UADER - FACULTAD DE CIENCIA Y TECNOLOGÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS AÑO 2012

APUNTES DE ALGORITMIA – Parte 1

Se entiende por Algoritmia la ciencia dedicada al estudio de algoritmos. Principalmente comprende técnicas de diseño de algoritmos y el análisis y evaluación de los mismos.

1. Análisis de algoritmos

Como medida de la eficiencia de un algoritmo, se suelen estudiar los recursos (memoria y tiempo) que consume el algoritmo. El análisis de algoritmos se ha desarrollado para obtener valores que de alguna forma indiquen (o especifiquen) la evolución del gasto de tiempo y memoria en función del tamaño de los valores de entrada.

El análisis y estudio de los algoritmos es una disciplina de las ciencias de la computación y, en la mayoría de los casos, su estudio es completamente abstracto sin usar ningún tipo de lenguaje de programación ni cualquier otra implementación; por eso, en ese sentido, comparte las características de las disciplinas matemáticas. Así, el análisis de los algoritmos se centra en los principios básicos del algoritmo, no en los de la implementación particular. Una forma de plasmar (o algunas veces "codificar") un algoritmo es escribirlo en pseudocódigo o utilizar un lenguaje muy simple tal como Léxico, cuyos códigos pueden estar en el idioma del programador.

Algunos escritores restringen la definición de algoritmo a procedimientos que deben acabar en algún momento, mientras que otros consideran procedimientos que podrían ejecutarse eternamente sin pararse, suponiendo el caso en el que existiera algún dispositivo físico que fuera capaz de funcionar eternamente. En este último caso, la finalización con éxito del algoritmo no se podría definir como la terminación de este con una salida satisfactoria, sino que el éxito estaría definido en función de las secuencias de salidas dadas durante un periodo de vida de la ejecución del algoritmo. Por ejemplo, un algoritmo que verifica que hay más ceros que unos en una secuencia binaria infinita debe ejecutarse siempre para que pueda devolver un valor útil. Si se implementa correctamente, el valor devuelto por el algoritmo será válido, hasta que evalúe el siguiente dígito binario. De esta forma, mientras evalúa la siguiente secuencia podrán leerse dos tipos de señales: una señal positiva (en el caso de que el número de ceros sea mayor que el de unos) y una negativa en caso contrario. Finalmente, la salida de este algoritmo se define como la devolución de valores exclusivamente positivos si hay más ceros que unos en la secuencia y, en cualquier otro caso, devolverá una mezcla de señales positivas y negativas.

El análisis de algoritmos es una parte importante de la Complejidad Computacional más amplia, que provee estimaciones teóricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Estas estimaciones resultan ser bastante útiles en la búsqueda de algoritmos eficientes.

2. Complejidad Computacional

La **teoría de la complejidad computacional** es la rama de la teoría de la computación que estudia, de manera teórica, la complejidad inherente a la resolución de un problema computable. Los recursos comúnmente estudiados son el tiempo (mediante una aproximación al número y tipo de pasos de ejecución de un algoritmo para resolver un problema) y el espacio (mediante una aproximación a la cantidad de memoria utilizada para resolver un problema). Se pueden estudiar igualmente otros parámetros, tales como el número de procesadores necesarios para resolver el problema en paralelo. La teoría de la complejidad difiere de la teoría de la computabilidad en que ésta se ocupa de la factibilidad de expresar problemas como algoritmos efectivos sin tomar en cuenta los recursos necesarios para ello.

Los problemas que tienen una solución con orden de complejidad lineal son los problemas que se resuelven en un tiempo que se relaciona linealmente con su tamaño.

Hoy en día las computadoras resuelven problemas mediante algoritmos que tienen como máximo una complejidad o coste computacional polinómico, es decir, la relación entre el tamaño del problema y su tiempo de ejecución es polinómica. Éstos son problemas agrupados en la clase P. Los problemas que no pueden ser resueltos por nuestras computadoras (las cuales son Máquinas Determinísticas), que en general poseen costes factorial o combinatorio pero que podrían ser procesados por una máquina no-determinista, están agrupados en la clase NP. Estos problemas no tienen una solución práctica, es decir, una máquina determinística (como una computadora actual) no puede resolverlos en un tiempo razonable.

