Análisis de la Complejidad de Algoritmos

Algoritmos y Estructuras de Datos

Análisis de la Complejidad de Algoritmos

- Para resolver problemas, tenemos que diseñar algoritmos y estructuras de datos que:
  - Funcionen correctamente
    - modelen bien el problema,
    - los algoritmos terminen
    - y produzcan el resultado correcto
  - Sean eficientes en términos del consumo de recursos
- Esa medida de eficiencia nos permitirá elegir entre
  - distintos algoritmos para resolver el mismo problema
  - distintas formas de implementar un TAD

Análisis de la Complejidad de Algoritmos

- ¿Cuáles son esos recursos que se consumen?
  - Tiempo de ejecución
  - Espacio (memoria)
  - Cantidad de procesadores (en el caso de algoritmos paralelos)
  - Utilización de la red de comunicaciones (para algoritmos paralelos)
- Otros criterios (de interés para la Ingeniería de Software):
  - Claridad de la solución
  - Facilidad de codificación

Dependiendo de cómo balanceemos la importancia de cada uno de los criterios, podremos decir que un algoritmo es mejor que otro.

- Nos vamos a ocupar principalmente del tiempo
- Un poquitito del espacio

Análisis de la Complejidad de Algoritmos

¿Cómo?

El análisis de la complejidad de un algoritmo se puede hacer de forma:

- Empírica o experimental
  - Medir el tiempo de ejecución para una determinada entrada y en una computadora concreta
  - Usando un cronómetro, o analizando el consumo de recursos de la computadora (tiempo de CPU)
  - Medidas del tipo: 3GB; 1,5 segundos.
- Teórica
  - Medida teórica del comportamiento de un algoritmo

Análisis de la Complejidad de Algoritmos

Ejemplo - Búsqueda Lineal

Problema: "Búsqueda en un array": dado un array A y un elemento x, encontrar si x pertence al array

```
boolean contiene(int[] s, int x) {
     int i = 0;
     while (i < s.length && s[i] != x) {
         i = i + 1;
     }
     return i < s.length;
}
```

- ¿Cuánto tarda la ejecución de contiene(5, [2, 6, 3, 5, 8])?
- ¿Cuánta memoria necesito?

Análisis teórico

Ventajas del enfoque teórico:

- El análisis se puede hacer a priori, aún antes de escribir una línea de código
- Vale para todas las instancias del problema
- Es independiente del lenguaje de programación utilizado para codificarlo
- Es independiente de la máquina en la que se ejecuta
- Es independiente de la pericia del programador
- Basado en un "modelo de máquina" o "modelo de cómputo" consensuado
  - En función del tamaño del input
  - Para distintos tipos de input
  - Análisis asintótico

Análisis teórico
Modelo de cómputo

Queremos una medida "universal", válida para distintas implementaciones del algoritmo

- La idea es tener un "banco de pruebas"...pero teórico, virtual.

Definimos una máquina "ideal", que vamos a usar para definir los conceptos de tiempo y espacio

- **Medida del tiempo:** número de pasos o instrucciones que se ejecutan en esa máquina "ideal" para determinado input
- **Medida del espacio:** número de posiciones de memoria en esa máquina ideal que se utilizan para determinado input

Análisis teórico
Modelo de cómputo

¿Cuánto tarda la ejecución de este algoritmo?

```
int I = exp(2, exp(2, x));
print(I)
```

¿Tarda más, menos o lo mismo que?

```
int I = 2*x;
print(I)
```

Análisis teórico

Operaciones Elementales

Operaciones elementales (OE): serán aquellas que el procesador realiza en tiempo acotado por una constante (que no depende del tamaño de la entrada).

Consideraremos OE las operaciones aritméticas básicas, comparaciones lógicas, transferencias de control, asignaciones a variables de tipos básicos, etc.

Nota: En el etc. está el peligro. Por eso es importante definir bien el modelo de cómputo, y cuáles son las operaciones elementales.

¿Qué pasa si el próximo modelo de computadora permite hacer las operaciones mucho más rápidamente?

¡Nada! (lo veremos después)

Con esto vamos a definir **t(I)**, una función que mida el número de operaciones elementales requeridas para la instancia I.

