Técnicas Algorítmicas

Prof. Agustín Gravano

MiM - UTDT - Segundo semestre de 2020

Clase 2: Dividir y Conquistar

Ejemplo: Sumar elementos de una lista

Algoritmo recursivo lineal, con inmersión de parámetros para evitar hacer `L[1:]` en cada llamado recursivo.

```
def sumar_aux(L, desde):
  if desde >= len(L):
    res = 0
  else:
    res = L[desde] + sumar_aux(L, desde+1)
  return res

def sumar(L):
  return sumar_aux(L, 0)
```

Observación: Este algoritmo ya tiene complejidad óptima. Cualquier algoritmo debe visitar al menos una vez cada elemento de la lista para poder sumarlo. Entonces, cualquier algoritmo que resuelva este problema es, como mínimo, lineal.

Complejidad de `sumar_aux`

```
def sumar_aux(L, desde):
  if desde >= len(L):
    res = 0
  else:
    res = L[desde] + sumar_aux(L, desde+1)
  return res
```

Llamamos $T(n)$ a la cantidad de operaciones básicas que se ejecutan, en función de una entrada de tamaño n.

En este caso, n es la cantidad de elementos que falta sumar.

Queremos ver que $T(n) \in O(n)$ (que crece linealmente con n).

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) && = T(n-1) + k \\
& && (k\text{: cantidad constante de operaciones}) \\
&= (T(n-2) + k) + k && = T(n-2) + 2 \cdot k \\
&= (T(n-3) + k) + 2 \cdot k && = T(n-3) + 3 \cdot k \\
&\ldots \\
&= T(0) + n \cdot k \\
&= O(1) + O(n) \ = \ O(n)
\end{aligned}
$$

Por lo tanto, el algoritmo recursivo `sumar_aux` tiene $O(n)$.

Dividir y Conquistar

- Táctica político-militar de dudoso origen, frecuentemente atribuida a Julio César.
- Consiste en dividir al enemigo, de modo que cada una de las partes sea más fácil de derrotar que el todo.

D&C como técnica de diseño de algoritmos:

1. **Dividir:** Dividir el problema en varios subproblemas de menor tamaño.
2. **Conquistar:** Resolver cada subproblema recursivamente.
3. **Combinar:** Combinar las soluciones de los subproblemas en una solución del problema original.

0. **Caso base:** Si el problema es suficientemente pequeño, resolverlo en forma directa.

Ejemplo: Sumar elementos de una lista

0. **Casos base:**
   - Si la lista es vacía, devolver 0.
   - Si tiene un solo elemento, devolverlo.
1. **Dividir:** Partir la lista en dos mitades.
2. **Conquistar:** Sumar los elementos de cada mitad (recursivamente).
3. **Combinar:** Sumar las dos sumas parciales y devolver el resultado.

Ejemplo: Sumar elementos de una lista

```
def sumar(L):
  if len(L) == 0:            # caso base
    res = 0
  elif len(L) == 1:          # caso base
    res = L[0]
  else:
    medio = len(L) // 2      # dividir
    s1 = sumar(L[:medio])    # conquistar
    s2 = sumar(L[medio:])    # conquistar
    res = s1 + s2            # combinar
  return res
```

¡Cuidado! La operación `L[:]` tiene complejidad lineal.
Con un razonamiento similar al visto la clase pasada, puede verse que este algoritmo es cuadrático: $O(len(L)^2)$.
¿Cómo lo podemos solucionar? ¡Con inmersión de parámetros!

Ejemplo: Sumar elementos de una lista

```
def saux(L, desde, hasta):
  if desde >= hasta:             # caso base
    res = 0
  elif desde == hasta-1:         # caso base
    res = L[desde]
  else:
    medio = (desde + hasta) // 2  # dividir
    s1 = saux(L, desde, medio)    # conquistar
    s2 = saux(L, medio, hasta)    # conquistar
    res = s1 + s2                 # combinar
  return res

def sumar(L):
  return saux(L, 0, len(L))
```

Usamos inmersión de parámetros para evitar usar la operación `L[i:j]` en cada llamado recursivo.

Ejercicio para resolver ahora

1. Escribir una función `maximo` con la técnica dividir y conquistar que, dada una lista no vacía de números, devuelva el número máximo.

Selection sort

Para cada `i` entre 0 y `len(A)`-1 (inclusive):

Buscar el menor elemento en `A[i:]`.

Intercambiarlo con `A[i]`.

0	1	2	3	4	5	6	7
59	7	388	41	2	280	50	123
2	7	388	41	59	280	50	123
2	7	388	41	59	280	50	123
2	7	41	388	59	280	50	123
2	7	41	50	59	280	388	123
2	7	41	50	59	280	388	123
2	7	41	50	59	123	388	280
2	7	41	50	59	123	280	388
2	7	41	50	59	123	280	388

Propiedad invariante: `A[0:i]` está ordenada.

```
def pos_minimo(L):
  minimo = 0                     O(1)
  for j in range(0, len(L)):     len(L) iteraciones
    if L[j] < L[minimo]:         O(1)
      minimo = j                 O(1)
  return minimo                  O(1)
```

Esto ejecuta `len(L)` veces operaciones de orden constante.

