Programación II

Tecnicatura Universitaria en Inteligencia Artificial

2023

Diseño de algoritmos

1. Resolución de Problemas

A lo largo de esta materia hemos discutido distintos conceptos relacionados a la algoritmia: desde ideas fundamentales como la recursión o el paradigma orientado a objetos, hasta estructuras de datos avanzadas como tablas hash. Sin embargo, no se ha dicho aún mucho sobre la *resolución de problemas* en general, una actividad diaria en la vida laboral de un programador profesional. La resolución de problemas implica buscar una técnica que permita hallar la solución al mismo.

En este apunte se presentan tres métodos algorítmicos generales que un diseñador de algoritmos puede emplear para encontrar la solución a un problema. Es importante notar, sin embargo, que no hay buenas recetas para idear recetas. Cada problema algorítmico es un desafío para el diseñador. Algunos problemas pueden ser sencillos, mientras que otros muy complicados. Incluso, un mismo problema puede admitir distintas soluciones, implicando que no hay un único camino o forma de resolverlo.

Este apunte muestra únicamente que ciertos algoritmos siguen bastante bien ciertos paradigmas generales. Como moraleja, el diseñador de algoritmos puede beneficiarse de estudiar primero las técnicas, intentando determinar si pueden ser utilizadas, o adaptadas para su uso, en la situación que esté queriendo resolver. En general, sin embargo, el diseño de algoritmos es una actividad creativa que requiere ingenio, pero que sin duda puede hacerse más sencilla y divertida dominando las técnicas y métodos disponibles. Por otro lado, gran parte del aprendizaje proviene del hacer, del sumergirse en el problema para entenderlo y así hallar una solución. La práctica y el entrenamiento es lo que nos dará agilidad a la hora de encontrar una solución.

2. Divide y vencerás

2.1. Introducción

El término Divide y Vencerás[1] en su acepción más amplia es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.

En nuestro contexto, Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Si los subproblemas

[1] O Divide & Conquer (DyC) por su nombre en en inglés.

son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

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 $0 \leq n_k \leq n$. 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. Combinar las soluciones de los subproblemas más pequeños para obtener la solución general.

2.2. Forma general de resolución

El funcionamiento de los algoritmos que siguen la técnica de Divide y Vencerás descrita anteriormente se refleja en el esquema general que presentamos a continuación:

```
def algoritmo_divide_y_venceras(x: TipoProblema) -> TipoSolucion:
    if es_caso_base(x):
        return resolver_caso_base(x)

    subproblema1, subproblema2 = dividir(x)
    solucion1 = algoritmo_divide_y_venceras(subproblema1)
    solucion2 = algoritmo_divide_y_venceras(subproblema2)
    solucion = combinar(subproblema1, subproblema2)
    return solucion
```

Nota El número k de subproblemas en el que dividiremos el problema original puede depender del problema, pero debe ser pequeño e independiente de una entrada determinada. Los algoritmos Divide y Vencerás que contienen sólo una llamada recursiva, es decir $k = 1$, son un caso particular, muchas veces llamados algoritmos **Decrease and Conquer**[2]. Un ejemplo de algoritmo de este tipo es el algoritmo recursivo para calcular el factorial de un número que vimos al inicio de la materia, donde sencillamente se reduce el problema a otro subproblema del mismo tipo de tamaño más pequeño. La ventaja de los algoritmos de simplificación es que consiguen reducir el tamaño del problema en cada paso, por lo que sus tiempos de ejecución suelen ser muy buenos (normalmente de orden logarítmico o lineal).

Entonces para resolver un problema mediante la técnica Divide y Vencerás necesitamos identificar las siguientes funciones.

- Una función `es_caso_base` que verifique si estamos en un caso base.
- Una función `resolver_caso_base` que resuelve los casos base.
- Una función `dividir` que nos indique cómo dividir el problema en varios subproblemas relacionados, pero de menor tamaño.
- Una función `combinar` que nos permita juntar las soluciones parciales del caso anterior para obtener la solución final.

[2] O "Disminuye y vencerás.en español. También llamadas .algoritmos de simplificación".

2.3. Ejemplo

Supongamos que tenemos una lista no ordenada de vuelos hacia distintos destinos y sus respectivos precios. El objetivo es elegir el pasaje más económico. Claramente podríamos simplemente recorrer la lista, comparando cada elemento con el mínimo hasta el momento, y actualizando dicho valor si el vuelo considerado tiene un precio menor. Sin embargo, también podemos utilizar la estrategia DyC para resolver este problema, como se muestra en la Figura 1 presentada a continuación.

