecuencia lexicográficamente más grande

Dada una cadena  $S$  y un entero  $K$ . La tarea es encontrar la subsecuencia lexicográficamente más grande de  $S$ , digamos  $T$ , de modo que cada carácter en  $T$  debe aparecer al menos  $K$  veces. <sup>6</sup>

Ejemplos:

► **Input** :  $S = \text{"zzwzawa"} K = 2$   
**Output** : zzzaa

---

<sup>6</sup><https://goo.gl/iwFFCA>

---

## Procedure 8 SUBSEQUENCE

---

**Input:**  $S : \text{String}, T : \text{String}, n : \text{Integer}, k : \text{Integer}$

$last \leftarrow 0$

$new\_last \leftarrow 0$

$size \leftarrow 0$

**for**  $ch \leftarrow 'z'$  **to**  $'a'$  **do**

$count \leftarrow 0$

**for**  $i \leftarrow last$  **to**  $n$  **do**

**if**  $S[i] = ch$  **then**

$count \leftarrow count + 1$

**end if**

**end for**

**if**  $count \geq k$  **then**

        NEXT SLIDE

**end if**

**end for**

---

```
for  $i \leftarrow last$  to  $n$  do
  if  $S[i] = ch$  then
     $T[size] \leftarrow ch$ 
     $size \leftarrow size + 1$ 
     $new\_last \leftarrow i$ 
  end if
end for
 $last \leftarrow new\_last$ 
```

---