Análisis teórico

Operaciones Elementales

Consideraciones generales:

- Vamos a considerar que el tiempo de una OE es, por definición, 1.
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

Análisis teórico

Contando Operaciones Elementales

```
boolean contiene(int[] s, int x) {
    int i = 0;
    while (i < s.length && s[i] != x) {
        i = i + 1;
    }
    return i < s.length;
}
```

- ¿Cuánto tarda la ejecución de `contiene(5, [2, 6, 3, 5, 8] )`?
- ¿Cuánto tarda la ejecución de `contiene(6, [2, 6, 3, 5, 8] )`?
- ¿Cuánto tarda la ejecución de `contiene(8, [2, 6, 3, 5, 8] )`?
- ¿Cuánto tarda la ejecución de `contiene(1, [2, 6, 3, 5, 8, 4, 8, 9, 2, 1])`?

Análisis teórico

Análisis del caso peor, medio, o mejor

Distintas instancias, aunque tengan el **mismo** tamaño, pueden hacer que el algoritmo se comporte de maneras muy diferentes, y por lo tanto, tomar **distinto tiempo**, y/o requerir **distinta cantidad** memoria.

Por eso estudiamos tres casos para un mismo algoritmo:

- caso peor
- caso mejor
- caso medio

Análisis teórico

Análisis del Caso Peor

- Sea $t(I)$ el tiempo de ejecución de un algoritmo sobre una instancia I.

  $T_{peor}(n) = max_{\text{instancias I},|I|=n}\{t(I)\}$

  Intuitivamente, $T_{peor}(n)$ es el tiempo de ejecución del algoritmo sobre la instancia que implica mayor tiempo de ejecución (entre los inputs de tamaño n).
- Da garantías sobre las prestaciones del algoritmo.

Análisis teórico

Análisis del Caso Mejor

- Sea $t(I)$ el tiempo de ejecución de un algoritmo sobre una instancia I.

  $T_{mejor}(n) = mmin_{\text{instancias I},|I|=n}\{t(I)\}$

  Intuitivamente, $T_{mejor}(n)$ es el tiempo de ejecución del algoritmo sobre la instancia que implica menor tiempo de ejecución (entre los inputs de tamaño n).
- No da mucha información...

Análisis teórico
Análisis del Caso Medio

- Intuitivamente, $T_{prom}(n)$ corresponde al tiempo **"promedio"** de ejecución. Al tiempo **"esperado"** sobre instancias **"típicas"**.
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.
  - $P(I)$: probabilidad de que el input sea la instancia I.
  - $T_{prom}(n) = \sum_{\text{instancias } I, |I|=n} P(I)t(I)$
- ¿Por qué las comillas?
  - Requiere conocer la distribución estadística del input: en muchos casos eso no es realista.
  - En muchos casos la matemática se complica, y se termina haciendo hipótesis simplificatorias poco realistas.
  - Podemos tener algoritmos para los cuales ningún input requiere tiempo medio.

Análisis teórico
Ejemplo

```
boolean contiene(int[] s, int x) {
    int i = 0;
    while (i < s.length && s[i] != x) {
        i = i + 1;
    }
    return i < s.length;
}
```

- $T_{mejor}(n) = ?$
- $T_{peor}(n) = ?$
- $T_{prom}(n) = ?$

¿Cambia si está ordenado?

¿Cómo quedan estos tiempos para búsqueda binaria?

Análisis teórico
Reglas Generales (Pensando en Análisis del Caso Peor)

- Tiempo de ejecución de `if C then S1 else S2 endif;`:

$$T = T(C) + \max\{T(S1), T(S2)\}$$

- Tiempo de ejecución de un bucle de sentencias `while C do S end;`:

$$T = T(C) + (n_{iteraciones}) * (T(S) + T(C)).$$

  Nota: tanto T(C) como T(S) pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.

- Tiempo de ejecución del resto de sentencias iterativas (ej: for): Basta expresarlas como un bucle while.

Análisis teórico

Reglas Generales (Pensando en Análisis del Caso Peor)

- Tiempo de ejecución de una llamada a un procedimiento o función F(P1, P2,..., Pn):

$$T = 1 + T(P1) + T(P2) + ... + T(Pn) + T(F).$$

Nota: 1 (por la llamada) + tiempo de evaluación de los parámetros P1, P2,..., Pn, + el tiempo que tarde en ejecutarse F, esto es:
  - No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura.
  - El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- Tiempo de ejecución de las llamadas a procedimientos recursivos: lo veremos posteriormente

Análisis teórico

Principio de invarianza

- Dado un algoritmo y dos máquinas (o dos implementaciones) M_1 y M_2, que tardan $T_1(n)$ y $T_2(n)$ respectivamente sobre inputs de tamaño n,
- Existe una cte. real $c > 0$ y un $n_0 \in \mathbb{N}$ tales que $\forall n \geq n_0$ se verifica que:

$$T_1(n) \leq cT_2(n)$$

- **Explicación:** dos ejecuciones distintas del mismo algoritmo solo difieren en cuanto a eficiencia en un factor constante para valores de la entrada suficientemente grandes.
- **Consecuencia:** no necesitamos usar ninguna unidad para medir el tiempo.