3. Técnicas de diseño de algoritmos

- **Algoritmos voraces (greedy)**: seleccionan los elementos más prometedores del conjunto de candidatos hasta encontrar una solución. En la mayoría de los casos la solución no es óptima.
- **Divide y vencerás:** dividen el problema en subconjuntos disjuntos obteniendo una solución de cada uno de ellos para después unirlas, logrando así la solución al problema completo.
- **Algoritmos paralelos:** permiten la división de un problema en subproblemas de forma que se puedan ejecutar de forma simultánea en varios procesadores.
- **Algoritmos probabilísticos:** algunos de los pasos de este tipo de algoritmos están en función de valores pseudoaleatorios.
- **Algoritmos determinísticos:** el comportamiento del algoritmo es lineal: cada paso del algoritmo tiene únicamente un paso sucesor y otro antecesor.
- **Algoritmos no determinísticos:** el comportamiento del algoritmo tiene forma de árbol y a cada paso del algoritmo puede bifurcarse a cualquier número de pasos inmediatamente posteriores, además todas las ramas se ejecutan simultáneamente.
- **Metaheurísticas:** encuentran soluciones aproximadas (no óptimas) a problemas basándose en un conocimiento anterior (a veces llamado experiencia) de los mismos.
- **Programación dinámica:** intenta resolver problemas disminuyendo su coste computacional aumentando el coste espacial.
- **Ramificación y acotación:** se basa en la construcción de las soluciones al problema mediante un árbol implícito que se recorre de forma controlada encontrando las mejores soluciones.
- **Vuelta atrás (backtracking):** se construye el espacio de soluciones del problema en un árbol que se examina completamente, almacenando las soluciones menos costosas.

Trataremos a continuación los algoritmos más relevantes de esta clasificación.

3.1. ALGORITMO VORAZ

Un **algoritmo voraz** es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento.

Tal como su nombre indica, el enfoque que aplican es miope, y toman decisiones basándose en la información que tienen disponible de modo inmediato, sin tener en cuenta los efectos que estas decisiones puedan tener en el futuro. Por tanto resultan fáciles de inventar, fáciles de implementar y, cuando funcionan, son eficientes. Sin embargo, como el mundo no suele ser tan sencillo, hay muchos problemas que no se pueden resolver correctamente con un enfoque tan grosero.

Los algoritmos voraces se utilizan típicamente para resolver problemas de optimización, como la búsqueda de la ruta más corta para ir desde un nodo a otro a través de una red de trabajo o la búsqueda del mejor orden para ejecutar un conjunto de tareas en una computadora.

En tales contextos el algoritmo voraz funciona seleccionando el arco, o la tarea que parezca más prometedora en un determinado instante; nunca reconsidera su decisión, sea cual fuere la situación que pudiera surgir más adelante. No hay necesidad de evaluar alternativas, ni de emplear sofisticados procedimientos de seguimiento que permitan deshacer las decisiones anteriores.

Esquema

Dado un conjunto finito de entradas C, un algoritmo voraz devuelve un conjunto S (seleccionados) tal que $S \subseteq C$ y que además cumple con las restricciones del problema inicial. Cada conjunto Sque satisfaga las restricciones se le suele denominar prometedor, y si este además logra que la función objetivo se minimice o maximice (según corresponda) diremos que Ses una solución óptima.

