Semana 2

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

08-2021

Introducción

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 grafos, 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: $C : Set$

```
S : Set
while C ≠ ∅ and SOLUTION(S) = false do
    x ← SELECT(C)
    S ← S + x
    C ← C − x
end while
if SOLUTION(S) then
    return S
else
    return ∅
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) = 3

Procedure 2 COIN_CHANGE

Input: $S : Array, c : Integer$

$min \leftarrow 0$

$SORT\_DESC(S)$

for $i \leftarrow 1$ **to** $S.length$ **do**

$\quad min \leftarrow min + (c/S[i])$

$\quad c \leftarrow c \mod S[i]$

end for

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 [1].

[1] `https://goo.gl/ZsAAzP`

Procedure 3 DFS

Input: $start : Vertex, g : Graph$

```
visited : Set
x_visit : Stack
x_visit.push(start)
while !x_visit.empty() do
    current ← x_visit.pop()
    if current ∉ visited then
        visited.add(current)
        for all v in current.connections() do
            x_visit.push(start)
        end for
    end if
end while
return visited
```

Word Transformation [2]

Un rompecabezas común que se encuentra en muchos periódicos y revistas es la transformación de palabras. Al tomar una palabra de inicio y alterar sucesivamente una sola letra para formar una nueva palabra, se puede construir una secuencia de palabras que cambia la palabra original a una palabra final dada. Por ejemplo, la palabra "spice"se puede transformar en cuatro pasos a la palabra "stock"de acuerdo con la siguiente secuencia: spice, slice, slick, stick, stock. Cada palabra sucesiva difiere de la palabra anterior en una sola posición de carácter, mientras que la longitud de la palabra sigue siendo la misma. Dado un diccionario de palabras de las cuales hacer transformaciones, más una lista de palabras iniciales y finales, escribe un programa para determinar el número de pasos en la transformación más corta posible.

[2] `https://onlinejudge.org/external/4/429.pdf`

Entrada

La entrada tendrá dos secciones. La primera sección será el diccionario de palabras disponibles con una palabra por línea, terminada por una línea que contiene un asterisco (*) en lugar de una palabra. Puede haber hasta 200 palabras en el diccionario; todas las palabras serán alfabéticas y en minúsculas, y ninguna palabra tendrá más de diez caracteres. Las palabras pueden aparecer en el diccionario en cualquier orden. Después del diccionario hay pares de palabras, un par por línea, con las palabras en el par separadas por un solo espacio. Estos pares representan las palabras iniciales y finales en una transformación. Se garantiza que todas las parejas tendrán una transformación utilizando el diccionario dado. Las palabras iniciales y finales aparecerán en el diccionario. Dos conjuntos de entrada consecutivos se separarán por una línea en blanco.

Salida

La salida debe contener una línea por par de palabras para cada conjunto de prueba, y debe incluir la palabra inicial, la palabra final y el número de pasos en la transformación más corta posible, separados por espacios individuales.

Ejemplo de entrada
dip
lip
mad
map
maple
may
pad
pip
pod
pop
sap
sip
slice
slick
spice
stick
stock
*
spice stock
may pod

Ejemplo de salida
spice stock 4
may pod 3

Subgrafos máximos

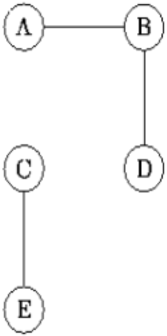
Considere un grafo G formado a partir de un gran número de vértices conectados por arcos. G se dice que está conectado si existe un camino entre cualquier par de vértices en G. Por ejemplo, el siguiente grafo no está conectado, porque no hay trayectoria de A a C.

Este grafo contiene, sin embargo, un número de subgrafos que están conectados, uno para cada uno de los siguientes conjuntos de vértices: (A), (B), (C), (D), (E), (A, B), (B,D), (C, E), (A, B, D).

Un subgrafo conectado es máximo si no hay vértices y arcos en el grafo original que podrían añadirse al subgrafo y todavía dejarlo conectado. En la imagen anterior, hay dos subgrafos máximos, uno asociada con los vértices (A, B, D) y el otro con los vértices (C, E).

Desarrollar un algoritmo para determinar el número de subgrafos máximos conectados de un gráfico dado.

http://bit.do/eNTvC



Procedure 4 COUNTING_GRAPHS

