Semana 1

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

07-2021

Análisis de complejidad

Notaciones

- Notación Big Theta - Θ
- Notación Big O - O
- Notación Big Omega - Ω
- Jerarquía de los algoritmos

Reglas prácticas para el cálculo de la complejidad

- Sentencias simples
- Condicionales
- Ciclos
- Algoritmos recursivos

Conjuntos

Un conjunto es una colección de elementos bien definida, en la cual no existen elementos repetidos y el orden entre ellos no importa. En la representación de estos, se acostumbra a llamar a los conjuntos con letras mayúsculas y a los elementos genéricos con letras minúsculas. Un conjunto se puede definir de dos formas:

- Enumerativa: se colocan todos sus elementos entre llaves, separados por comas. Por ejemplo, $A = \{a, e, i, o, u\}$.
- Descriptiva: se coloca entre llaves una descripción de los elementos que lo forman. Por ejemplo, $A = \{\ x \mid x \textit{ es una vocal }\}$.

- Cuando un elemento a pertenece a un conjunto S, se escribe $a \in S$. Cuando un elemento a NO pertenece a un conjunto S, se escribe $a \notin S$.
- El número de elementos que contiene un conjunto A se denomina cardinalidad de A y se representa como $|A|$. Si la cardinalidad de un conjunto es un entero n, se dice que el conjunto es finito. En caso contrario, se dice que el conjunto es infinito.
- Si todos los elementos de un conjunto A están también en el conjunto B, se dice que A es un subconjunto de B y que B es un superconjunto de A y se escribe $A \subseteq B$. Si A es un subconjunto de B y no son iguales se puede escribir $A \subset B$ y se dice que A es un subconjunto propio de B.

- El conjunto formado por todos los posibles subconjuntos de un conjunto A se llama conjunto potencia de A. Si $|A| = n$, la cardinalidad de su conjunto potencia es 2^n, por lo que es común representar el conjunto potencia de A como 2^A.
- Si se quiere saber cuántos subconjuntos de cardinalidad k se pueden tener de un conjunto de cardinalidad n, la respuesta se obtiene por medio de las combinaciones de n en k.

$$C(n,k) = nCk = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Operaciones sobre conjuntos:

- La intersección de dos conjuntos A y B está formada por todos los elementos que están en A y en B, y se presenta como $A \cap B$.
- La unión de dos conjuntos A y B está formada por todos los elementos que están en A o en B (o en ambos) y se representa como $A \cup B$.
- La diferencia de dos conjuntos A y B está formada por todos los elementos que están en A, pero no están en B y se representa como $A \breve{} B$.
- El complemento de un conjunto A está formado por todos los elementos que no están en el conjunto A y se representa como A'.

Tuplas

- Una tupla es una sucesión finita de elementos, donde el orden sí importa. Se representa por los elementos colocados entre paréntesis y separados por comas. Por ejemplo, la tupla (a, b, c) es diferente de la tupla (b, c, a).
- Si la tupla tiene dos elementos se le conoce como "par", si tiene tres, se le conoce como "tercia", pero si tiene más se complican los nombres, por lo que, en general, a una tupla de k elementos se le conoce como una k-tupla.

- Es muy importante recordar que el orden en los elementos de una tupla sí importa. Por ejemplo, los puntos en un plano cartesiano se pueden representar por un par ordenado o 2-tupla, llamada coordenadas del punto.
- Los elementos de una tupla pueden ser de diferentes tipos. En algunas aplicaciones de Ciencias Computacionales como las Bases de Datos o la Ciencia de Datos, los datos se pueden representar por medio de una tupla, en donde el primero elemento corresponde al valor de una variable o característica (feature) en ciencia de datos, el segundo al valor de otra variable y así sucesivamente.

Las tuplas nos permiten definir una operación más entre dos conjuntos, llamada productor cruz y representada con el símbolo X. El producto cruz de dos conjuntos A y B, representado por $A \text{ X } B$, está formado por todas las parejas ordenadas en las que el primer elemento pertenece a A y el segundo pertenece a B. Por ejemplo, si $A = \{a, b, c\}$ y $B = \{1, 2\}$, el producto cruz $A \text{ X } B$ es:

$$A \text{ X } B = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$$

Relaciones

- Una relación es cualquier subconjunto de un producto cruz, es decir, es un conjunto donde sus elementos son tuplas. Una relación muy común en las matemáticas es la de "menor que" entre los elementos del conjunto de los números enteros, denotado por $\mathbb{Z}$.
- Si la relación R es un subconjunto del producto cruz de A X A, es decir, $R \subseteq A$ X A, se dice que la relación es:
  - Reflexiva: si para toda $x \in A$, $(x, x) \in R$.
  - Simétrica: si para $(x, y) \in R$, existe $(y, x) \in R$.
  - Transitiva: si siempre que $(x, y) \in R$ y $(y, z) \in R$, entonces también $(x, z) \in R$.

