Introducción a la Computación (Matemática)

Primer Cuatrimestre de 2018

Complejidad en Recursión Algorítmica

Recursión

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else $\{$
  $RV \leftarrow Sumatoria(n-1) + n$
$\}$

Recursión

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

1. Resuelvo el problema para los casos base.
2. Supongo que tengo resuelto el problema para instancias de menor tamaño; modifico dichas soluciones para obtener una solución al problema original.

Recursión

La solución a un problema depende de la solución a instancias de menor tamaño del mismo problema.

1. Resuelvo el problema para los casos base.
2. Supongo que tengo resuelto el problema para instancias de menor tamaño; modifico dichas soluciones para obtener una solución al problema original.

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!

Recordemos: Orden del tiempo de ejecución

En general, decimos que $T(n) \in O(f(n))$ si existen constantes enteras positivas c y n_0 tales que para $n \geq n_0$, $T(n) \leq c \cdot f(n)$.

Ejemplo: $T(n) = 3n^3 + 2n^2$.

$T(n) \in O(n^3)$, dado que si tomamos $n_0 = 1$ y $c = 5$, vale que para $n \geq 1$, $T(n) \leq 5 \cdot n^3$.

Ejemplo 1.1: $T(|A|) = 5 + \frac{25}{2}\,|A| + \frac{11}{2}\,|A|^2 \ \in O(|A|^2)$ (orden cuadrático)

Ejemplo 1.2: $T(|A|) = 6 + 18\,|A| \ \in O(|A|)$ (orden lineal)

Nota: Si $T(n) = cte.$ entonces $T(n) \in O(1)$ (orden constante)

La recursión y el consumo de memoria

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else $\{$
  $RV \leftarrow Sumatoria(n-1) + n$
$\}$

La recursión y el consumo de memoria

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ {
  $RV \leftarrow 0$
} else {
  $RV \leftarrow Sumatoria(n-1) + n$
}

Para cada llamado a una función, se crea un nuevo espacio de variables en la memoria.

Si se ejecutan muchos llamados recursivos, el programa puede morir por falta de memoria.

Ejemplo: `sumatoria.py`.

Cálculo de complejidad temporal

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$

if $(n = 0)$ $\{$

$\quad RV \leftarrow 0$

$\}$ else $\{$

$\quad RV \leftarrow Sumatoria(n-1) + n$

$\}$

Cálculo de complejidad temporal

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { (2)
$RV \leftarrow 0$ (1)
} else {
$RV \leftarrow Sumatoria(n-1) + n$ $(5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

Cálculo de complejidad temporal

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$

if $(n = 0)$ { (2)

$RV \leftarrow 0$ (1)

} else {

$RV \leftarrow Sumatoria(n-1) + n$ $(5 + T(n-1))$

}

$T(0) = 3$

$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?

Cálculo de complejidad temporal

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { (2)
  $RV \leftarrow 0$ (1)
} else {
  $RV \leftarrow Sumatoria(n-1) + n$ $(5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?
$T(n) = 7n + 3$ (Debe probarse que ambas def's son equivalentes.)

Cálculo de complejidad temporal

Sumatoria : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { (2)
  $RV \leftarrow 0$ (1)
} else {
  $RV \leftarrow Sumatoria(n-1) + n$ $(5 + T(n-1))$
}

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Podemos encontrar una definición no recursiva para $T(n)$?
$T(n) = 7n + 3$ (Debe probarse que ambas def's son equivalentes.)

Finalmente, $T(n) \in O(n)$.

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Elegimos $c = 10, n_0 = 1$.

- Caso base:
- Paso inductivo:

Cálculo de complejidad temporal

$T(0) = 3$
$T(n) = T(n-1) + 7$ para $n > 0$.

¿Y si no encontramos una definición no recursiva para $T(n)$?

Podemos probar por inducción que $T(n) \in O(n)$.
O sea, qvq $\exists c, n_0 \in \mathbb{N}_{>0}$ tq $T(n) \leq c \cdot n, \forall n \geq n_0$.

Elegimos $c = 10, n_0 = 1$.

- Caso base: $T(1) = T(0) + 7 = 10 = c \cdot n$ ✓
- Paso inductivo:
  Para $n \geq 1$, sup. $T(n) \leq c \cdot n$, qvq $T(n+1) \leq c(n+1)$.
  $$T(n{+}1) \stackrel{def}{=} T(n) + 7 \stackrel{HI}{\leq} 10n + 7 \leq 10n + 10 = c(n{+}1)$$ ✓

Entonces, $T(n) \in O(n)$.

Recursión algorítmica: otro ejemplo (Fibonacci)

Fib : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ $\{$
  $RV \leftarrow 0$
$\}$ else if $(n = 1)$ $\{$
  $RV \leftarrow 1$
$\}$ else $\{$
  $RV \leftarrow Fib(n-1) + Fib(n-2)$
$\}$

Recursión algorítmica: otro ejemplo (Fibonacci)

Fib : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$T(0) = O(1)$
$T(1) = O(1)$
$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$

Recursión algorítmica: otro ejemplo (Fibonacci)

Fib : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$$T(0) = O(1)$$
$$T(1) = O(1)$$
$$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$$

$T(n) \in O(2^n)$!!!