```
Input: G : Graph
  Reached : Set
  acum ← 0
  Mark all the vertexes in G as No_Explored
  for vertex in G do
    if vertex is not marked Explored then
      DFS(vertex, G, Reached)
      acum ← acum + 1
    end if
  end for
  return acum
```

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. [3]

Ejemplo:

- **Input** :
  start[] = (1, 3, 0, 5, 8, 5)
  finish[] = (2, 4, 6, 7, 9, 9)
  Output : 4

[3] https://goo.gl/1RG25M

Procedure 5 ACTIVITIES_SELECTION

Input: $A : Activity\_Array$

```
SORT_ASC_BY_END(A)
i ← 1
S ← ∅ + A[i]
for j ← 2 to A.length do
    if A[j].start ≥ A[i].end then
        S ← S + A[j]
        i ← 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. [4]

[4] https://goo.gl/7e6idL

Procedure 6 BOOKING_PROBLEM

Input: $Arrival : Array, Departure : Array, k : Integer$

$SORT\_ASC(Arrival)$

$SORT\_ASC(Departure)$

$i \leftarrow 1$

$j \leftarrow 1$

$current \leftarrow 0$

$required \leftarrow 0$

```
while i < Arrival.length and j < Departure.length do
   if Arrival[i] < Departure[j] then
      current ← current + 1
      required = MAX(current, required)
      i ← i + 1
   else
      current ← current − 1
      j ← j + 1
   end if
end while
```

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 $k \geq required$

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. [5]

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

[5] 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 es el producto de todos los números, excepto los ceros con una excepción. La excepción es cuando 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
  if A[1] < 1 then
    return 0
  else
    return A[1]
  end if
end if
max_neg ← INT_MIN
count_neg ← 0
count_zero ← 0
product ← 1
```

```
for i ← 1 to A.length do
    if A[i] = 0 then
        count_zero ← count_zero + 1
    else
        if A[i] < 0 then
            count_neg ← count_neg + 1
            max_neg ← MAX(max_neg, count_neg)
        end if
        product ← product * A[i]
    end if
end for
```

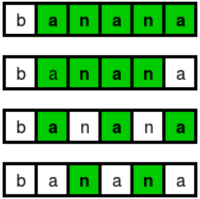
if $count\_zero = n$ **then**
  return 0
end if
if $count\_neg \mod 2 = 1$ **then**
  if $count\_neg = 1$ **and** $count\_zero > 0$
  and $(count\_neg + count\_zero) = n$ **then**
    return 0
  end if
  $product \leftarrow product / max\_neg$
end if
return $product$

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. [6]

Entrada: $S = banana$, $K = 2$

Salida: nn



De las opciones anteriores, nn es la lexicográficamente más grande.

[6] https://goo.gl/iwFFCA

Procedure 8 SUBSEQUENCE

Input: S : $String$, T : $String$, k : $Integer$