Elementos de los que consta la técnica

- El conjunto C de candidatos, entradas del problema.
- Función solución. Comprueba, en cada paso, si el subconjunto actual de candidatos elegidos forma una solución (no importa si es óptima o no lo es).
- Función de selección. Informa de cuál es el elemento más prometedor para completar la solución. Éste no puede haber sido escogido con anterioridad. Cada elemento es considerado una sola vez. Luego, puede ser rechazado o aceptado y pertenecerá a $C \setminus S$.
- Función de factibilidad. Informa si a partir de un conjunto se puede llegar a una solución. Lo aplicaremos al conjunto de seleccionados unido con el elemento más prometedor.
- Función objetivo. Es aquella que queremos maximizar o minimizar, el núcleo del problema.

Funcionamiento

El algoritmo escoge en cada paso al mejor elemento $x \in C$posible, conocido como el elemento más prometedor. Se elimina ese elemento del conjunto de candidatos ($C \leftarrow C \setminus \{x\}$) y, acto seguido, comprueba si la inclusión de este elemento en el conjunto de elementos seleccionados ($S \cup \{x\}$) produce una solución factible.

En caso de que así sea, se incluye ese elemento en S. Si la inclusión no fuera factible, se descarta el elemento. Iteramos el bucle, comprobando si el conjunto de seleccionados es una solución y, si no es así, pasando al siguiente elemento del conjunto de candidatos.

Ejemplo

Supongamos que disponemos de las siguientes monedas: 1 peso, 50 centavos, 25 centavos, 10 centavos y 5 centavos. Nuestro problema consiste en diseñar un algoritmo que para pagar una cierta cantidad a un cliente, utilice el menor número posible de monedas. Por ejemplo, si tenemos que pagar 3 pesos con 80 centavos, la mejor solución consiste en dar al cliente 3 monedas de 1 peso, una moneda de 50 centavos, una de 25 centavos y una de 5 centavos. Casi todos nosotros resolvemos este tipo de problema todos los días sin pensarlo dos veces, empleando de forma inconsciente un algoritmo evidentemente voraz: empezamos por nada, y en cada fase vamos añadiendo a las monedas que ya estén seleccionadas una moneda de la mayor denominación posible, pero que no debe llevarnos más allá de la cantidad que haya que pagar. El algoritmo se puede formalizar de la siguiente manera:

```
Función devolver cambio(n): conjunto de monedas
{la constante C especifica las monedas disponibles}
Const C = {100, 50, 25, 10, 5}   'se utiliza 100 en lugar de 1 por conveniencia
S ← ϕ {S es un conjunto que contendrá la solución}
s ← 0  {s es la suma de los elementos de S}
mientras s ≠ n hacer
    x  ← el mayor elemento de C tal que s + x ≤ n
    si no existe ese elemento entonces
        devolver "no encuentro la solución"
    S  ← S U {una moneda de valor x}
    s  ← s + x
Devolver S
```

Es fácil convencerse de que los valores dados para las monedas, y disponiendo de un suministro adecuado de cada denominación, este algoritmo siempre produce una solución óptima para nuestro problema. Sin embargo, con una serie de valores diferente, o si el suministro de alguna de las monedas está limitado, el algoritmo voraz puede no funcionar. El algoritmo es voraz porque en cada paso selecciona la mayor de las monedas que pueda encontrar, sin preocuparse por lo correcto de esta decisión a la larga. Además, nunca cambia de opinión: una vez que una moneda se ha incluido en la solución, la tal moneda queda allí para siempre.

Características generales de los algoritmos voraces

Generalmente, los algoritmos voraces y los problemas que éstos resuelven se caracterizan por la mayoría de las propiedades siguiente (o quizá por todas):

Tenemos que resolver algún problema de forma óptima. Para construir la solución de nuestro problema, disponemos de una conjunto (o una lista) de candidatos: las monedas disponibles, las aristas de un grafo que se pueden utilizar para construir una ruta, etc.

A medida que avanza el algoritmo, vamos acumulando dos conjuntos. Una contiene candidatos que ya han sido considerados y seleccionados, mientras que el otro contiene candidatos que han sido considerados y rechazados.

