Apuntes de EDA

Raúl Gilabert Gámez

26 de octubre de 2022

Índice general

Capítulo 1

Análisis de algoritmos

Un algoritmo es un proceso computacional bien definido que toma un valor o conjunto de valores y produce un valor o conjunto de valores, es decir, convierte una entrada en una salida.

El análisis de un algoritmo se basa en sacar el tiempo de ejecución del algoritmo en base al tamaño de la entrada de este. El tamaño de la entrada depende del problema que se esté tratando, pero en la gran mayoría de problemas se suele usar como medida el número de elementos de la entrada, como puede ser el tamaño de un array. El tiempo de ejecución de un algorimo suele ser la cantidad de pasos que tiene que ejecutar el programa para terminarlo.

Para analizar los algoritmos siempre se tendrá en cuenta el peor caso de ejecución de estem ya que eso permite que se asegure que el algoritmo nunca tardará más que el definido por este cálculo. Además, el peor caso suele ocurrir con frecuencia, como puede ser el caso de una búsqueda de un elemento inexistente en la estructura de datos.

Por ejemplo, en un Insertion Sort, el mejor caso se puede expresar con $an + b$, siendo a y b constantes del algoritmo y n el tamaño de la entrada, siendo este caso la situación en la que la estructura de datos introducida ya esté ordenada correctamente. En cambio, el peor caso, se puede expresar con $an^2 + bn + c$, siendo a, b y c constantes del algoritmo y n el tamaño de la entrada, siendo en esta situación el caso de que la entrada esté ordenada de forma decreciente.

1.1. Factor de crecimiento

Si ya en el ejemplo anterior se ignoró el coste actual de cada instrucción usando constantes para estas, en el análisis de algoritmos en realidad además de ignorar eso se ignoran las demás constantes, dejando solo el factor de crecimiento (n). Además, como ya se ha dicho antes, se usa únicamente el peor caso, quedando que el Insertion Sort tiene un coste de $\Theta(n^2)$.

De esta manera se considera que un algoritmo es más eficiente que otro si el factor de crecimiento es menor. Aun así por el tema de las constantes que se ignoran puede ser que un algoritmo con un factor de crecimiento mayor tarde menos con entradas pequeñas, lo que provoca que a veces, en el caso de los algoritmos de ordenación, dependiendo del tamaño de la entrada a ordenar sea preferible usar un algoritmo menos eficiente que otro.

1.2. Orden de magnitud

El orden de magnitud es necesario para poder tener una notación que permita expresar el peor caso de ejecución del algoritmo y que sea independiente de constantes multiplicativas.

Está notación es la llamada O grande, de forma que teniendo una función f, $O(f)$ representa la clase de funciones que "crecen como f o más lentamente".

Formalmente $g \in O(f)$ si existen c > 0 y $n_0 \in \mathbb{N}$ tales que $\forall_{n \geq n_0} g(n) \leq c \cdot f(n)$

1.3. Notación asintótica

Este tipo de notación permite la clasificación de las funciones en base al crecimiento de estas "a la larga". De esta forma se centra en el comportamiento de las funciones para entradas largas.

1.3.1. Notación O

La notación O se usa cuando se puede acotar f(n) por encima, es decir: $f(n) \in O(g(n))$ si existe una constante c de forma que $f(n) < c \cdot g(n)$ para una n lo suficientemente grande, es decir, $n > n_0$.

1.3.2. Notación Ω

De forma similar a la notación O, la notación Ω acota la función f(n) solo por un lado, en este caso, la acota por debajo, es decir, que $f(n) \in \Omega(g(n))$ si existe una constante c de forma que $c \cdot g(n) < f(n)$ para una n lo suficientemente grande, es decir, para $n > n_0$.

1.3.3. Notación Θ

Esta notación se usa para denotar que $f(n) \in \Theta(g(n))$ si existen dos constantes c_1 y c_2 de forma que $c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)$ para una n lo suficientemente grande, es decir, para $n > n_0$. Aunque $\Theta(g(n))$ es un conjunto y lo correcto es hace $f(n) \in \Theta(g(n))$ vamos a usar también $f(n) = \Theta(g(n))$.