Recursión algorítmica: otro ejemplo (Fibonacci)

Fib : $n \in \mathbb{Z} \to \mathbb{Z}$
if $(n = 0)$ { $\quad O(1)$
$\quad RV \leftarrow 0 \quad O(1)$
} else if $(n = 1)$ { $\quad O(1)$
$\quad RV \leftarrow 1 \quad O(1)$
} else {
$\quad RV \leftarrow Fib(n-1) + Fib(n-2) \quad T(n-1) + T(n-2) + O(1)$
}

$T(0) = O(1)$
$T(1) = O(1)$
$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$

$T(n) \in O(2^n)$!!! Algoritmo iterativo: $O(n)$.

Cálculo complejidad Fibonacci

Para probar que $fib(n) \in O(2^n)$ podemos hacer una prueba por inducción de la siguiente manera:
El caso base es trivial por definición.
Asumamos que $T(n-1) = O(2^{n-1})$. Por lo tanto como:
$T(n) = T(n-1) + T(n-2) + O(1) \quad (n \geq 2)$
entonces por hipótesis inductiva:
$T(n) = O(2^{n-1}) + O(2^{n-2}) + O(1) = O(2^n)$

Recursión

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!

Recursión

La recursión es útil para escribir código simple y claro, pero...

- ¡Cuidado con el consumo de memoria!
- ¡Cuidado con la complejidad temporal!

La recursión no es mala o buena *per se*.

Sólo hay que tener cuidado.

A veces nos lleva a algoritmos ineficientes (ej: Fibonacci) y otras veces a algoritmos muy eficientes (ej: Mergesort).

Complejidad de Torres de Hanoi. 1

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 $hacia$.

Complejidad de Torres de Hanoi. 2

Si queremos mover sólo 1 disco desde una torre a otra:
$T(1) = 1$
Si queremos mover n discos tenemos:
$T(n) = 2 * T(n-1) + T(1) = 2 * T(n-1) + 1$
Si $n = 2$, $T(n) = 2 * T(1) + 1 = 3$
Si $n = 3$, $T(n) = 2 * T(2) + 1 = 7$
Si $n = 4$, $T(n) = 2 * T(3) + 1 = 15$
Una fórmula no recursiva para $T(n)$ podría ser $T(n) = 2^n - 1$

Complejidad de Torres de Hanoi. 2

Si queremos mover sólo 1 disco desde una torre a otra:
$T(1) = 1$
Si queremos mover n discos tenemos:
$T(n) = 2 * T(n-1) + T(1) = 2 * T(n-1) + 1$
Si $n = 2$, $T(n) = 2 * T(1) + 1 = 3$
Si $n = 3$, $T(n) = 2 * T(2) + 1 = 7$
Si $n = 4$, $T(n) = 2 * T(3) + 1 = 15$
Una fórmula no recursiva para $T(n)$ podría ser $T(n) = 2^n - 1$
Intentemos probarlo por inducción.
Caso Base: $T(1)$ se cumple trivialmente.
Paso Inductivo: Si $T(n)$ se cumple entonces ¿se cumple $T(n+1)$? HI. $T(n) = 2^n - 1$.
TI. $T(n+1) = 2^(n+1) - 1$.
$T(n+1) = 2 * T(n) + 1 =$ [def. $T(n+1)$]
$= 2 * (2^n - 1) + 1 = 2^{n+1} - 1$

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$.
2. **Conquer:** Ordenar cada subarreglo recursivamente.
   Si un subarreglo tiene tamaño 1, no hacer nada.
3. **Combine:** Combinar los 2 subarreglos ordenados.

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$. $O(1)$
2. **Conquer:** Ordenar cada subarreglo recursivamente. $2T(\frac{n}{2})$
   Si un subarreglo tiene tamaño 1, no hacer nada. $O(1)$
3. **Combine:** Combinar los 2 subarreglos ordenados. $O(n)$

Complejidad de Mergesort

Problema: Ordenar un arreglo A de enteros de tamaño n.

1. **Divide:** Dividir A en 2 subarreglos de tamaño $\sim \frac{n}{2}$. $O(1)$
2. **Conquer:** Ordenar cada subarreglo recursivamente. $2T(\frac{n}{2})$
   Si un subarreglo tiene tamaño 1, no hacer nada. $O(1)$
3. **Combine:** Combinar los 2 subarreglos ordenados. $O(n)$

Por lo tanto:

$$
\begin{aligned}
T(1) &= O(1) \\
T(n) &= 2T(\tfrac{n}{2}) + O(n)
\end{aligned}
$$

Complejidad de Mergesort

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

Complejidad de Mergesort

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

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

Complejidad de Mergesort

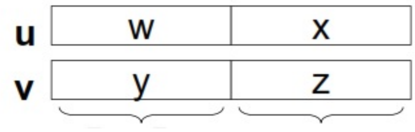
Queremos ver que $T(n) \in O(n \log n)$.

