TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 6
Complejidad algorítmica

Resumen

En la clase de hoy veremos:

- Repaso noción de complejidad algorítmica.
- Definición formal de órden de complejidad: O.
- Peor caso, caso promedio, mejor caso.
- Cálculo de complejidad de algoritmos **imperativos**.
- Cálculo de complejidad de algoritmos **recursivos** (de *Divide & Conquer*).

Repaso noción de complejidad algorítmica

Dado un vector de enteros no vacío y ordenado de forma creciente, ¿cuál algoritmo elegirían para encontrar el mínimo del vector?

```cpp
int minimo(const vector<int> & v){
    int res = v[0];
    int i = 0;
    while(i < v.size()){
        if (v[i] < res){
            res = v[i];
        }
        i = i + 1;
    }
    return res;
}
```

```cpp
int minimo(const vector<int> & v){
    return v[0];
}
```

El primero toma tiempo lineal y el segundo tiempo constante.

Repaso noción de complejidad algorítmica

> Complejidad algorítmica es una herramienta para analizar algoritmos según su consumo de recursos, en particular su tiempo de ejecución.

- La pregunta central es ¿cómo crece el tiempo de ejecución en función del tamaño de la entrada?
- Las operaciones sobre tipos básicos toman tiempo constante.
  - Ej: `2 + 7`, `75.0/3.2`, `5 > 0`
- La lectura y escritura de variables de tipos básicos toman tiempo constante.
  - Ej: `int i = 0`, `i = i + 1`, `char c = 'a'`
- El costo de un ciclo es la suma de los costos de todas sus iteraciones.
- [Nuevo] El costo de las operaciones de tipos de la biblioteca estándar está indicado en su documentación.
  - Ej: las operaciones `v[i]` y `v.size()` toman tiempo constante
    https://es.cppreference.com/w/cpp/container/vector/operator_at
    https://es.cppreference.com/w/cpp/container/vector/size

$T(n)$: costo temporal

- En TD1: este algoritmo tiene complejidad $O(n)$, donde $n = |v|$.
- Pero, ¿qué significa exactamente la O?

```
int minimo(const vector<int> & v){
    int res = v[0];
    int i = 0;
    while(i < v.size()){

        if (v[i] < res){
            res = v[i];
        }
        i = i + 1;
    }
    return res;
}
```

$T(n)$: costo temporal

- En TD1: este algoritmo tiene complejidad $O(n)$, donde $n = |v|$.
- Pero, ¿qué significa exactamente la O?

```
int minimo(const vector<int> & v){
    int res = v[0];          // 3 ops: v, [], =
    int i = 0;               // 1 op: =
    while(i < v.size()){     // n iteraciones
                             // 4 ops: i, v, size(), <
        if (v[i] < res){     // 5 ops: v, i, [], res, <
            res = v[i];      // 4 ops: v, i, [], =
        }
        i = i + 1;           // 3 ops: i, +, =
    }
    return res;              // 1 op: res
}
```

$$\text{Costo temporal } T(n) = 3+1+n\cdot(4+5+4+3)+4+1$$
$$= 9+16n$$

$T(n)$: costo temporal

- Otra versión del algoritmo:

```
int minimo(const vector<int> & v){
    int res = v[0];
    int i = v.size()-1;
    while(i > 0) {

        if (v[i] < res){
            res = v[i];
        }
        i = i - 1;
    }
    return res;
}
```

$T(n)$: costo temporal

- Otra versión del algoritmo:

```
int minimo(const vector<int> & v){
    int res = v[0];         // 3 ops: v, [], =
    int i = v.size()-1;     // 4 ops: v, size(), -, =
    while(i > 0) {          // n-1 iteraciones
                            // 2 ops: i, >
        if (v[i] < res){    // 5 ops: v, i, [], res, <
            res = v[i];     // 4 ops: v, i, [], =
        }
        i = i - 1;          // 3 ops: i, +, =
    }
    return res;             // 1 op: res
}
```

$$
\begin{aligned}
\text{Costo temporal } T(n) &= 4+3+(n-1)\cdot(2+5+4+3)+2+1 \\
&= 10+14(n-1) \leftarrow \text{¡El costo temporal es distinto!}
\end{aligned}
$$