En base a lo visto en estos puntos, se puede ver que $f(n) \in \Theta(g(n)) \iff f(n) \in O(n) \wedge f(n) \in \Omega(g(n))$

1.3.4. Propiedades

- $f(n) \in O(f(n))$
  $f(n) \in \Theta(f(n))$
  $f(n) \in \Omega(f(n))$
- $h(n) \in O(g(n)) \wedge g(n) \in O(f(n)) \Rightarrow h(n) \in O(f(n))$
  $h(n) \in \Theta(g(n)) \wedge g(n) \in \Theta(f(n)) \Rightarrow h(n) \in \Theta(g(n))$
  $h(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(f(n)) \Rightarrow h(n) \in \Omega(g(n))$
- $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$
  $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$
  $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
- $g(n) \in O(f(n)) \iff O(g(n)) \subseteq O(f(n))$

- $g_1(n) \in O(f_1(n)) \wedge g_2(n) \in O(f_2(n)) \Rightarrow g_1(n)+g_2(n) \in O(f_1(n)+f_2(n)) = O(max(f_1(n), f_2(n)))$
- $g_1 \in O(f_1) \wedge g_2 \in O(f_2) \Rightarrow g_1 \cdot g_2 \in O(f_1 \cdot f_2)$

1.3.5. Formas de crecimiento

- Constante $\Theta(1)$ Número par o impar.
- Logarítmico $\Theta(\log n)$ Búsqueda binaria.
- Radical $\Theta(\sqrt{n})$ Test básico de primalidad.
- Lineal $\Theta(n)$ Búsqueda secuencial en un vector.
- Quasilineal $\Theta(n \log n)$ Ordenación eficiente de un vector.
- Cuadrático $\Theta(n^2)$ Suma de dos matrices cuadradas de tamaño $n \times n$.
- Cúbico $\Theta(n^3)$ Multiplicación de dos matrices cuadradas de tamaño $n \times n$.
- Polinómico $\Theta(n^k)$ Para $k \geq 1$ constante, combinaciones de n elementos cogidos de k en k.
- Exponencial $\Theta(k^n)$ Para k constante, búsqueda en un espacio de configuraciones de anchura k y altura n.

1.4. Coste de los algoritmos

Una operación elemental (asignación de tipo básico, incremento o decremente de alguna variable de tipo básico, operaciones aritméticas, lectura o escritura de tipo básico, comparación, acceso a componentes de un vector, etc.) tiene coste $\Theta(1)$. Sabiendo esto, evaluar una expresión tiene un coste igual a la suma de los constes de las operaciones que se realizan. Además el coste de una operación `return E` es el de analizar E y copiar el resultado. Copiar un vector de tamaño n tiene un conste $\Theta(n)$. El paso por referencia de parámetros tiene un coste $\Theta(1)$, a diferencia del pasos de parámetros copiando que el coste depende del tipo de parámetro que es.

1.4.1. Algoritmos no recursivos

Si el coste de un fragmento es F_1 es C_1 y el de un fragmento f_2 es C_2 entonces el coste de ejecutar una secuencia con los dos parámetros es de $C_1 + C_2$.

Si el coste de F_1 es C_1 y de F_2 es C_2 y de B es D, entonces evaluar `if (B) F1; else F2` es $D + C_1$ en caso de que B sea cierto y $D + C_2$ en caso de que B sea falso. De forma que el coste en el peor caso es $D + max(C_1, C_2)$.

Si el coste de F durante la k-ésima iteración es C_k, el de evaluar B es D_k y el número de iteraciones es N entonces el coste de `while(B) F;` es $(\sum_{k=1}^{N} C_k + D_k) + D_{n+1}$ (N veces la evaluación de la condición y la ejecución del contenido y una evaluación final que no entra en el bucle).

1.4.2. Algoritmos recursivos

El coste de un algoritmo recursivo se expresa en forma de recurrencia, es decir, una ecuación o inecuación que describe una función expresada en términos de su valor para entradas más pequeñas.

Para encontrar la recurrencia que describe el coste de un algoritmo recursivo se han de determinal el parámetro de recursión n (normalmente el tamaño de la entrada) y el coste del caso inductivo (número de llamadas recursivas, valores del parámetro recursivo de las llamadas y coste de los cálculos extra no recursivos). De esta manera se ve que hay dos tipos de recurrencia en base a su coste:

- Sustractiva

  Siendo del tipo

$$T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n), & \text{si } n \geq n_0 \end{cases}$$

- Divisora

  Siendo del tipo

$$T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{si } n \geq n_0 \end{cases}$$

Capítulo 2

Divide y vencerás

La estrategía de divide y vencerás para resolver problemas mediante algoritmos se basa tres pasos:

1. Dividir el problema en subproblemas, es decir, casos más pequeños del mismo problema
2. Resolver los subproblemas recursivamente
3. Combinar las respuestas de forma adecuada

La cosa es que la llamada recursiva solo se produce si el subproblema es demasiado grande (en un algoritmo de ordenación si hay muchos elementos para ordenar) pero en caso que no, se entra en el caso base (en un algoritmo de ordenación, usar uno más eficiente para pocos elementos).

