Introducción a los Algoritmos

Andrés Becerra Sandoval

30 de julio de 2007

Resumen

Haremos una introducción a los principales temas de todo el curso de una forma panorámica, es decir, sin hacer énfasis en los detalles. Mas adelante retomaremos los mismos temas con un mayor nivel de profundidad. El avance lo haremos a través de preguntas y de mapas conceptuales. Un mapa conceptual une *conceptos* a través de *proposiciones* o frases. Un mapa conceptual responde siempre a una *pregunta de enfoque*. En estas notas tenemos tres mapas que responden a las preguntas ¿Cual es la importancia de los algoritmos?, ¿Cual es el rol de la abstracción en el estudio de algoritmos? ¿Como medimos la eficiencia de los algoritmos?

¿Como especificar un algoritmo?

- ¿En español? No, el lenguaje natural tiene muchas ambiguedades intrínsecas que dificultarían la comprensión de los algoritmos.
- ¿Con un diagrama de flujo? No, un algoritmo largo ocupa mucho espacio cuando se expresa en un diagrama de flujo.
- ¿En un lenguaje de programación muy usado? (Java, C#, C) ? Sería un error privilegiar un lenguaje sobre todos los demás.
- ¿En un lenguaje de máquina?. Sería un problema, porque estos lenguajes dependen de una familia específica de procesadores (Pentium IV por ejemplo), y esto sería privilegiar una arquitectura de computador sobre todas las demás.

- ¿En pseudocódigo? Si, permite implementar los algoritmos en cualquier otro lenguaje de alto y bajo nivel. No se compromete (si somos cuidadosos) con los detalles de una arquitectura de computador en especial, y, el pseudocódigo elimina las ambiguedades del lenguaje natural. Para éste curso usaremos clrscode, un paquete para LaTeXque ustedes pueden encontrar en `http://www.cs.dartmouth.edu/~thc/clrscode/`. Además el texto guía[1] documenta las convenciones de éste tipo de pseudocódigo en la página 19, y las va aumentando paulatinamente.

Para poner manos a la obra con la notación que vamos a usar aquí hay algunos ejemplos de algoritmos:

```
RUM(y, z, A)
  if z = 0
    then return A

  if impar(z)
    then return RUM(2y,⌊z/2⌋,A+y)
    else  return RUM(2y,⌊z/2⌋, A)
```

Esto realiza la *multiplicación rusa* entre **y** e **z,** pruebe a implementar este algoritmo en su lenguaje de programación favorito. Para valores positivos de y,z debe producir $y \times z$ como resultado.

A continuación algo mas familiar, la multiplicación de dos matrices de números:

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2    then error "incompatible dimensions"
3    else
4          for i ← 1 to rows[A]
5              do for j ← 1 to columns[B]
6                     do C[i, j] ← 0
7                        for k ← 1 to columns[A]
8                            do C[i, j] ← C[i, j] + A[i,k] × B[k, j]
9          return C
```

Observe bien que la indentación marca los bloques, a la python. Que los índices para los arreglos de dos dimensiones se acceden con la notación [fila,columna], que la asignación se hace con una flecha, y que los atributos de las variables (como *columns* y *rows*) se acceden con la notación atributo[variable]. En éste segundo ejemplo las líneas aparecen numeradas, lo cual sirve para explicar porciones del código.

¿Como le explicamos a alguien un algoritmo?

- ¿Mostrando el pseudocódigo, y esperando que el otro *se la pille*?. No, para ciertos algoritmos, el código o pseudocódigo no permite comprender fácilmente lo que hacen.

- ¿Haciendo seguimientos? No. Los seguimientos ayudan a validar algunos casos de prueba, lo que aumenta nuestra confianza en el algoritmo, pero, si los seguimientos son largos y complejos, podemos hacerlos perfectamente de inicio a final *sin entender* el algoritmo. Una prueba de esto mediante un experimento mental la pueden encontrar en el argumento del salón chino del filósofo John Searle (chinesse room).

- ¿Mostrando los invariantes? Si, la clave para comprender un algoritmo radica en comprender el invariante o los invariantes que obedece. Estos invariantes, a menudo, se pueden representar graficamente, de manera que con un vistazo podemos saber que hace un algoritmo, sin necesitar el código para entenderlo. Después de comprender un algoritmo, si utilizaremos el pseudocódigo para analizarlo.

¿Como garantizamos la corrección de un algoritmo?

- ¿A través de casos de prueba? No. El número de entradas para un algoritmo puede ser astronomicamente grande, y las pruebas no pueden recorrer todos los casos de prueba.

- ¿Con una demostración formal?. Si y no. La demostración es la única forma rigurosa de afirmar que un algoritmo hace lo que debe hacer, pero las demostraciones también pueden presentar bugs. Asi que lo mejor es combinar demostraciones con casos de prueba.

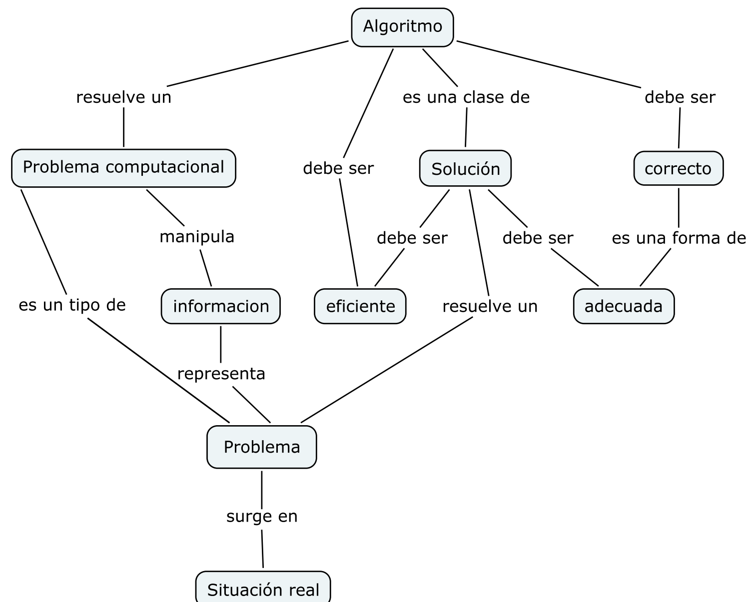
¿Como medimos la eficiencia de un algoritmo?

- ¿Empíricamente, corriéndolo con datos de entrada y midiendo el tiempo? No. El comportamiendo de un algoritmo para diferentes entradas puede variar enormemente, hay casos fáciles y casos difíciles.

- ¿Comparativamente, tomando dos algoritmos y decidiendo de alguna manera cual es el mejor? Si y no. Si los comparamos improvisadamente podemos obtener resultados erroneos.

La forma correcta consiste en tener una *métrica* o medida de la eficiencia para cada algoritmo por separado, para compararlas debidamente. Mas adelante veremos que la métrica puede utilizarse en colaboración con las medidas empíricas para obtener resultados mas precisos.

1. Importancia de los algoritmos



Como ingenieros tenemos que resolver *problemas* que se dan en la vida real. Las *soluciones* que proponemos deben ser *adecuadas* al contexto y *eficientes* con respecto al consumo de recursos. Como ingenieros informáticos muchas soluciones a problemas son *algoritmos,* que deben ser correctos y eficientes en consumo de *tiempo y memoria*.

Desde este punto de vista un algoritmo es una creación científica y tecnológica, que, aunque intangible en la forma de pseudocódigo, puede transformarse en un producto de software mediante su traducción a un lenguaje de programación específico.

2. El análisis de los algoritmos

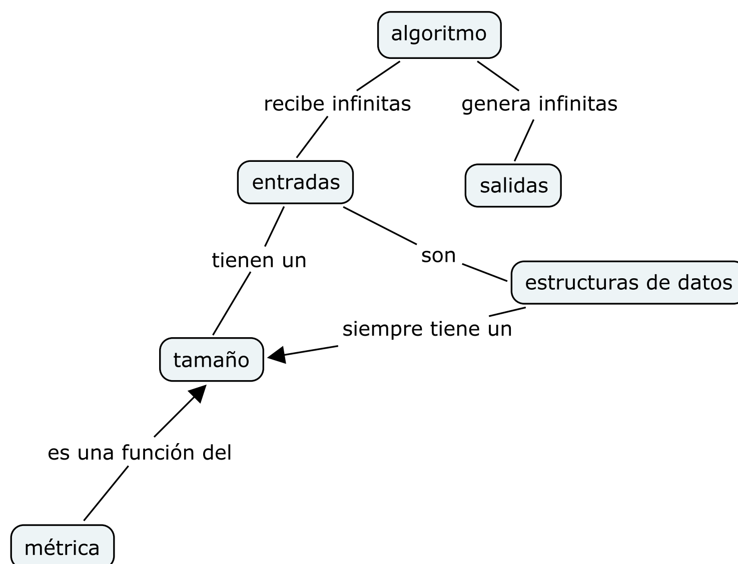
2.1. Corrección