Existe una función que comprueba si un cierto conjunto de candidatos constituye una solución de nuestro problema, ignorando si es o no óptima por el momento. Por ejemplo, ¿suman las monedas seleccionadas la cantidad que hay que pagar?

Hay una segunda función que comprueba si un cierto conjunto de candidatos es factible, esto es, si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución de nuestro problema. Una vez más, no nos preocupa aquí si esto es óptimo o no. Normalmente, se espera que el problema tenga al menos una solución que sea posible obtener empleando candidatos del conjunto que estaba disponible inicialmente.

Hay otra función más, la función de selección, que indica en cualquier momento cuál es el más prometedor de los candidatos restantes que no han sido seleccionados ni rechazados.

Por último, existe una función objetivo que da el valor de la solución que hemos hallado: el número de monedas utilizadas para dar el vuelto o cualquier otro valor que estemos intentando optimizar. A diferencia de las tres funciones mencionadas anteriormente, la función objetiva no aparece explícitamente en el algoritmo voraz.

Para resolver nuestro problema, buscamos un conjunto de candidatos que constituya una solución, y que optimice (minimice o maximice, según los casos) el valor de la función objetivo. Los algoritmos voraces avanzan paso a paso. Inicialmente, el conjunto de elementos seleccionados está vacío. Entonces, en cada paso se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra elección por la función de selección. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible, rechazamos el candidato que estamos

considerando en ese momento. Sin embargo, si el conjunto aumentado sigue siendo factible, entonces añadimos el candidato actual al conjunto de candidatos seleccionados, en donde pasará a estar desde ahora en adelante. Cada vez que se amplía el conjunto de candidatos seleccionados, comprobamos si éste constituye ahora una solución para nuestro problema. Cuando el algoritmo voraz funciona correctamente, la primera solución que se encuentre de esta manera es siempre óptima:

```
Función voraz(C: conjunto): conjunto
    {C es el conjunto de candidatos}
    S ← ϕ {Construimos la solución en el conjunto S}
    mientras C ≠ ϕ y no solución(s) hacer
        x ← seleccionar(C)
        C ← C \ {x}
        Si factible(S U {x} entonces S ← S U {x}
    Si solución(S) entonces devolver S
        Sino devolver "no hay soluciones"
```

Está clara la razón por la cual tales algoritmos se denominan voraces, en cada paso, el procedimiento selecciona el mejor bocado que pueda tragar, sin preocuparse por el futuro. Nunca cambia de opinión: una vez que un candidato se ha incluido en la solución, queda allí para siempre, una vez que se excluye un candidato de la solución, nunca vuelve a ser considerado.

La función de selección suele estar relacionada con la función objetivo. Por ejemplo, si estamos intentando maximizar nuestros beneficios, es probable que seleccionemos aquel candidato restante que posea un mayor valor individual. Si intentamos minimizar el coste, entonces quizá seleccionemos el más barato de los candidatos disponibles, y así sucesivamente. Sin embargo, veremos que en algunas ocasiones puede haber varias funciones de selección plausibles, así que hay que seleccionar la adecuada si deseamos que nuestro algoritmo funcione correctamente.

Volviendo por un momento al ejemplo de dar cambio, lo que sigue es una forma de adecuar las características generales de los algoritmos voraces a las características particulares de este problema:

- Los candidatos son un conjunto de monedas, que representan en nuestro ejemplo 1 peso, 50, 25, 10 y 5 centavos, con tantas monedas de cada valor que nunca las agotamos (sin embargo el conjunto de candidatos debe ser finito).
- La función de solución comprueba si el valor de las monedas seleccionadas hasta el momento es exactamente el valor que hay que pagar.
- Un conjunto de monedas será factible si su valor total no sobrepasa la cantidad que haya que pagar.
- La función de selección toma la moneda de valor más alto que quede en el conjunto de candidatos.
- La función objetivo cuenta el número de monedas utilizadas en la solución.