Orden de complejidad O

- El órden de complejidad indica en qué medida crece el tiempo de ejecución de un algoritmo a medida que crece el tamaño de la entrada.
- Formalmente es el conjunto de todas las funciones cuyo ritmo de crecimiento es el mismo asintóticamente.

Por ejemplo, a las siguientes funciones...

- $\frac{5}{3}n + 5$
- $2n$
- $\frac{2}{3}n^3 + 8$
- $7n^2 + n$
- $3n + 5$
- $n^2 + 5$
- $2n^3 + 3$

Nos gustaría agruparlas así:

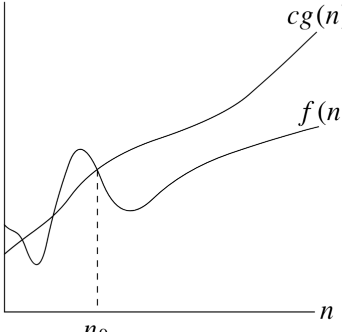
- Órden lineal
  - $2n, \quad 3n + 5, \quad \frac{5}{3}n + 5$
- Órden cuadrático
  - $n^2 + 5, \quad 7n^2 + n$
- Órden cúbico
  - $2n^3 + 3, \quad \frac{2}{3}n^3 + 8$

Notación O: cota superior asintótica

$f(n) \in O(g(n))$ si y sólo si

existen $c > 0$ y $n_0 > 0$ tal que

$0 \leq f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$



$f(n) = O(g(n))$

Notación O: cota superior asintótica

Veamos que $\frac{5}{3}n+5$ está en $O(n)$.

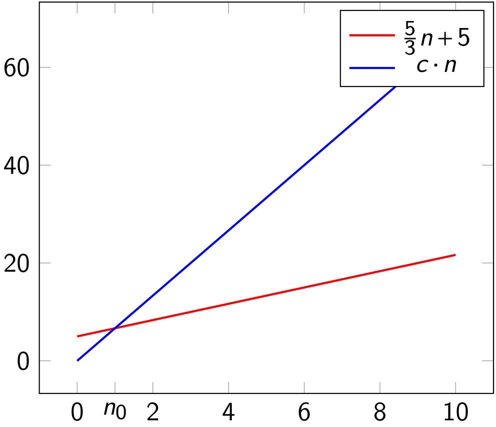
- Queremos ver que existen $c>0$ y $n_0>0$ tal que $0 \leq \frac{5}{3}n+5 \leq c \cdot n$ para todo $n \geq n_0$.
- Para demostrar podemos elegir c y n_0.
- La desigualdad $0 \leq \frac{5}{3}n+5$ vale porque $n_0>0$ y la función es creciente.
- Ahora planteamos la segunda desigualdad y despejamos c:

$$\begin{aligned} \frac{5}{3}n+5 &\leq c \cdot n \\ (\frac{5}{3}n+5)/n &\leq c \\ \frac{5}{3}+\frac{5}{n} &\leq c \end{aligned}$$

- Tomando $n_0=1$ y $c=\frac{5}{3}+5$, la desigualdad vale para todo $n \geq n_0$.