Análisis teórico

Tamaño de la Entrada

- ¿Por qué complejidad en función del tamaño de la entrada?
  - Queremos una complejidad relativa, no absoluta.
  - Es una medida general de lo que podemos encontrarnos al ejecutar.
  - Más abstracta que pensarlo en función de cada input (¡en general, podría haber infinitos inputs distintos!).
- $T(n)$**:** complejidad temporal (o en tiempo) para una entrada de tamaño n.
- $S(n)$**:** complejidad espacial para una entrada de tamaño n.

Análisis teórico
Análisis Asintótico

- Los distintos algoritmos que resuelven un mismo problema pueden tener grandes diferencias en su tiempo de ejecución, a veces, de **órdenes de magnitud.**
- Interesa calcular, de forma aproximada, el **orden de magnitud** que tiene el tiempo de ejecución de cada algoritmo.
- Cuando el tamaño de los datos es pequeño no habrá diferencias significativas en el uso de distintos algoritmos.
- Cuando el **tamaño de los datos es grande**, los **costos** de los diferentes algoritmos sí pueden **variar de manera significativa**.
- **El orden** (logarítmico, lineal, cuadrático, exponencial, etc.) de la función T(n) expresa el comportamiento dominante cuando el tamaño de la entrada es grande.
- **Comportamiento Asintótico:** comportamiento para valores de la entrada suficientemente grandes

Análisis teórico
Análisis Asintótico

¿Por qué quiero algoritmos que sean eficientes asintóticamente?

Complejidad Temporal	Tamaño n					
	10	20	30	40	50	60
n	0.00001 segundos	0.00002 segundos	0.00003 segundos	0.00004 segundos	0.00005 segundos	0.00006 segundos
n^2	0.0001 segundos	0.0004 segundos	0.0009 segundos	0.0016 segundos	0.0025 segundos	0.0036 segundos
n^3	0.001 segundos	0.008 segundos	0.027 segundos	0.064 segundos	0.125 segundos	0.216 segundos
n^4	0.1 segundos	3.2 segundos	24.3 segundos	1.7 minutos	5.2 minutos	13.0 minutos
2^n	0.001 segundos	1.0 segundos	17.9 minutos	12.7 días	35.7 años	366 siglos
3^n	0.059 segundos	58 minutos	6.5 años	3855 siglos	2×10^5 siglos	$1,3 \times 10^3$ siglos

Comparación de algoritmos con distintas complejidades, polinomiales y exponenciales

Fuente: Aho, Hopcroft, Ullman

Análisis teórico
Análisis Asintótico

¿Por qué?

Complejidad Temporal	Máximo tamaño de un problema resoluble en una hora		
	Con una computadora a actual	Con una computadora a 100 veces más veloz	Con una computadora a 1000 veces más veloz
n	N_1	100 N_1	1000 N_1
n^2	N_2	10 N_2	31,6 N_2
n^3	N_3	4,64 N_3	10 N_3
n^4	N_4	2,5 N_4	3,98 N_4
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Efecto de la mejora de la tecnología en algoritmos con distintas complejidades

Fuente: Aho, Hopcroft, Ullman

Análisis Asintótico

El objetivo del **estudio de la complejidad** algorítmica es determinar el **comportamiento asintótico** de un algoritmo.

Medidas del comportamiento asintótico de lacomplejidad:

- O (O grande) cota superior.
- Ω (omega) cota inferior.
- Θ (theta) orden exacto de la función.

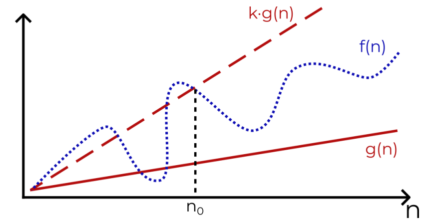
Análisis Asintótico
Cota Superior (O)

La notación O (O grande, O mayúscula) sirve para representar el límite o cota superior del tiempo de ejecución de un algoritmo.