Para algoritmos iterativos utilizaremos invariantes de ciclo y para los recursivos inducción estructural. Para algoritmos que son iterativos y recursivos al mismo tiempo mezclaremos las dos técnicas probando la corrección del algoritmo por partes. Los números de línea en los trozos de pseudocódigo nos permitirán demarcar las partes sobre las que estamos probando algo.

Si un algoritmo utiliza subrutinas, la forma de demostrar que funciona correctamente consiste en:

- Demostrar, por separado, que cada subrutina funciona correctamente
- Asumiendo que las subrutinas funcionan bien, realizar la demostración global de la corrección

2.2. Eficiencia



La complejidad o eficiencia temporal de un algoritmo se refiere a la medida del tiempo que tardaría un algoritmo en operar sobre sus entradas. Para medir este tiempo tenemos que considerar que tan grandes son estas entradas, esto es, debemos definir un tamaño apropiado para las estructuras de datos que codifican la entrada del algoritmo. Si la entrada está dada por un arreglo un buen tamaño es el número de elementos, si es una matriz un buen tamaño es el número de filas por el número de columnas.

La métrica o medida de eficiencia de un algoritmo, entonces, es una *función* del *tamaño* de las entradas que recibe un algoritmo. Si utilizamos una variable para indicar el tamaño de las entradas de un algoritmo (como n, por ejemplo), entonces la métrica para la eficiencia de un algoritmo será una función $f(n)$. Por ejemplo, un algoritmo que tenga como entrada un arreglo, tendría como complejidad alguna función de n, como $f(n) = n^2$ ó $f(n) = 3n$

Si estamos calculando el consumo de memoria en función del tamaño de las entradas, nos referiremos a complejidad espacial. Si estamos calculando el consumo de tiempo de procesador, nos referiremos a complejidad temporal. En áreas específicas de la computación como en las redes de computadoras, podríamos estar interesados en medir el consumo de ancho de banda, pero las técnicas de análisis que vamos a ver en el curso también pueden adaptarse a esas otras situaciones.

2.2.1. ¿Como construimos una función del tamaño de las entradas de un algoritmo?

- Una forma es a través de fuerza bruta. Esto consiste en analizar un algoritmo línea a línea como en la página 24 del libro guía. A cada instrucción básica (while, if, asignación, comparación, for) se le asigna un tiempo constante que dependerá del ambiente en el que se implemente el algoritmo (lenguaje de programación + compilador + arquitectura del procesador). La suma global contendrá una gran cantidad de constantes y variables que indican el tamaño de las entradas del algoritmo (como n).

- Una mejora al proceso de de conteo línea a línea consiste en definir una operación básica, que es la que domina el tiempo de ejecución del algoritmo. En el contexto de ordenamiento una operación básica puede ser la comparación, o el intercambio de un par de elementos de un arreglo. En el contexto de cálculos matemáticos una operación básica puede ser una multiplicación o división, en el contexto de redes y criptografía una operación básica puede ser la transformación de un bloque de bytes con un tamaño determinado.

- Si el algoritmo es recursivo, plantearemos una *ecuación de recurrrencia*. La solución de una ecuación de recurrencia siempre es una *función*.

2.2.2. ¿Como comparamos la eficiencia de dos algoritmos?

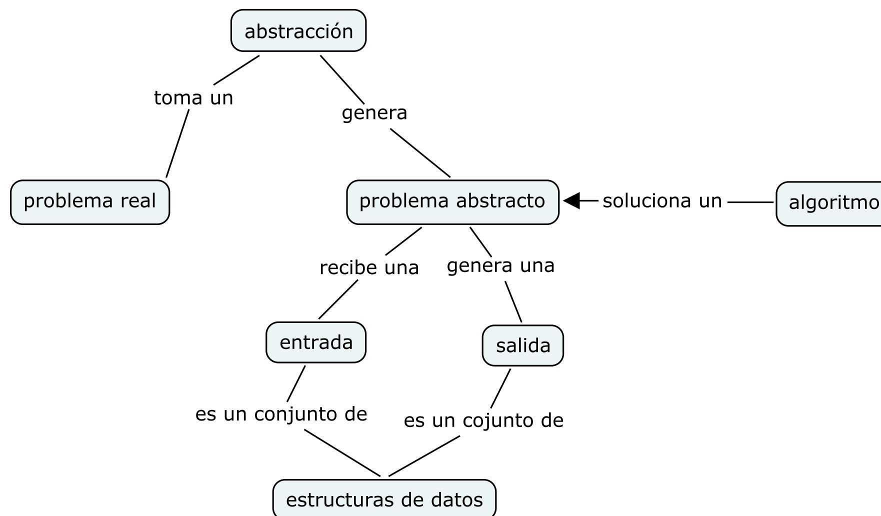
Como la eficiencia de los algoritmos va a medirse por medio de funciones, necesitaremos una teoría de comparación de funciones. Esto nos lo provee el estudio de los ordenes de crecimiento (las notaciones O, Θ, Ω y sus propiedades). Esta teoría elimina detalles engorrosos de las funciones y nos permite concentrarnos en lo mas importante: como aumenta el tiempo de ejecución de un algoritmo como consecuencia de un aumento en el tamaño de las entradas.