Notación O: cota superior asintótica

Veamos que $\frac{5}{3}n + 5$ está en $O(n)$.



$$c = \frac{5}{3} + 5, n_0 = 1$$

Operando con O: álgebra de órdenes

Para todas:

- constante $k > 0$,
- funciones f_1, f_2, g_1, g_2 tales que $f_1 \in O(g_1)$ y $f_2 \in O(g_2)$.

valen las siguientes igualdades:

$$k \in O(1) \text{ y } O(k) = O(1)$$

(complejidad constante)

$$Ej : 35 \in O(1)$$

$$k.f_1 \in O(g_1) \; y \; O(k.g_1) = O(g_1)$$

(producto por constante)

$$Ej : 35n \in O(n)$$

$$f_1 + f_2 \in O(max\{g_1, g_2\})$$

(suma de funciones)

$$Ej : 35n^2 + 100n \in O(max\{n^2, n\}) = O(n^2)$$

$$Ej : O(n) + O(n) = O(n)$$

$$f_1.f_2 = O(g_1.g_2)$$

(producto de funciones)

$$Ej : 35n * 50 \log n = O(n * \log n)$$

Álgebra de órdenes : ejemplo

```
int minimo(const vector<int> & v){
    int res = v[0];          // 3 ops: v, [], =
    int i = 0;               // 1 op: =
    while(i < v.size()){     // n iteraciones
                             // 4 ops: i, v, size(), <
        if (v[i] < res){     // 5 ops: v, i, res, [], <
            res = v[i];      // 4 ops: v, i, [], =
        }
        i = i + 1;           // 3 ops: i, +, =
    }
    return res;              // 1 op: res
}
```

Álgebra de órdenes : ejemplo

```
int minimo(const vector<int> & v){
    int res = v[0];         // O(3)
    int i = 0;              // O(1)
    while(i < v.size()){    // n
                            // O(4)
        if (v[i] < res){    // O(5)
            res = v[i];     // O(4)
        }
        i = i + 1;          // O(3)
    }
    return res;             // O(1)
}
```

$O(k) = O(1)$ para toda constante k

Álgebra de órdenes : ejemplo

```cpp
int minimo(const vector<int> & v){
    int res = v[0];          // O(1)
    int i = 0;               // O(1)
    while(i < v.size()){     // n
                             // O(1)
        if (v[i] < res){     // O(1)
            res = v[i];      // O(1)
        }
        i = i + 1;           // O(1)
    }
    return res;              // O(1)
}
```

Complejidad:

$O(1) + O(1) + n \times \left(O(1) + O(1) + O(1) + O(1)\right) + O(1)$

$= O(1) + n \times O(1) + O(1)$

$= O(n) + O(1)$

$= O(n)$

Jerarquía de complejidades

Descripción	Ejemplo	Algoritmos
Constante	$O(1)$	Acceso a vector
Logarítmica	$O(\log n)$	Búsqueda binaria
Sublineal	$O(\sqrt{n})$	Chequeo de primalidad
Lineal	$O(n)$	Counting sort
“Linearítmica”	$O(n \log n)$	Mergesort
Subcuadrática	$O(n^{\log 3})$	Karatsuba
Cuadrática	$O(n^2)$	Insertion sort
Cúbica	$O(n^3)$	Producto de matrices de $n \times n$
Exponencial	$O(2^n)$	Enumerar subconjuntos
Factorial	$O(n!)$	Enumerar permutaciones

Peor caso, caso promedio, mejor caso

Sea A un algoritmo que toma una entrada de tamaño n, nos interesa analizar su costo $T(n)$.

- Peor caso: Dado n, ¿qué valores tiene que tomar la entrada para que el algoritmo tenga el costo computacional más alto?
- Caso promedio: Dado n, ¿cuál es el costo computacional promedio considerando la probabilidad de todos los posibles valores de entrada de tamaño n?
- Mejor caso: Dado n, ¿qué valores tiene que tomar la entrada para que el algoritmo tenga el costo computacional más bajo?

- La elección de entre estos casos potencialmente cambia el costo $T(n)$ obtenido.
- En esta materia nos concentraremos únicamente en el peor caso.