- La notación $f \in O(g)$ expresa que la función f no crece más rápido que alguna función proporcional a g.
- A g se le llama cota superior de f.
- Si para un algoritmo sabemos que $T_{peor} \in O(g)$, se puede asegurar que para inputs de tamaño creciente, en todos los casos el tiempo será a lo sumo proporcional a la cota.
- Si para un algoritmo sabemos que $T_{prom} \in O(g)$, se puede asegurar que para inputs de tamaño creciente, en promedio el tiempo será proporcional a la cota.

Análisis Asintótico
Cota Superior (O)

- Asumimos funciones reales no negativas con dominio en los naturales.
- $f \in O(g)$ significa que f no crece más que g.
- $O(g) = \{f \mid \exists n_0, k > 0 \text{ tal que } n \geq n_0 \Rightarrow f(n) \leq k \cdot g(n)\}$



Ejemplos:

- $100n^2 + 300n + 1000 \in O(n^2)$
- $100n^2 + 300n + 1000 \in O(n^3)$

Análisis Asintótico

Cota Superior (O) - Propiedades

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \Rightarrow O(f) \subseteq O(g)$.
3. $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$.
4. Si $f \in O(g)$ y $g \in O(h) \Rightarrow f \in O(h)$.
5. Si $f \in O(g)$ y $f \in O(h) \Rightarrow f \in O(\min(g, h))$.
6. Regla de la suma: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g, h))$.
7. Regla del producto: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:
   8.1 Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$.
   8.2 Si $k = 0$ entonces $f \in O(g)$, es decir, $O(f) \subseteq O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

Análisis Asintótico

Funciones de Complejidad Temporal Comunes

- $O(1)$ **Complejidad constante.** Es independiente de los datos de entrada.
- $O(\lg n)$ **Complejidad logarítmica.** Suele aparecer en determinados algoritmos con iteración o recursión (p.e., búsqueda binaria). Todos los logaritmos, sea cual sea su base, son del mismo orden, por lo que se representan en cualquier base.
- $O(n)$ **Complejidad lineal.** Suele aparecer en bucles simples cuando la complejidad de las operaciones internas es constante o en algunos algoritmos con recursión.

Análisis Asintótico

Funciones de Complejidad Temporal Comunes

- $O(n \log n)$
  - En algunos algoritmos "Divide & Conquer"(p.e. Mergesort)
- $O(n^2)$ Complejidad cuadrática
  - Aparece en bucles o recursiones doblemente anidados
- $O(n^3)$ Complejidad cúbica
  - En bucles o recursiones triples
- $O(n^k)$ Complejidad polinómica ($k \geq 1$)
- $O(2^n)$ Complejidad exponencial
  - Suele aparecer en subprogramas recursivos que contengan dos o más llamadas internas

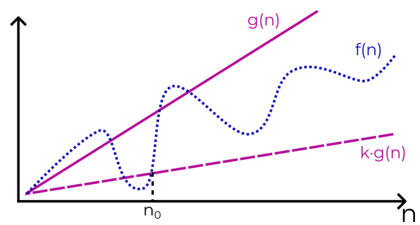
Notación: Si $f \in O(g)$ vamos a abusar de la notación y escribir a veces $f = O(g)$ y decir cosas como "f es O de g.º "f es O grande de g.º "f es del orden de g".

Cota Inferior (Ω)

- La notación $\Omega(g)$ permite representar el límite o cota inferior del tiempo de ejecución de un algoritmo.
- La notación $f \in \Omega(g)$ expresa que la función f está acotada inferiormente por alguna función proporcional a g.
- A g se le llama cota inferior de f.
- Si para un algoritmo sabemos que $T_{peor} \in \Omega(g)$, se puede asegurar que para inputs de tamaño creciente, el tiempo será, en el peor caso, al menos proporcional a la cota.
- La notación se usa también para dar cotas inferiores para problemas. A veces se puede decir para un problema que para cualquier algoritmo que lo resuelva, $T_{peor} \in \Omega(g)$, lo que significa que cualquier algoritmo que lo resuelva tiene una complejidad, en el peor caso, proporcional a la cota.

Cota Inferior (Ω)

- $f \in \Omega(g)$ significa que f crece al menos como g.
- $\Omega(g) = \{f \mid \exists n_0, k > 0 \text{ tal que } n \geq n_0 \Rightarrow f(n) \geq k \cdot g(n)\}$



Ejemplos:

- $100n^2 + 300n + 1000 \in \Omega(n^2)$
- $100n^2 + 300n + 1000 \in \Omega(n)$

Cota Inferior (Ω)

Obtener buenas cotas inferiores **ajustadas** es en general **difícil**, aunque siempre existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior.