Está claro que es más eficiente rechazar todas las monedas restantes de 1 peso cuando el valor restante que hay que pagar car por debajo de ese valor. El uso de la división entera para calcular cuántas monedas de un cierto valor hay que tomar también es más eficiente que actuar por sustracciones sucesivas. Si se adopta cualquiera de estas tácticas, entonces podemos relajar la condición consistente en que el conjunto de monedas debe ser finito.

3.2. DIVIDE Y VENCERÁS

Esta técnica (como cualquier otra de diseño de algoritmos) no se puede aplicar a todo tipo de problemas. En algunos casos se puede aplicar pero no da lugar a soluciones óptimas y en otros casos sí, no existiendo una clasificación de problemas en los que conviene aplicarla, por lo que no intentamos dar una clasificación de problemas que se resuelvan por una técnica u otra, sino unas técnicas de las más usadas, que hay que conocer para poder decidir ante un problema concreto si consideramos conveniente aplicar una determinada técnica, una mezcla de varias o ninguna de ellas.

La técnica divide y vencerás consiste en intentar resolver un problema dividiéndolo en k subproblemas que se resuelvan más fácilmente, resolverlos, y combinar los resultados de los subproblemas para obtener una solución del problema original.

En la cultura popular, divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas.

En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (quicksort, mergesort, entre muchos otros), multiplicar números grandes (Karatsuba), análisis sintácticos (análisis sintáctico top-down) y la transformada discreta de Fourier.

Por otra parte, analizar y diseñar algoritmos de DyV son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y no hay un método sistemático de generalización.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada (o su equivalente en computación numérica, el algoritmo de bisección para búsqueda de raíces). Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de "divide y vencerás"; en particular, si es usando una serie de recursiones que lo convierten en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de "divide y vencerás". El nombre decrementa y vencerás ha sido propuesta para la subclase simple de problemas.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n/k y donde $0 \leq n/k < n$. A esta tarea se le conoce como división.
2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3. Por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Ejemplo. Multiplicación de enteros de n cifras

Algoritmo clásico

1234*5678 = 1234* (5*1000 + 6*100+7*10+8)
= 1234*5*1000 + 1234*6*100 + 1234*7*10 + 1234*8

Operaciones básicas:

- Multiplicaciones de dígitos
- Sumas de dígitos
- Desplazamientos

1234 = 12*100 + 34
5678 = 56*100 + 78
1234*5678 = (12*100 + 34)*(56*100 + 78)
= 12*56*10000 + (12*78+34*56)*100 + (34*78)
Idea: Se reduce una multiplicación de 4 cifras a cuatro multiplicaciones de 2 cifras, más tres sumas y varios desplazamientos.

1. Dividir

X = 12345678 =	xi * 10^4 + xd	xi=1234	xd=5678
Y = 24680135 =	yi * 10^4 + yd	yi=2468	yd=0135

2. Combinar

X*Y = (xi*10^4 + xd) * (yi*10^4 + yd)

= xi*yi*10^8 + (xi*yd + xd*yi) * 10^4 + xd * yd

En general:
X = xi*$10^{n/2}$ + xd
Y = yi*$10^{n/2}$ + yd
X*Y = (xi*$10^{n/2}$+ xd) * (yi*$10^{n/2}$+ yd)

= xi*yi*10^n + (xi*yd+xd*yi)* $10^{n/2}$+ xd*yd

La técnica “divide y vencerás” (DV) consiste en:

- Descomponer el problema que hay que resolver en cierto número de subproblemas más pequeños del mismo tipo.
- Resolver de forma sucesiva e independiente todos estos subproblemas.
- Combinar las soluciones obtenidas para obtener la solución del problema original.

Características de los problemas resolubles utilizando “divide y vencerás”

- El problema se puede descomponer en otros del mismo tipo que el original y de tamaño más pequeño (formulación recursiva).
- Los subproblemas pueden resolverse de manera independiente.
- Los subproblemas son disjuntos, sin solapamiento.
- La solución final se puede expresar como combinación de las soluciones de los subproblemas.