Relaciones

Las funciones son un tipo especial de relación. Dada una relación del conjunto A al conjunto B, dicha relación sería una función si cumple con la siguiente condición especial: cada elemento de A debe estar relacionado con uno y sólo un elemento de B.

- La función $floor(x)$ devuelve el entero más pequeño o igual a x. Por ejemplo, $floor(3.3) = floor(3.99999) = floor(3.5) = 3$.
- La función $ceil(x)$ devuelve el entero más grande o igual a x. Por ejemplo, $ceil(3.3) = ceil(3.99999) = ceil(3.5) = 4$.

Logaritmos

- $\log_b a$ es una función estrictamente creciente.
- $\log_b 1 = 0$.
- $\log_b b^a = a$.
- $\log_b(XY) = \log_b X + \log_b Y$
- $\log_b X^a = a \log_b X$
- $X^{\log_b Y} = Y^{\log_b X}$
- $\log_c X = \frac{\log_b x}{\log_b c}$

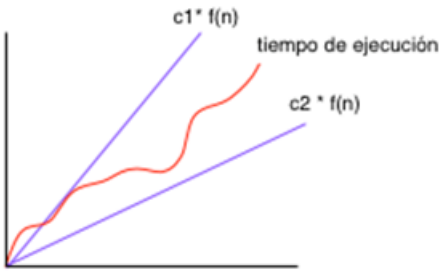
Sucesiones

Las sucesiones son en realidad un tipo especial de función que tiene por dominio un conjunto de enteros consecutivos, los cuales indican las posiciones de los elementos dentro de la posición, por lo que se les conoce como índices. El n-ésimo término de una sucesión S se denomina S_n o con notación de función $S(n)$. Los elementos de una sucesión pueden ser identificados por la posición que guardan dentro de la sucesión, donde, el primer índice puede ser cualquier entero y los que le siguen serán consecutivos.

1. $\sum_{i=1}^{n} c = c * n$
2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6}$

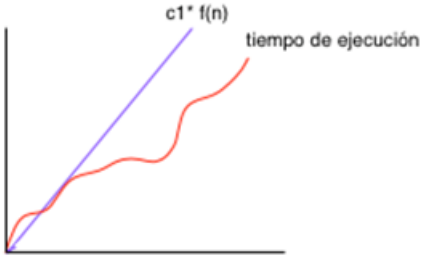
Notación Big Theta - Θ

En general cuando tenemos una función de tiempo de ejecución $f(n)$, cuando n es suficientemente grande, el tiempo de ejecución estará entre $c1 * f(n)$ y $c2 * f(n)$. Mientras existan contantes $c1$ y $c2$ que delimiten $f(n)$ para n muy grandes decimos que el tiempo de ejecución del algoritmo es $\Theta(n)$.



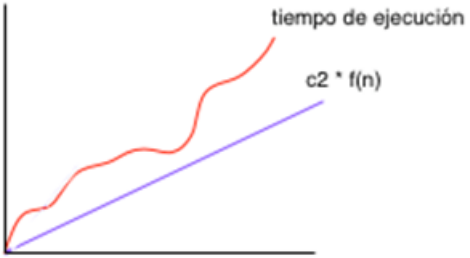
Notación Big O - O

Usaremos la notación Big-O para las cotas superiores asintóticas para entradas muy grandes y se dice que el tiempo de ejecución es "Big O de f(n", "O de f(n" o simplemente "Orden de f(n". Entonces podemos decir que la búsqueda secuencial $O(n)$.



Notación Big Omega - Ω

La Notación Big Ω es cuando se desee decir que un algoritmo toma por lo menos cierta cantidad de tiempo, sin querer ofrecer la cota superior. Ya que la notación Big Ω es para mostrar lo límites asintóticos inferiores.



Jerarquía de los algoritmos

Notación O	Nombre
$O(1)$	Constante
$O(\log(n))$	Logarítmica
$O(n)$	Lineal
$O(n \log n)$	n log n
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(n^m)$	Polinomial
$O(m^n)$ m >= 2	Exponencial
$O(n!)$	Factorial

Sentencias simples

La sentencias simples son aquellas que ejecutan operaciones básicas, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño está relacionado con el tamaño del problema. La inmensa mayoría de las sentencias simples requieren un tiempo constante de ejecución y su complejidad es $O(1)$.
Ejemplos:

$x \leftarrow 1$
$y \leftarrow z + x + w$
print x
read x

Condicionales

Los condicionales suelen ser $O(1)$, a menos que involucren un llamado a un procedimiento, y siempre se debe tomar la peor complejidad posible de las alternativas del condicional, bien en la rama afirmativa o bien en la rama positiva. En decisiones múltiples (*switch*) se tomará la peor de todas las ramas. Ejemplo:

if $a > b$ **then**
  for $i \leftarrow 1$ **to** n **do**
    $sum \leftarrow sum + 1$
  end for
else
  $sum \leftarrow 0$
end if

Ciclos (while, for, repeat-until)

En los ciclos con un contador explícito se distinguen dos casos: que el tamaño n forme parte de los límites del ciclo, con una complejidad basada en n, o que dependa de la forma como avanza el ciclo hacia su terminación.