Peor caso, caso promedio, mejor caso

¿Cuál es el tamaño de la entrada para este problema? ¿Cuál es el peor caso y el mejor caso para este algoritmo?

```cpp
bool buscar(int elem, const vector<int> & v){
    bool res = false;
    int i = 0;
    while(i < v.size() && res == false){
        if (v[i] == elem){
            res = true;
        }
        i = i + 1;
    }
    return res;
}
```

- Tamaño de la entrada: $|v|$
- Peor caso: *elem* no está en v.
- Mejor caso: *elem* es el primer elemento de v.

Cálculo de complejidad de algoritmos imperativos

1. Determinar cuál es la medida de tamaño de entrada para el algoritmo dado.
2. Determinar qué valores de entrada constituyen el peor caso del algoritmo.
3. Derivar del código del algoritmo la función de costo T que toma el tamaño de entrada como parámetro y calcula el costo computacional para el peor caso.
4. Proponer una función f que será el órden de complejidad y demostrar que $T \in O(f)$.

Cálculo de complejidad de algoritmos **recursivos**

Volvamos al algoritmo **suma_rec2** de la clase pasada...

```
int suma_rec2(vector<int> v, unsigned desde, unsigned hasta) {
  if (hasta <= desde)
      // Caso base 1: rango vacío
      return 0;
  else if (desde == hasta-1)
      // Caso base 2: 1 elemento
      return v[desde];
  else  { // 2 o más elementos: recursión
      unsigned int medio = (desde+hasta)/2;
      return suma_rec2(v,desde,medio)
           + suma_rec2(v,medio,hasta);
  }
}
```

...qué complejidad $T(n)$ tendrá?

Cálculo de complejidad de algoritmos **recursivos**

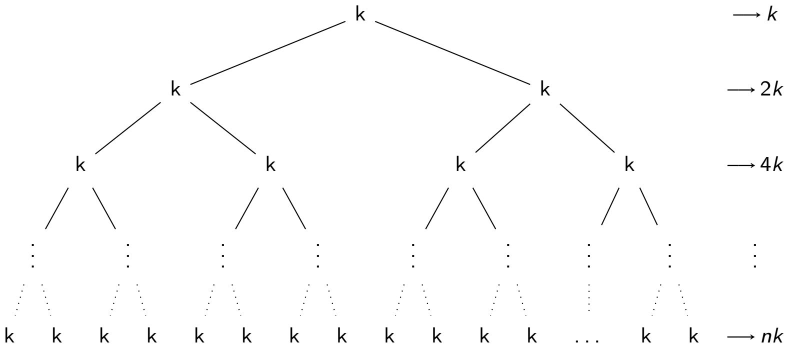
Contemos las operaciones constantes...

```
int suma_rec2(vector<int> v, unsigned desde, unsigned hasta) {
  if (hasta <= desde)                                 // 3 op. const.
      // Caso base 1: rango vacío
      return 0;                                       // 1 op. const.
  else if (desde == hasta-1)                          // 4 op. const.
      // Caso base 2: 1 elemento
      return v[desde];                                // 3 op. const.
  else  { // 2 o más elementos: recursión
      unsigned int medio = (desde+hasta)/2;           // 5 op. const
      return suma_rec2(v,desde,medio)                 // T(n/2)
           + suma_rec2(v,medio,hasta);                // T(n/2)
  }
}
```

El tamaño de la entrada n es igual a $hasta - desde$.

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + k & \text{si } n > 1 \end{cases}$$

Árbol de recursion para la $T(n)$ planteada para **suma_rec2**



- Altura del árbol: $\log n$ (con $\log n + 1$ niveles).
- Suma del nivel i: $2^i k$.
- Total: $k \sum_{i=0}^{\log_2(n)} 2^i$ (es un caso de la serie geométrica)

$$= k \sum_{i=0}^{\log_2(n)-1} 2^i + kn$$
$$= k(n-1) + kn = 2k \cdot n - k$$

- Propuesta: $T(n) \in O(n)$

Demostración por inducción

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + k & \text{si } n > 1 \end{cases}$$

- Queremos ver que $T(n) \leq c \cdot n$, para algún c y a partir de algún n_0.
- Casos base:
  - Con $n = 0$,
    $T(0) \leq c \cdot 0$. Sólo sirve si k fuese 0.
  - Con $n = 1$,
    $T(1) \leq c \cdot 1$
    $k \leq c \cdot 1$, se cumple si tomo $c \geq k$.