Por ejemplo, para ordenar n números una cota inferior sería $\Omega(n)$, y para multiplicar dos matrices de orden n sería $\Omega(n^2)$; sin embargo, los mejores algoritmos conocidos son de órdenes $O(nlogn)$ y $O(n^{2,8})$ respectivamente.

Cota Inferior (Ω) - Propiedades

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \Rightarrow \Omega(f) \subset \Omega(g)$.
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$.
4. Si $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$.
5. Si $f \in \Omega(g)$ y $f \in \Omega(h) \Rightarrow f \in \Omega(\text{máx}(g, h))$.
6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$.
7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$.
8. Si existe $\text{lím}_{n\to\infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:
   8.1 Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$.
   8.2 Si $k = 0$ entonces $g \in \Omega(f)$, es decir, $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $g \notin O(f)$.

Orden Exacto (Θ)

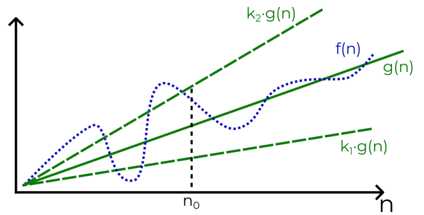
- Como última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

$$\Theta(f) = O(f) \cap \Omega(f)$$

- Intuitivamente, $t \in \Theta(f)$ indica que t está acotada por f tanto superior como inferiormente.

Orden Exacto (Θ)

- $f \in \Theta(g)$ significa que f igual que g.
- $\Theta(g) = \{f \mid \exists n_0, k_1, k_2 > 0 \text{ tal que } n \geq n_0 \Rightarrow k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)\}$



Ejemplo:
- $100n^2 + 300n + 1000 \in \Theta(n^2)$

Orden Exacto (Θ) - Propiedades

1. Para cualquier función f se tiene que $f \in \Theta(f)$.
2. $f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$.
3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$.
4. Si $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$.
5. Si $f \in \Theta(g)$ y $f \in \Theta(h) \Rightarrow f \in \Theta(\text{máx}(g, h))$.
6. Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(\text{máx}(g, h)) = \Theta(g + h)$.
7. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 \cdot f_2 \in \Theta(g \cdot h)$.
8. Si existe $\text{lím}_{n \to \infty} \frac{f(n)}{g(n)} = k$, según los valores que tome k:
   8.1 Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$.
   8.2 Si $k = 0$ entonces $\Theta(g) \neq \Theta(f)$.

Observaciones

La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad.

Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño, o cuando las constantes involucradas son demasiado grandes, etc.

Última sobre el tamaño de la entrada

¿Cual es la complejidad de multiplicar dos enteros?

- Depende de cual sea la medida del tamaño de la entrada.
- Podrá considerarse que todos los enteros tienen tamaño O(1), pero eso no será útil para comparar este tipo de algoritmos.
- En este caso, conviene pensar que la medida es el **logaritmo del número.**
- Si por el contrario estuviésemos analizando algoritmos que ordenan arreglos de enteros, lo que importa no son los enteros en sí, sino **cuántos** tengamos.
- Entonces, para ese problema, la medida va a decir que todos los enteros miden lo mismo.

Complejidad de algoritmos recursivos

¿Cómo calculamos la complejidad de los algoritmos recursivos?

- Dijimos: "El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente."
- Igual podemos pensarlo: hay que poder resolver las ecuaciones de recurrencia, por ejemplo:
  T(n)= n + T(n-1)

Intro a Estructuras de Datos

- Hasta ahora nos venimos preocupando mucho de la complejidad temporal sólo desde los algoritmos.
- Recordar el ejemplo de búsqueda lineal vs búsqueda binaria.
- Pero tambien podemos lograr mejorar la complejidad modificando cómo representamos los datos en la computadora.
  - Cambiando cómo se representamos algo en una misma estructura (lo vemos en una par de diapos)
  - Cambiando la estructura (lo que queda de la materia)
- Obviamente, cambiar estructuras, genera que cambien los algoritmos.