$$
\begin{aligned}
T(n) &= 2T(\tfrac{n}{2}) + O(n) \\
&\leq 2T(\tfrac{n}{2}) + c \cdot n \\
&\leq 2(2T(\tfrac{n}{4}) + c \cdot \tfrac{n}{2}) + c \cdot n = 4T(\tfrac{n}{4}) + 2 \cdot c \cdot n \\
&\leq 4(2T(\tfrac{n}{8}) + c \cdot \tfrac{n}{4}) + 2 \cdot c \cdot n = 8T(\tfrac{n}{8}) + 3 \cdot c \cdot n \\
&\leq 8(2T(\tfrac{n}{16}) + c \cdot \tfrac{n}{8}) + 3 \cdot c \cdot n = 16T(\tfrac{n}{16}) + 4 \cdot c \cdot n \\
&\ldots \\
&\leq 2^k \cdot T(\tfrac{n}{2^k}) + k \cdot c \cdot n \\
&\approx n \cdot T(1) + \log_2 n \cdot c \cdot n \qquad \text{porque } k \approx \log_2 n \\
&= O(n) + O(n \log n) \\
&= O(n \log n)
\end{aligned}
$$

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

Multiplicación rápida de enteros largos. 3

Método de Karatsuba y Ofman:



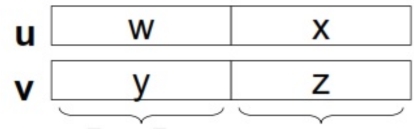
Tomo $\mathbf{S} = \mathbf{n/2}$

$$\mathbf{u} = \mathbf{w} * 10^{\mathbf{S}} + \mathbf{x}$$

$$\mathbf{v} = \mathbf{y} * 10^{\mathbf{S}} + \mathbf{z}$$

Multiplicación rápida de enteros largos. 3

Método de Karatsuba y Ofman:



Tomo $\mathbf{S} = \mathbf{n/2}$

$\mathbf{u} = \mathbf{w} * 10^{\mathbf{S}} + \mathbf{x}$

$\mathbf{v} = \mathbf{y} * 10^{\mathbf{S}} + \mathbf{z}$

Calculo la multiplicación con D&C usando:

$$\mathbf{u*v} = \mathbf{w*y} * 10^{\mathbf{2S}} + \mathbf{[(w+x)*(z+y)\ -\ w*y\ -\ x*z]} * 10^{\mathbf{S}} + \mathbf{x*z}$$

```
def karatsuba(num1, num2)
    if (num1 < 10) or (num2 < 10):
        return num1*num2
    ♯ calcular la dimensión de los números
    m = max(size(num1), size(num2))
    s = m/2
    ♯ dividir las secuencias en mitades
    high1, low1 = split(num1, s)
    high2, low2 = split(num2, s)
    ♯ hacer las tres llamadas recursivas
    z0 = karatsuba(low1, low2)
    z1 = karatsuba((low1 + high1), (low2 + high2))
    z2 = karatsuba(high1, high2)
    return (z2 * 10^(2*s)) + ((z1 - z2 - z0) * 10^s) + (z0)
```

En particular, si $n = 2^k$ para algún $k > 0$ entonces podemos ver que la cantidad de ejecuaciones de karatsuba será $3^k = n^c$ con $c = log_2 3$. En general tendremos que el número de multiplicaciones elementales será: $3^{\lceil log_2 n \rceil} \leq 3 * n^{log_2 3}$.
Entonces :
$T(n) = 3T(\lceil \frac{n}{2} \rceil) + dn$

Haciendo una construcción similar a las anteriores podemos concluir que:
$T(n) \in O(n^{log_2 3}) \approx O(n^{1,66}$

1. Ordenar los puntos según su coordenada X.
2. Si el tamaño del conjunto es 2, devolver la distancia entre ellos. Si el conjunto tiene 0 o 1 elementos, devolver infinito.
3. Dividir el conjunto de puntos en dos partes iguales (del mismo número de puntos).
4. Solucionar el problema de forma recursiva en las partes izquierdas y derecha. Esto devolverá una solución para cada parte, llamadas dLmin y dRmin. Escoger el mínimo entre estas dos soluciones, llamado dLRmin.
5. Seleccionar los puntos de la parte derecha e izquierda que están a una distancia horizontal menor que dLRmin de la recta divisoria entre ambos. Aprovechar que los puntos están ordenados para elegir los últimos puntos de la parte izquierda y los primeros de la parte derecha.
6. Encontrar la distancia mínima dCmin entre todos los pares de puntos formados por un punto de cada parte del

Ejemplo de D&C: Par más cercano

Dados n puntos (x_i, y_i) en el plano, encontrar el par de puntos más cercanos entre sí (considerar la distancia euclideana).

Algoritmo exhaustivo: $O(n^2)$.
La relación de recurrencia para el número de pasos es $T(n) = 2T(n/2) + O(n)$, y podemos hacer el desarrollo de los casos anteriores para concluir que:
Algoritmo *divide and conquer*: $O(n \log n)$.

Repaso de la clase de hoy

- Complejidad de algoritmos recursivos.
- Consumo de memoria de la recursión.

Próximos temas

- Tipos abstractos de datos.