Demostración por inducción

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + k & \text{si } n > 1 \end{cases}$$

- Caso inductivo: queremos ver que $T(n) \leq c \cdot n$
- Hipótesis inductiva: vale $T(x) \leq c \cdot x$ para $x < n$.
- Sabemos que $T(n) = 2T(n/2) + k$
- $\leq 2c(n/2) + k$, aplicando hipótesis inductiva con $x = n/2$:
- $= cn + k$
- $\leq c \cdot n$?
- ... No podemos demostrarlo: hay que elegir una hipótesis inductiva más fuerte.

Demostración por inducción: segundo intento

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + k & \text{si } n > 1 \end{cases}$$

Elegimos probar una propiedad más fuerte que la propiedad que nos interesa.

- Queremos ver que $T(n) \leq c \cdot n - k$, para algún c y a partir de algún n_0. *(pues, si probamos esto, probamos indirectamente que $T(n) \leq c \cdot n$)*
- Casos base:
  - Con $n = 0$,
    $T(0) \leq c \cdot 0 - k$.
    $k \leq -k$. No se puede porque k es positivo.
  - Con $n = 1$,
    $T(1) \leq c \cdot 1 - k$
    $k \leq c - k$ Sirve para $c \geq 2k$.
  - Tomamos $c = 2k$ y $n_0 = 1$

Demostración por inducción: segundo intento

$$T(n) = \begin{cases} k & \text{si } n = 0, n = 1, \\ 2T(n/2) + k & \text{si } n > 1 \end{cases}$$

- Caso inductivo: queremos ver que $T(n) \leq c \cdot n - k$.
- Hipótesis inductiva: vale $T(x) \leq c \cdot x - k$ para $x < n$.
- Sabemos que $T(n) = 2T(n/2) + k$
- $\leq 2(c(n/2) - k) + k$, aplicando hipótesis inductiva con $x = n/2$:
- $= c \cdot n - 2k + k$
- $= c \cdot n - k$
- Queda demostrado por inducción que $\mathrm{T(n)} \in \mathrm{O}(n)$.

Complejidad de Divide & Conquer general

Los problemas de Divide & Conquer tienen la siguiente recurrencia:

$$T(n) = \begin{cases} c & \text{si } n = 0, n = 1, \\ aT(n/b) + f(n) & \text{si } n > 1 \end{cases}$$

(a subproblemas de tamaño n/b, $f(n)$: costo de conquer)

Hay una técnica llemada Teorema Maestro (*ver Cormen*) que da una "receta" para calcular la complejidad de recurrencias de esta forma.

Algunas recurrencias comunes y sus órdenes de complejidad

$aT(n/a) + O(n) = O(n \log_a n)$ *(a=2 es mergesort)*

$2T(n/2) + O(1) = O(n)$ *(recorrer arbol binario)*

$T(n/2) + O(1) = O(\log n)$ *(búsqueda binaria)*

Repaso de la clase

Hoy vimos

- Definición formal de O.
- Concepto de peor caso, mejor caso y caso promedio
- Cómo calcular complejidad de un algoritmo imperativo.
- Complejidad de algoritmos (recursivos) de Divide & Conquer

Bibliografía:

- Peor caso y análisis de algoritmos
  - **"Introduction to Algorithms, Fourth Edition" por Thomas Cormen**, capítulo 2.2.
- Notación asintótica O.
  - **"Introduction to Algorithms, Fourth Edition" por Thomas Cormen**, capítulo 3.