```
last ← 1
new_last ← 1
for ch ← 'z' to 'a' do
    count ← 0
    for i ← last to S.length do
        if S[i] = ch then
            count ← count + 1
        end if
    end for
    if count ≥ k then
        NEXT SLIDE
    end if
end for
return T
```

---

for $i \leftarrow last$ **to** $S.length$ **do**
  if $S[i] = ch$ **then**
    $T \leftarrow T + ch$
    $new\_last \leftarrow i$
  end if
end for
$last \leftarrow new\_last$

Introducción

La programación dinámica, al igual que Divide y conquista, resuelve problemas combinando soluciones a subproblemas; pero a diferencia de esta, se aplica cuando los subproblemas se solapan, es decir, cuando comparten problemas más pequeños. Aquí la técnica cobra importancia, ya que calcula cada subproblema una sola vez; esto es, parte del principio de no calcular dos veces la misma información. Por tanto, utiliza estructuras de almacenamiento como vectores, tablas, arreglos, archivos, con el fin de almacenar los resultados parciales a medida que se resuelven los subcasos que contribuyen a la solución definitiva.

- Es una técnica ascendente que, normalmente, empieza por los subcasos más pequeños y más sencillos. Combinando sus soluciones, obtenemos las respuestas para los subcasos cada vez más grandes, hasta que llegamos a la solución del problema original.
- Se aplica muy bien a problemas de optimización. El mayor número de aplicaciones se encuentran en problemas que requieren maximización o minimización, ya que se pueden hallar múltiples soluciones y así evaluar para determinar cuál es la óptima.

Forma general

La forma general de las soluciones desarrolladas mediante programación dinámica requiere los siguientes pasos:

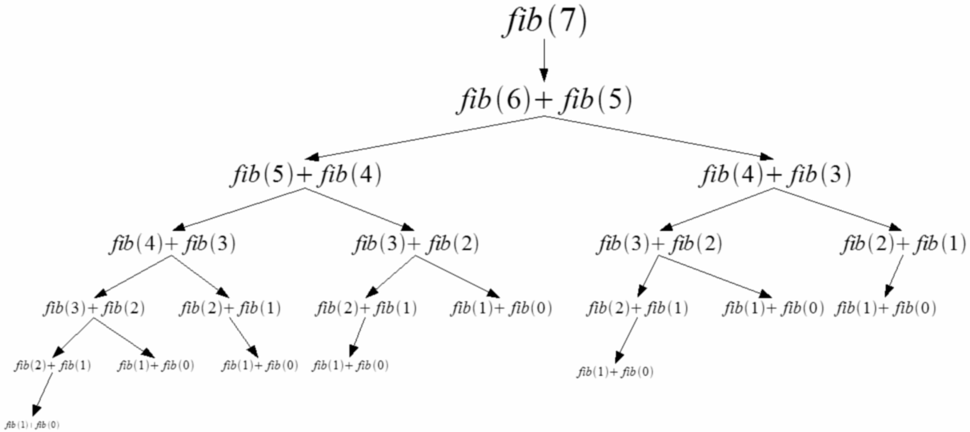
1. Plantear la solución, mediante una serie de decisiones que garanticen que será óptima, es decir, que tendrá la estructura de una solución óptima.
2. Encontrar una solución recursiva de la definición.
3. Calcular la solución teniendo en cuenta una tabla en la que se almacenen soluciones a problemas parciales para su reutilización, y así evitar un nuevo cálculo.
4. Encontrar la solución óptima utilizando la información previamente calcular y almacenada en las tablas.

Principio de optimalidad de Bellman: Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve.

Procedure 9 FIBONACCI

Input: n : *Integer*

```
if n < 1 then
  return −1
else if n = 1 or n = 2 then
  return 1
else
  return FIBONACCI(n − 1) + FIBONACCI(n − 2)
end if
```


fib(7)
fib(6)+ fib(5)
fib(5)+ fib(4)
fib(4)+ fib(3)
fib(4)+ fib(3)
fib(3)+ fib(2)
fib(3)+ fib(2)
fib(2)+ fib(1)
fib(3)+ fib(2)
fib(2)+ fib(1)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)

Procedure 10 FIBONACCI_WITH_MEMORY1

Input: $n : Integer, A : Array$

if $n < 1$ **then**
  return -1
else if $n = 1$ **or** $n = 2$ **then**
  return 1
else if $A[n] <> -1$ **then**
  return $A[n]$
else
  $A[n] \leftarrow FIBONACCI(n-1) + FIBONACCI(n-2)$
  return $A[n]$
end if

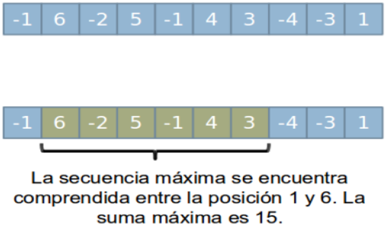
Procedure 11 FIBONACCI_WITH_MEMORY2

Input: $n : Integer, A : Array$