Entonces, `pos_minimo(L)` es lineal respecto de `len(L)`.

```
def selection_sort(A):
  for i in range(0, len(A)):       O(len(A)) iters.
    min = pos_minimo(A[i:]) + i    O(len(A))
    aux = A[i]                     O(1)
    A[i] = A[min]                  O(1)
    A[min] = aux                   O(1)
```

Esto ejecuta `len(A)` veces operaciones que tienen `O(len(A))`.

Entonces, `selection_sort(A)` es cuadrático respecto de `len(A)`.

Mergesort

Problema: Ordenar una lista L de enteros de longitud n.

0. **Caso base:**
1. **Dividir:**
2. **Conquistar:**
3. **Combinar:**

3	1	15	7	0	9	12	4	10	6	2
0	1	2	3	4	5	6	7	8	9	10

Mergesort

Problema: Ordenar una lista L de enteros de longitud n.

0. **Caso base:** Si L tiene longitud 0 ó 1, no hacer nada. $O(1)$
1. **Dividir:** Dividir L en 2 sublistas de longitud $\sim \frac{n}{2}$. $O(1)$
2. **Conquistar:** Ordenar cada sublista recursivamente. $2T(\frac{n}{2})$
3. **Combinar:** Combinar las 2 sublistas ordenadas. $O(n)$

Ejercicio (para después): implementar Mergesort en Python.

Complejidad:
$$T(1) = O(1)$$
$$T(n) = 2T(\tfrac{n}{2}) + O(n)$$

Complejidad de Mergesort

Queremos ver que $T(n) \in O(n \log n)$.

$$
\begin{array}{rcllcl}
T(1) & = & O(1) & & & \\
T(n) & = & 2T(\frac{n}{2}) + O(n) & & = & 2T(\frac{n}{2}) + c \cdot n \\
 & = & 2(2T(\frac{n}{4}) + c \cdot \frac{n}{2}) + c \cdot n & & = & 4T(\frac{n}{4}) + 2 \cdot c \cdot n \\
 & = & 4(2T(\frac{n}{8}) + c \cdot \frac{n}{4}) + 2 \cdot c \cdot n & & = & 8T(\frac{n}{8}) + 3 \cdot c \cdot n \\
 & \ldots & & & & \\
 & = & 2^k \cdot T(\frac{n}{2^k}) + k \cdot c \cdot n & & &
\end{array}
$$

¿Cuántas veces puedo dividir a n por 2 hasta llegar a 1? Respuesta: $\log_2 n$

$$
\begin{array}{cll}
\approx & n \cdot T(1) + \log_2 n \cdot c \cdot n & \text{cuando } k \approx \log_2 n \\
= & O(n) + O(n \log n) & \\
= & O(n \log n) &
\end{array}
$$

Por lo tanto, Mergesort tiene $O(n \log n)$.

Ejemplo de D&C: Búsqueda binaria

(Esto ya lo vimos en Fundamentos de Algoritmos.)

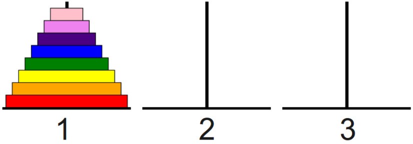
Problema: Dados una lista ordenada de enteros `A` y un entero `x`, determinar si `x` está en `A`.

0	1	2	3	4	6	7	9	10	12	15
0	1	2	3	4	5	6	7	8	9	10

1. **Dividir:** Dividir `A` en dos mitades (aproximadas).
2. **Conquistar:** Buscar recursivamente a `x` en la mitad donde pueda estar. Ignorar la otra mitad.
3. **Combinar:** Devolver la respuesta del punto anterior.
0. **Caso base:** Si `A` tiene longitud `1`, responder `True` si su único elemento es `x`, y `False` en caso contrario.

Como ya vimos, este algoritmo tiene complejidad $O(\log($`len(A)`$))$.

Ejemplo de D&C: Torre de Hanoi (difícil)



Objetivo: Mover N discos de la estaca 1 a la 3.

Restricciones:

- Mover de a un disco por vez.
- No se puede poner un disco sobre otro de menor tamaño.

Demo: https://www.mathsisfun.com/games/towerofhanoi.html

Solución: en la última página. ¡Pero piénsenlo antes! 🙏

Ayuda: la etapa de dividir no consiste en mover $N/2$ discos.

Repaso de la clase de hoy

- Técnica de dividir y conquistar
- Es una forma de pensar la recursión algorítmica
- A veces sirve para bajar órdenes de complejidad (ej: mergesort, búsqueda binaria)
- Otras veces sirve para encontrar soluciones a problemas difíciles (ej: Hanoi)

Con lo visto hoy, ya pueden resolver la sección 2 de la guía de ejercicios.

Ejemplo de D&C: Torre de Hanoi - Solución

Idea: Mover recursivamente $n-1$ discos a otra estaca, dejando el más grande en su lugar (esto sería la primera mitad del trabajo). Después mover el disco más grande a la estaca libre. Por último, mover recursivamente los $n-1$ discos sobre el disco más grande (esto sería la segunda mitad del trabajo).

```
Hanoi(n, desde, hacia, otra):
    if (n > 1):
            Hanoi(n − 1, desde, otra, hacia)
            Mover el disco superior de desde a hacia.
            Hanoi(n − 1, otra, hacia, desde)
    else:
            Mover el disco superior de desde a hacia.
```

Ej: para resolver Hanoi de 8 discos hacemos: Hanoi(8, 1, 3, 2).