2.2.3. ¿Como resolvemos ecuaciones de recurrencia?

Nos vemos en la necesidad de tener métodos de solución de recurrencias. Existen varios que veremos mas adelante:

- Iterar la recurrencia algebraicamente para armar una serie que se puede resolver con técnicas del Cálculo.
- Plantear un árbol de recursión.
- Utilizar el teorema maestro.
- Adivinar la solución y demostrarla por inducción matemática.
- Usando funciones generatrices (generating functions), aunque esta técnica está fuera del alcance de nuestro curso.

3. Problemas abstractos



En la vida real nos piden soluciones a problemas muy específicos. Pero lo que vemos en clase y en libros de texto no son problemas tan específicos, son mucho mas genéricos, mas abstractos. Un problema abstracto trabaja sobre unas entradas para producir unas salidas que cumplen unas restricciones definidas. Las entradas y salidas están dadas como estructuras de datos computacionales (variables, arreglos, listas, árboles, grafos, entre otras). Para nosotros es importante realizar el proceso de la abstracción que, a partir de un problema real, genera un problema abstracto. El proceso inverso, a partir del problema abstracto, devolverse al mundo real no es tema de este curso, pero es muy importante, ya que involucra el diseño cuidadoso de sistemas que actúen en un contexto social (tema de la ingeniería de software), de interfaces de usuario que sean funcionales y fáciles de usar (tema de la interacción humano-computador que verán en computación gráfica).

Un problema abstracto transforma un problema real para dejarlo en términos de una entrada y una salida especificadas como estructuras de datos. Un ejemplo de este tipo de problemas es :

- MENOR(A), hallar el menor elemento A_k de un arreglo. Esto es: $\forall i A_k \leq A_i$

El libro considera dos problemas básicos en los primeros capítulos: ordenamiento y búsqueda. Según Donald Knuth, todos los detalles interesantes del estudio de los algoritmos surgen de alguna manera en el contexto del ordenamiento y la búsqueda. Estos dos problemas abstractos son:

- ORDENAR(A), permutar el arreglo A de forma que el resultado sea A', tal que $A'_1 \leq A'_2 \leq \ldots A'_n$ (este es ordenamiento ascendente, aunque el descendente sería casi igual)

- BUSCAR(A,x), buscar la posición de x en el arreglo A (si está contenido). Esto es, debe retornar k, tal que $x = A_k$.

Observe que la mayor utilidad de la noción de problema abstracto reside en que va a permitir:

1. Demostrar que un algoritmo propuesto es correcto con respecto a la especificación de la *salida*

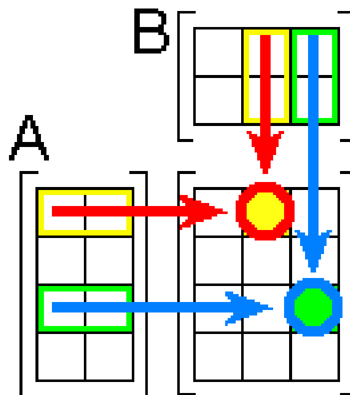
2. Calcular la complejidad de los algoritmos propuestos en función del *tamaño de la entrada*

Esto implica que antes de empezar el análisis de cualquier algoritmo es necesario e indispensable escribir el problema de forma abstracta

4. Presentación de varios algoritmos

Siguiendo con la idea de que la clave para comprender un algoritmo es captar sus invariantes, vamos a presentar algunos ejemplos sencillos de esta manera, antes de presentar el pseudocódigo:

4.1. Multiplicación de matrices



La multiplicación de dos matrices $A_{m\times n}$, $B_{n\times p}$ resulta en un matrix $C = AB$ con dimensiones $m \times p$. Cada elemento de la matrix producto es el producto punto entre una fila de A y una columna de B. Algebraicamente esto se puede expresar por la siguiente formula:

$$(AB)_{ij} = \sum_{r=1}^{n} a_{ir}b_{rj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \tag{1}$$

El algoritmo podemos obtenerlo mediante refinamiento. La primera aproximación:

```
MULT(A,B)
1  m ← filas[A]
2  p ← columnas[B]
3  for i ← 1 to m
4       do for j ← 1 to p
5               do
6                   C[i, j] ← fila_i[A] • columna_i[B]
7  return C
```

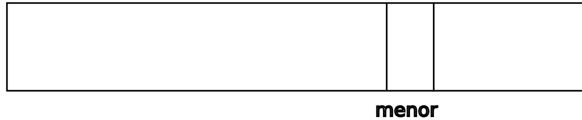
La línea 4 esconde mucho trabajo, que podemos pensar como un subproblema: el producto punto entre dos arreglos, uno está dado por una fila de A, y el otro por una columna de B. Calcular este producto requiere otro ciclo que multiplique cada elemento de A con uno de B, y que acumule el resultado:

```
MULT(A,B)
1  m ← filas[A]
2  p ← columnas[B]
3  n ← columnas[A]
4  for i ← 1 to m
5       do for j ← 1 to p
6               do C[i, j] ← 0
7                   for k ← 1 to n
8                        do C[i, j] ← A[i,k] × B[k,i]
9  return C
```

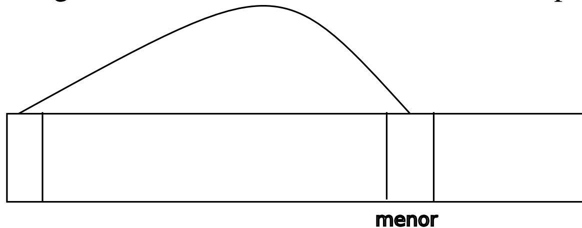
La demostración de corrección la podemos hacer estableciendo un adecuado invariante de ciclo, que en la última iteración nos permita deducir la formula (1).

4.2. Ordenamiento por selección

Este fue el primer algoritmo de ordenamiento que se me vino a la mente. Primero se encuentra la posición del menor elemento del arreglo:



Luego se intercambia éste elemento con el primero:



Ahora, se resuelve el subproblema de ordenar el arreglo sin considerar el primer elemento. Una observación interesante: si los datos se ponen en una cola de prioridad, en la que seleccionar el menor elemento es una operación rápida, obtendremos un algoritmo de ordenamiento mas eficiente (Heapsort), que toma su nombre de los Heaps (árboles binarios codificados en un arreglo, que preservan el invariante de tener siempre en la raíz el elemento menor, o el mayor).

4.3. Ordenamiento por inserción

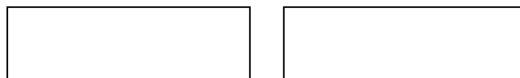
Este algoritmo se explica mejor en el contexto de ordenar un conjunto de cartas:

Para ordenar las cartas mientras las sostenemos en una mano, hace falta deslizar el las cartas 5,6 y 7 de corazones hacia la derecha, para dejar el lugar en el que debe ser insertado el 4 de corazones. El invariante de insertionSort consiste en agrandar una región de cartas (o elementos) ordenadas de una en una, insertando la carta que está inmediatamente a la derecha de la región en el lugar en el que debe ir para preservar el orden. Esta carta o elemento que está afuera de la región es la llave (key) a insertar.

En [1], es el primer ejemplo con el que se introduce el análisis de algoritmos.

4.4. Ordenamiento por mezclas

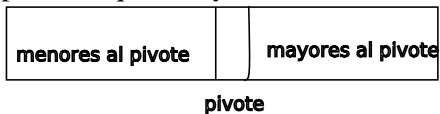
MergeSort toma el arreglo, lo separa en dos regiones:



Las ordena recursivamente, y luego las mezcla, con un algoritmo sencillo (merge).

4.5. Ordenamiento Quicksort

Hoare se inventó ésta técnica que se basa en pivotar (escoger un elemento de la arreglo), para dejar a su izquierda los menores a este, y a la derecha los mayores a este. Una ves se ha pivotado (o particionado), se procede a ordenar recursivamente las partes izquierda y derecha.



Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.