```
if n < 1 then
  return −1
else
  A[1] ← 1
  A[2] ← 1
  for i ← 3 to n do
    A[i] ← A[i − 1] + A[i − 2]
  end for
end if
```

Secuencia de suma máxima

Dado un arreglo de n números enteros positivos y negativos, encontrar los i elementos del arreglo cuya suma se la máxima posible.



Procedure 12 MAX_SUM(A:Array)

Input: n : $Integer$, A : $Array$

```
sum ← 0
ans ← 0
for i ← 1 to A.length do
    sum ← sum + A[i]
    ans ← MAX(ans, sum)
    if sum < 0 then
        sum ← 0
    end if
end for
return ans
```

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 : 2
  Explanation : The change will be (3 + 3) = 2

$$C[j] = \begin{cases} \infty & \text{if } j < 0, \\ 0 & \text{if } j = 0, \\ 1 + \min\limits_{1 \le i \le k} \{C[j - d_i]\} & \text{if } j \ge 1 \end{cases}$$

Procedure 13 COIN_CHANGE

Input: $S : Array, c : Integer$

$Aux : Array[0, c]$
$INIT\_ARRAY(A, \infty)$
$Aux[0] \leftarrow 0$
for $i \leftarrow 1$ to $S.length$ **do**
  for $j \leftarrow S[i]$ to c **do**
    $Aux[j] \leftarrow MIN(1 + Aux[j - S[i]], Aux[j])$
  end for
end for
return Aux[c]

Números feos

Los números feos son números cuyos únicos factores primos son 2, 3 o 5 (o combinación de ellos). La secuencia 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... muestra los primeros 11 números feos. Por convención, se incluye 1.

Como vemos en la diapositiva anterior, la secuencia de los primeros números feos es 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... Es más fácil de determinar la forma en que se genera está secuencia si la dividimos entre sus tres factores:

- 1x2, 2x2, 3x2, 4x2, 5x2...
- 1x3, 2x3, 3x3, 4x3, 5x3...
- 1x5, 2x5, 3x5, 4x5, 5x5...

¿Cuál se imprime primero?

Procedure 14 UGLY_NUMBER

Input: n : *Integer*

$A : Array[n]$

$A[1] = 1$

$i2 \leftarrow 1$

$i3 \leftarrow 1$

$i5 \leftarrow 1$

$ugly\_number \leftarrow 0$

for $i \leftarrow 2$ to n **do**

  next slide

end for

return $ugly\_number$

$ugly\_number = MIN((A[i2] * 2), (A[i3] * 3), (A[i5] * 5))$
$A[i] \leftarrow ugly\_number$
if $ugly\_number == (A[i2] * 2)$ **then**
  $i2 \leftarrow i2 + 1$
end if
if $ugly\_number == (A[i3] * 3)$ **then**
  $i3 \leftarrow i3 + 1$
end if
if $ugly\_number == (A[i5] * 5)$ **then**
  $i5 \leftarrow i5 + 1$
end if

Cuenta el número de formas posibles

Queremos dar un cambio de C pesos y tenemos un suministro infinito de monedas de valor S = $[S_1, S_2, \ldots, S_n]$ pesos, ¿de cuántas maneras podemos dar el cambio? Por ejemplo para C = 4, S = [1, 2, 3], existen cuatro formas de dar el cambio: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3]. Para C = 10, S = [2, 5, 3, 6], existen cinco soluciones: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5], [5, 5].

$$COUNT(type, value) = \begin{cases} 0, & \text{si } value < 0, \\ 1, & \text{si } value = 0, \\ COUNT(type + 1, value) & \\ +COUNT(type, value - coins[type]), & \text{si } value > type. \end{cases}$$

Procedure 15 COUNT

Input: $S : Array, c : Integer$

```
table : Matrix[0...S.length][0..c]
for i ← 0 to c do
    table[0][i] ← 0
end for
for i ← 0 to c do
    table[i][0] ← 1
end for
NEXT SLIDE
return table[S.length][c]
```

```
for i ← 1 to S.length do
    for j ← 1 to c do
        /* Can not the currency S[i] be used to give the change? */
        if S[i] > j then
            table[i][j] ← table[i − 1][j]
        else
            /* If we don't use S[i] */
            table[i][j] ← table[i − 1][j] + table[i][j − S[i]]
        end if
    end for
end for
```