Aunque lo parezca, los subproblemas no tienen por qué ser de una entrada que sea una fracción de la entrada original como puede ser en el caso de un algoritmo de recorrido de una estructura de datos elemento a elemento, de forma que la entrada recursiva sea de un tamaño inferior en 1 a la entrada del que hace la llamada.

2.1. Algoritmos

2.1.1. Merge sort

El merge sort es un algoritmo que muestra muy bien el funcionamiento de la estrategia de divide y vencerás para hacer algoritmos. El merge sort se basa en dividir en dos mitades la entrada y ejecutar el mismo algoritmo sobre las dos divisiones. Al recibir la respuesta de estas dos nuevas ordenaciones va ordenando en una nueva estructura de datos en base comparar el primer elemento de cada división antes hecha y ordenada y añadiendo siempre el menor de estos.

Hay una variante más eficiente en el tiempo que realiza el merge sort hasta un cierto tamaño de entrada que hace un insert sort que es más eficiente a partir de ese tamaño, de forma que se reduce el tiempo de ejecución del algoritmo.

2.1.2. Quick sort

El algoritmo de ordenación rápida (quick sort) es el algoritmo de ordenación genérico más rápido. Aunque el peor caso sea $\Theta(n^2)$, el caso promedio es $\Theta(n \log n)$ y la eficiencia del bucle interno hace que sea el mejor algoritmo de ordenación en la práctica.

El funcionamiento del algoritmo es el siguiente para un vector t de mínimo de 2 elementos:

1. Escoge un elemento x de T
2. divide T en en dos grupos diferentes entre sí T_1 y T_2 de forma que se cuempla lo siguiente:
   - T_1 contiene elementos $\leq x$ de T
   - T_2 contiene elementos $\geq x$ de T
3. Ordena T_1 y T_2 de forma recursiva
4. Devuelve T_1 seguido de T_2

De esta manera se ve que el merge sort y el quick sort siguen la estrategia de divide y vencerás pero la llevan a cabo de formas distintas ya que el merge sort divide en subproblemas de forma directa y después hace una fusión de los vectores haciendo las comparaciones y en cambio, el quick sort produce las divisiones con comparaciones y después hace la fusión de elementos de forma directa.

El problema que genera el quick sort es saber qué elemento x hay que escoger del vector ya que puede producirse que se hagan subdivisiones del problema que sean innecesarias si se usara un criterio distinto para escoger el elemento pivot. El algoritmo original tomaba el primer elemento como pivot pero esto puede ser ineficiente dependiendo del valor que tenga y el del resto de elementos.

De esta manera se crean 3 estrategias distintas que se pueden seguir para escoger el elemento pivot:

- Escoger el primer elemento Esto es aceptable si la entrada es totalmente aleatoria pero en casos donde la entrada está totalmente ordenada el cualquier orden el algoritmo tiene un coste $\Theta(n^2)$ y no hace ningún cambio en la entrada.
- Escoger un elemento aleatorio De media suele dividir el problema en subproblemas similares pero no siempre hace el algoritmo más rápido debido al coste de la generación de números aleatorios
- Escoger la mediana de los elementos Es la mejor opción pero al ser demasiado cara se hace una alternativa que produce un número que en muchos casos es una buena estimación de esta. Esta alternativa es hacer la mediana de 3 elementos, siendo estos el primero, el final y el central.

Al igual que pasa con el merge sort que para vectores muy pequeños es preferible usar un insert sort al ser este más eficiente en vectores pequeños, en este caso cuando el vector es de un tamaño entre 5 y 20 elementos.

2.1.3. Algoritmo de Karatsuba

El algoritmo de multiplicación que se estudia a nivel escolar tiene una eficiencia de $\Theta(n^2)$ siendo n el número de dígitos de ambos números pero se encontró un algoritmo con un coste $\Theta(n^{\log_2 3})$, siendo el algoritmo el siguiente:

Suponiendo que x e y son dos naturales de n bits siendo n par se dividen dividen tanto x como y en dos partes, como puede ser en este ejemplo:

$$x = 10010111_2, y = 11001010_2$$
$$x = 2^{n/2}x_I + x_D \Rightarrow x_I = 1001_2, x_D = 0111_2$$
$$y = 2^{n/2}y_I + y_D \Rightarrow y_I = 1100_2, y_D = 1010_2$$

De esta manera, el producto

$$xy = (2^{n/2}x_I + x_D)(2^{n_2}y_I + y_D)$$

Se puede reescribir como

$$xy = 2^n x_I y_I + 2^{n/2}(x_I y_D + x_D y_I) + x_D y_D$$

De esta manera, un algoritmo que basado en esta expresión calculara recursivamente los productos tendría un coste $T(n) = 4T(n/2) + \Theta(n)$.