Si el ciclo se realiza un número constante de veces, independientemente de n, entonces la repetición solo introduce una constante multiplicativa que puede absorberse, lo cual da como resultado $O(1)$.
Ejemplo:

for $i \leftarrow 1$ **to** k **do**
  sentencias simples $O(1)$
end for

Si el tamaño n aparece como límite de las iteraciones, entonces la complejidad será: n * $O(1) \rightarrow O(\text{n})$.

Si los ciclos son aninadados...
Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    sentencias simples $O(1)$
  end for
end for

En este caso, la complejidad sería: n * n * $O(1) \rightarrow O(n^2)$.

Para ciclos anidadados pero con variables independientes:
Ejemplo:

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** i **do**
    sentencias simples $O(1)$
  end for
end for

$$\sum_{i=1}^{n} \sum_{j=1}^{i} O(1) = \sum_{i=1}^{n} i = \frac{n(n-1)}{2} = O(n^2)$$

A veces aparecen ciclos multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores):
Ejemplo:

$c \leftarrow 1$
while $c < n$ **do**
  $c \leftarrow c * 2$
end while

El valor inicial de la variable c es 1, y llega a 2^n al cabo de n iteraciones $\rightarrow \log_2 n$.

Y la combinación de los anteriores:
Ejemplo:

```
for i ← 1 to n do
  c ← n
  while c > 0 do
    c ← c/2
  end while
end for
```

Se tiene un ciclo interno de orden $O(\log_2 n)$ que se ejecuta n veces en el ciclo externo; por lo que, el ejemplo es de orden $O(n \log_2 n)$.

Algoritmos recursivos

Para poder analizar la eficiencia de los algoritmos recursivos, se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva. Normalmente se comportan de una de las siguientes formas:

- $O(n)$ - Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.
- $O(\log_b n)$ - Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.
- $O(C^n)$ - Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.
- $O(n^{\log_b c})$ - Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.

Introducción

- La historia de la computación está compuesta por la historia del hardware y del software. A su vez, la historia del software es, en gran medida, la historia de los algoritmos.
- Los algoritmos son una de las partes medulares de los sistemas computacionales. Los científicos de la computación se han enfrentado a problemas cuya solución se da por medio de un algoritmo y la aparición de estos ha sido fundamental para el desarrollo de la computación, sobre todo cuando logran solucionar problemas que se presentan a menudo en los sistemas.

¿Cómo se diseña un algoritmo?

- No existe "LA TÉCNICA"de diseño de un algoritmo.
- Una buena forma de aprender a diseñar nuevos algoritmos o inclusive, nuevos enfoques para el diseño de algoritmos, es estudiar y entender los enfoques que se han desarrollado y que han sido muy exitosos.
- ¿Cuáles son estos enfoques?
  - Divide y vencerás.
  - Algoritmos ávidos.
  - Programación dinámica.
  - Backtracking.
  - Brach and bound.

Pero, además ...

Debemos recordar que la implementación de un algoritmo siempre va acompañada de una estructura de datos y que la selección de dicha estructura afecta directamente la eficiencia del algoritmo, por lo que también será necesario recordar algunas de las estructuras de datos clásicas o aprender algunas nuevas, a efecto de seleccionar la mejor.

Para crear un nuevo algoritmo, existen dos posibilidades:

- Crear un algoritmo que resuelva un problema que hasta el momento no tenía solución.
- Crear un algoritmo cuya eficiencia sea mejor que la de los algoritmos previos que resuelven el mismo problema.

Divide y vencerás

Es una técnica que permite encontrar la solución de un problema descomponiéndolo en subproblemas más pequeños (dividir) y que tienen la misma naturaleza del problema original, es decir, son similares a este. Luego resuelve cada uno de los subproblemas recursivamente hasta llegar a problemas de solución trivial o conocida con antelación (conquistar) para, finalmente, unir las diferentes soluciones (combinar) y así conformar la solución global al problema.

Procedure 1 DIVIDE_AND_CONQUER

Input: X
  if X is simple or known **then**
    return $SOLUTION(X)$
  else
    Decompose X into smaller problems $x_1, x_2, ..., x_n$
    for $i \leftarrow 1$ **to** n **do**
      $y_i \leftarrow DIVIDE\_AND\_CONQUER(X_i)$
    end for
    Combine the y_i to get the Y that is solution of X
    return Y
  end if

Exponenciación rápida

Dados dos números, x y n, calcular el resultado de x^n, haciendo uso de la técnica de dividir y conquistar.