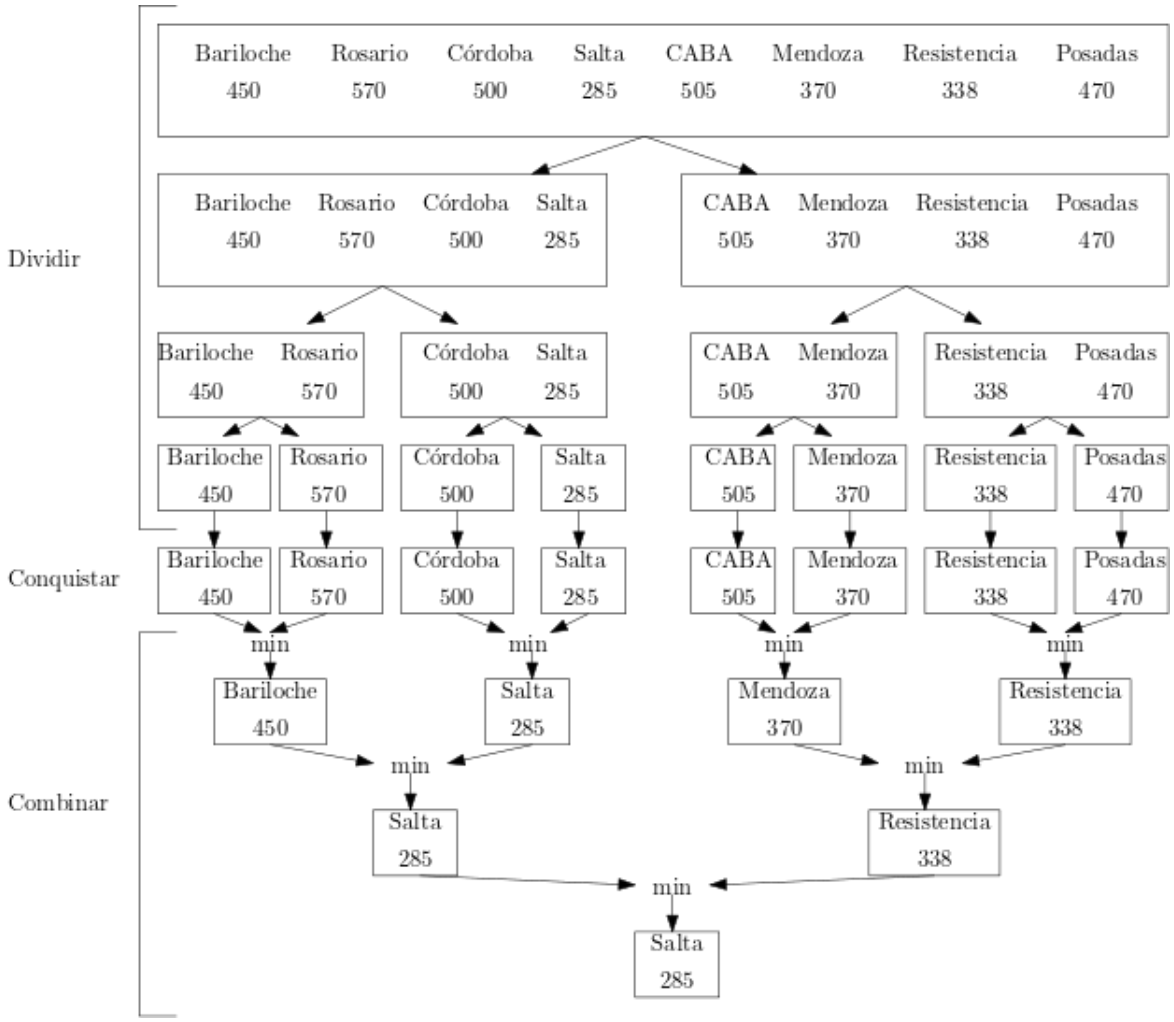

Figura 1: Cálculo del elemento mínimo de una lista desordenada.

La estrategia para encontrar el mínimo elemento ilustrada en la figura es la siguiente:

1. Dividimos recusirvamente la lista en dos mitades, hasta obtener listas de un único elemento.

2. Resolvemos el problema para listas de un único elemento. En este caso, calcular el mínimo de una lista de un único elemento es trivial.

3. Combinamos los resultados obtenidos, aplicando la función `min` en cada paso.

2.3.1. Implementación

Para implementar la solución propuesta, debemos definir las funciones mencionadas en la sección 2.2. Recordemos que el problema es encontrar el elemento mínimo en una lista no ordenada. Por lo tanto, el objetivo final será tener una función `minimo` que, dada una lista de números, retorne el elemento mínimo.