Conjunto

Recordemos el TAD conjunto:

```
TAD Conjunto<T> {
    obs elems: conj<T>

    proc conjVacío(): Conjunto<T>
        asegura {res.elems = ⟨⟩}

    proc pertenece(in c: Conjunto<T>, in e: T): bool
        asegura {res = true ↔ e ∈ c.elems}

    proc agregar(inout c: Conjunto<T>, in e: T)
        requiere {c = C0}
        asegura {c.elems = C0.elems ∪ ⟨e⟩}

    proc ...
}
```

- ¿Cómo podemos implementar (o sea, representarlo en la computadora) esto?
- Vamos a tener que elegir una estructura de datos (ustedes conocen solo Array)
- Y con eso, pensar los algoritmos

Conjunto

```
Modulo ConjSobreArray<T> implementa Conjunto<T> {
    elems: Array<T>

    proc conjVacío(): ConjSobreArray<T>
      **CODIGO**

    proc pertenece(in c: ConjSobreArray<T>, in e: T): bool
      **CODIGO**

    proc agregar(inout c: ConjSobreArray<T>, in e: T)
      **CODIGO**

}
```

¿El Array me permite representar todo el comportamiento del TAD Conjunto?

- ¿Cuántos elementos puede tener el TAD Conjunto?
- ¿Cómo hago para que mantener ese comportamiento con el Array?
- Más específicamente, si tenemos que pensar el algoritmo de conjVacio, ¿de qué tamaño inicializamos el Array? ¿Qué consecuencias va a traer esto a futuro (a nivel algoritmico, de complejidad, etc)?
- Ya vamos a ver más adelante más detalles de estos problemas
- Por ahora, simplifiquemos el problema

Conjunto Acotado

Redefinamos el TAD conjunto, agregándole una cota de elementos:

TAD ConjAcotado<$\mathbb{N}$> {
  obs elems: conj<$\mathbb{N}$>
  obs cota: $\mathbb{N}$

  proc conjVacío(in cota: $\mathbb{N}$): ConjAcotado<$\mathbb{N}$>
    asegura $\{res.elems = \langle\rangle \wedge res.cota = cota\}$

  proc pertenece(in c: ConjAcotado<$\mathbb{N}$>, in e: $\mathbb{N}$): bool
    asegura $\{res = true \leftrightarrow e \in c.elems\}$

  proc agregar(inout c: ConjAcotado<$\mathbb{N}$>, in e: $\mathbb{N}$)
    requiere $\{c = C_0 \wedge e \leq c.cota\}$
    asegura $\{c.elems = C_0.elems \cup \langle e\rangle\}$

  proc ...
}

- Y ahora, ¿Podemos trabajar más tranquiles con Arrays?

Conjunto Acotado sobre Array de Enteros

Modulo ConjAcotadoSobreArray<Int> implementa ConjAcotado<$\mathbb{N}$> {
  elems: Array<Int>
  cota: Int
  cant: Int

  proc conjVacío(in cota: Int): ConjAcotadoSobreArray<Int>
    Creo un Array de *cota* elementos
    Asigno $res.cota = cota$
    Asigno $res.cant = 0$

  proc pertenece(in c: ConjAcotadoSobreArray<Int>, in e: Int): bool
    Recorro el Array buscando a e.
    ¿Complejidad?

  proc agregar(inout c: ConjAcotadoSobreArray<Int>, in e: Int)
    Recorro el Array buscando a e
    Si no lo encuentro, lo agrego.
    Y Asigno $res.cant += 1$
    ¿Complejidad?
}

- Pensemos los algorimos
- ¿Podemos mejorar las complejidades de estos algoritmos?
- Vamos a ver que dependiendo el problema podemos elegir otras Estructuras
- Pero además, podemos representar el conjunto sobre Array de una forma diferente

Conjunto Acotado sobre Array de Bools

Modulo ConjActadoSobreBitArray<Bool> implementa ConjAcotado<$\mathbb{N}$> {
  elems: Array<Bool>
  cota: Int

  proc conjVacío(in cota: Int): ConjActadoSobreBitArray<Int>
    Creo un Array de *cota* elementos, todos en False

  proc pertenece(in c: ConjActadoSobreBitArray<Int>, in e: Int):
bool
    Me fijo el estado de la posición e.
    ¿Complejidad?

  proc agregar(inout c: ConjActadoSobreBitArray<Int>, in e: Int)
    Me fijo el estado de la posición e.
    Si no lo está (False), lo cambio.
    ¿Complejidad?
}

- Pensemos los algorimos
- ¡¡Logramos pasar todo a $O(1)$!!!

Bibliografía

- Data Structures and Algorithms. Alfred V. Aho, Jefrey D. Ullman, John E. Hopcroft. Addison-Wesley.
- Introduction to Algorithms, Second Edition. Thomas H.Cormen, Charles E. Leiserson and Ronald L. Rivest. MIT Press, 2001.
- Cualquier libro de Estructuras de Datos y/o Algoritmos