Procedure 2 FAST_POW

Input: $x : Real, n : Integer$
  if $n < 0$ **then**
    return $FAST\_POW(1/x, -n)$
  else if $n == 0$ **then**
    return 1
  else if $n == 1$ **then**
    return x
  else if $n \mod 2 = 0$ **then**
    return $FAST\_POW(x * x, n/2)$
  else if $n \mod 2 = 1$ **then**
    return $x * FAST\_POW(x * x, (n-1)/2)$
  end if

Prefijo común más largo

Dado un conjunto de cadenas, encontrar el prefijo común más largo. Por ejemplo, dadas la siguiente cadenas: “geeksforgeeks”, “geeks”, “geek”, “geezer”, el resultado esperado es: "gee". [1]

[1] https://goo.gl/rqjV76

Procedure 3 FIND_PREFIX

Input: $A : String, B : String$

```
result ← ""
i ← 1
j ← 1
while i < A.length and j < B.length do
    if A[i] <> B[j] then
        break
    end if
    result ← result + A[i]
    i ← i + 1
    j ← j + 1
end while
return result
```

Procedure 4 COMMON_PREFIX

Input: $A : Array, low : Index, high : Index$

```
if low == high then
    return A[low]
end if
if low < high then
    mid ← FLOOR((high + low)/2)
    str1 ← COMMON_PREFIX(A, low, mid)
    str2 ← COMMON_PREFIX(A, mid + 1, high)
    return FIND_PREFIX(str1, str2)
end if
```

Contando inversiones

Dado arreglo de números enteros distintos, A, determinar el número de inversiones que existen. Decimos que dos indices i $<$ j forma una inversión si $A[i] > A[i]$.

El número de inversiones de un arreglo indica: a qué distancia (o qué tan cerca) está el arreglo de estar ordenado. Si el arreglo ya está ordenado, el conteo de inversión es 0. Si el arreglo está ordenado en orden inverso, el conteo de inversión es el máximo.

Ejemplo: ¿cuántas inversiones hay en el siguiente arreglo: 2, 4, 1, 3, 5? Tiene tres inversiones (2, 1), (4, 1), (4, 3). [2]

[2] https://goo.gl/DxqxBe

Procedure 5 SORT_AND_COUNT

Input: $A, B : Array, low, high : Index$

$r \leftarrow 0$
$left \leftarrow 0$
$right \leftarrow 0$
if $(high - low + 1 = 1)$ **then**
  return 0
else
  $mid \leftarrow FLOOR((high + low)/2)$
  $left \leftarrow SORT\_AND\_COUNT(A, B, low, mid)$
  $right \leftarrow SORT\_AND\_COUNT(A, B, mid + 1, high)$
  $r \leftarrow MERGE\_AND\_COUNT(A, b, low, mid, high)$
  $COPY(A, B, low, high)$
end if
return $r + left + right$

Procedure 6 MERGE_AND_COUNT

Input: $A, B : Array.low, mid, high : Index$

...
while $left \leq mid$ AND $right \leq high$ **do**
  if $A[left] < A[right]$ **then**
    $B[i] \leftarrow A[left]$
    $left \leftarrow left + 1$
  else
    $B[j] \leftarrow A[right]$
    $right \leftarrow right + 1$
    $count \leftarrow count + (mid - left)$
  end if
  $i \leftarrow i + 1$
end while
...
return $count$

Encontrar el número más cercano en el arreglo

Dado un arreglo de números enteros ordenados. Necesitamos encontrar el valor más cercano al número dado. El arreglo puede contener valores duplicados y números negativos. [3]

[3] https://goo.gl/XTgBzX

Procedure 7 FIND_CLOSEST

Input: $A : Array, target : Key$
  if $target < A[1]$ **then**
    return $A[1]$
  end if
  if $target > A[A.length]$ **then**
    return $A[A.length]$
  end if
  $low \leftarrow 1$
  $high \leftarrow A.length$
  while $low < high$ **do**
    NEXT SLIDE
  end while

$mid \leftarrow FLOOR((high + low)/2)$
if $target = A[mid]$ **then**
  return $A[mid]$
else if $target < A[mid]$ **then**
  if $mid > 0$ **and** $target > A[mid - 1]$ **then**
    return $CLOSEST(A[mid - 1], A[mid], target)$
  end if
  $high \leftarrow mid$
else
  if $mid < A.length$ **and** $target < A[mid + 1]$ **then**
    return $CLOSEST(A[mid], A[mid + 1], target)$
  end if
  $high \leftarrow mid$
end if