En este problema, el caso base es la lista de un único elemento. Por consiguiente, la función `es_caso_base` debe determinar si la lista contiene un solo elemento.

```
def es_caso_base(l):
    return len(l) == 1
```

Como ya se mencionó, en este problema la resolución del caso base es trivial, ya que se reduce a determinar el elemento mínimo en una lista que contiene un solo elemento. Por ello, la función `resolver_caso_base` se limita a retornar el único elemento de la lista.

```
def resolver_caso_base(l):
    return l[0]
```

La función `dividir` se encarga de dividir adecuadamente el problema en sub-problemas más pequeños pero de la misma clase. En este caso, debe ser capaz de dividir una lista por la mitad, retornando dos listas. Notemos que, si la cantidad de elementos en la lista original es impar, la función dividirá la lista en dos listas `l1` y `l2`, en donde `len(l1)+1 = len(l2)`.

```
def dividir(l):
    middle = math.floor(len(l)/2)
    return (l[0:middle], l[middle:len(l)])
```

Una vez dividido el problema en sub-problemas, y resueltos estos sub-problemas, debemos combinar dichas soluciones para obtener la solución final. En nuestro caso, la función `combinar` debe encargarse de combinar los resultados parciales calculando el elemento mínimo en cada caso. Es decir, tomará dos números, y simplemente retornará el mínimo de ellos.

```
def combinar(l1, l2):
    return min(l1[0], l2[0])
```

Por último, nos queda construir la definición de la función `minimo` que toma una lista y retorna el elemento mínimo de ella, utilizando las definiciones dadas previamente.

```
def minimo(l):
    if es_caso_base(l):
        return resolver_caso_base(l)
    l1, l2 = dividir(l)
    return combinar(minimo(l1), minimo(l2))
```

2.4. Ventajas y desventajas

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

- Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad del código y facilidad para detectar y corregir errores.

- Sin embargo, en un modelo secuencial, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos. Sin embargo, si contamos con una computadora con más de un procesador podemos **paralelizar** la resolución de los subproblemas, acortando así el tiempo de ejecución.

- Además, las repetidas llamadas a función ocupan espacio en la memoria de trabajo, ya que se deben guardar sus parametros y variables locales por separado para cada iteración.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Un claro ejemplo de esto es el algoritmo que utilizamos para calcular los números de Fibonacci en la unidad sobre Recursión. Para aquellos problemas en los que la solución haya de construirse a partir de las soluciones de subproblemas entre los que se produzca necesariamente solapamiento existe otra técnica de diseño más apropiada, y que permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos: la **Programación Dinámica**. Sin embargo, no estudiaremos dicha técnica en este curso.

Otra consideración importante a la hora de diseñar algoritmos Divide y Vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible. Si vamos a dividir un problema de tamaño n en dos, por ejemplo, no obtendremos un algoritmo eficiente si lo dividimos en un subproblema de tamaño 2 y un subproblema de tamaño $n-2$: esta división es injusta dado que casi todo el trabajo se realizará al resolver uno solo de los subproblemas, dando algoritmos poco eficientes.

2.5. Búsqueda binaria

El algoritmo de búsqueda binaria es un ejemplo claro de la técnica Divide y Vencerás. El problema de partida es decidir si existe un elemento dado x en una lista de enteros **ordenada**. El hecho de que esté ordenada va a permitir utilizar esta técnica, pues podemos plantear un algoritmo con la siguiente estrategia:

1. Comparamos x con el elemento que ocupa la posición media de la lista.

2. Si la comparación es verdadera, hemos terminado, podemos concluir que x se encuentra en la lista.

3. Si la comparación es falsa, pueden darse dos situaciones: o bien x es menor a este elemento, o bien es mayor a este elemento. En cualquiera de los dos casos, podemos descartar la mitad de la lista y continuar buscando únicamente con la otra mitad.

4. Aplicamos recursivamente el algoritmo.

En este ejemplo la división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de "simplificación".

¿Cuál es el caso base de esta recursión? Claramente, si la lista esta vacía, no contiene ningún elemento, luego, de seguro no contiene a x. Por otro lado, si una lista contiene un solo elemento, podemos decidir inmediatamente si se trata de x o no.

Ejercicio. Implementar la búsqueda binaria como un algoritmo Divide y Vencerás.

3. Ejercicios

1. Juan esta desarrollando una aplicación utilizando búsqueda binaria, donde la velocidad de ejecución es crítica, entonces se lo ocurrió lo siguiente: ¿porque repartir el trabajo en solo dos subproblemas? Entonces invento un algoritmo, al que bautizó Búsqueda Ternaria, que se comporta del siguiente modo:

   Recibe como entrada una lista ordenada L de n números enteros y luego:

   Primero, compara x con el elemento que ocupa la posición $n//3$.

Si $n == L[n//3]$, hemos terminado.

Caso contrario si $n < L[n//3]$, llama recursivamente con la lista $L[0 : n//3]$.

Si $n > L[n//3]$, compara n con el elemento que ocupa la posición $(2n//3)$.

Si $n == L[2n//3]$ hemos terminado.

Caso contrario, si $n < L[2n//3]$, llama recursivamente con la lista $L[n//3 : 2n//3]$.

Si $n > L[n//3]$ llama recursivamente con la lista $L[2n//3 : n]$

¿Ha conseguido alguna mejora de perfomance?