Teniendo en cuenta que

$$x_I y_D + x_D y_I = (x_I + x_D)(y_I + y_D) - x_I y_I - x_D y_D$$

Tomamos

$$a = x_I y_I, b = x_D y_D, c = (x_I + x_D)(y_I + y_D)$$

Entonces la ecuación

$$xy = 2^n x_I y_I + 2^{n/2}(x_I y_D + x_D y_I) + x_D y_D$$

Se puede reescribir de esta manera

$$2^n a + 2^{n/2}(c - a - b) + b$$

De esta manera logramos que la multiplicación dependa solo de 3 subproductos, de forma que se da lugar a un algoritmo con coste $T(n) = 3T(n/2) + \Theta(n)$ y por el teorema de recurrencias divisoras sabemos que $T(n) \in \Theta(n^{log_2 3}) \approx \Theta(n^{1,585})$.

2.1.4. Exponenciación rápida

El algoritmo iterativo evidente para calcular x^n haría $\Theta(n-1) = \Theta(n)$ multiplicaciones.

Con un enfoque recursivo vemos que en caso de que n sea par tenemos que

$$x^n = (x^{n/2})^2$$

y si n es impar tenemos que

$$x^n = x^{n-1} \cdot x = (x^{(n-1)/2})^2 \cdot x$$

y el caso base que dice que

$$x^0 = 1$$

De esta manera, siguiendo estos casos se puede hacer un algoritmo con un coste definido por la recurrencia $T(n) = T(n/2) + \Theta(1)$, de forma que por el teorema maestro de recurrencias divisorias esto implica que $T(n) \in \Theta(\log n)$.

2.1.5. Algoritmo de Strassen

Tal y como se sabe, el producto de dos matrices X e Y de tamaño $x \times y$ es una matriz Z de tamaño $n \times n$ de forma que

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

Es decir, Z_{ij} es el producto de la fila i-ésima de X por la columna j-ésima de Y.

Este algoritmo tiene un coste de $\Theta(n^3)$ pero hay una alternativa con coste $\Theta/n^{\log_2 7} \approx \Theta(n^{2,81})$.

Una primera idea que se puede tener es que el producto de matrices se puede hacer por bloques, de forma que dividiendo X e Y en cuatro cuadrantes cada uno nos queda que

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

De esta manera se puede ver que

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Y estos productos de matrices se pueden calcular de forma recursiva.

De esta manera se ve que el coste de $T(n)$ es la suma de hacer 8 productos de matrices de tamaño $n/2$: 8T(n/2) y 4 sumas de matrices de tamaño $n/2$: $\Theta(n^2)$. Por tanto tenemos la recurrencia $T(n) = 8T(n/2) + \Theta(n^2)$ que por el teorema de recurrencias divisoras tenemos que $T(n) \in \Theta(n^{log_2 8}) = \Theta(n^3)$. pero la cuestión es que el número de productos se puede reducir a 7 de forma que

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

donde

$$P_1 = A(F - H), P_2 = (A + B)H,$$
$$P_3 = (C + D)E, P_4 = D(G - E),$$
$$P_5 = (A + D)(E + H), P_6 = (B - D)(G + H),$$
$$P_7 = (A - C)(E + F)$$

De forma que nos queda

$$P_5 + P_4 - P_2 + P_6 = (A + D)(E + H) + D(G - E) - (A + B)H + (B - D)(G + H) =$$

$$AE + AH + DE + DH + DG - DE - HA - BH + BG - BH - DG - DH = AE + BG$$

$$P_1 + P_2 = A(F - H) + (A + B)H = AF - AH + AH + BH = AF + BH$$

$$P_3 + P_4 = (C + D)E + D(G - E) = CE + DE + DG - DE = CE + DG$$

$$P_1 + P_5 - P_3 - P_7 = A(F - H) + (A + D)(E + H) - (C + D)E - (A - C)(E + F) =$$

$$AF - AH + AE + AH + DE + DH - CE - DE - AE - AF + CE + CF = DH + CF$$

De forma que se cumplen las igualdades entre esta acumulación de productos sumados y restados entre sí y el contenido de las matrices en cuadrantes. De esta forma siguiendo este algoritmo que tiene solo 7 productos acabamos con un costo $T(n) = 7T(n/2) + \Theta(n^2)$ que por el teorema maestro de recurrencias divisoras tenemos que $T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2,81})$.