1 COMPLEJIDAD

En la práctica, el desarrollo de programas debe adelantarse teniendo en cuenta restricciones que imponen límites a la simple labor de encontrar una solución correcta para un problema. Con frecuencia es necesario considerar, por ejemplo, los equipos donde va a funcionar el programa, la capacidad en memoria principal y secundaria de los mismos, el tiempo que se tiene para el desarrollo, quien será el usuario final, con qué rapidez se necesita la información que el programa pueda dar, etc. Quizás las restricciones más importantes son las limitaciones de espacio de almacenamiento de los equipos (aunque cada día disponen de más capacidad) y la necesidad de respuestas rápidas a los requerimientos de los usuarios.

Un programa se concibe como la implantación, sobre una máquina real, de un algoritmo que resuelve un problema. Un algoritmo es un "programa abstracto", en el sentido de que se puede pensar que es ejecutable sobre una máquina abstracta ideal, cuyo funcionamiento es implantable en una máquina real. Algunas de las restricciones que se pueden concebir para los programas pueden relacionarse directamente con restricciones sobre los algoritmos. Por ejemplo, una restricción de tiempo para un programa puede traducirse, para un algoritmo correspondiente, en una restricción de número de operaciones básicas; una de espacio puede abstraerse en una sobre el tamaño de las estructuras de datos, etc.

En general, una restricción es una cota que indica la cantidad máxima disponible de un recurso de cálculo. Si se logran medir, de una forma natural, los recursos que -en abstracto- demanda un algoritmo, se tendrá una manera de juzgar el desempeño de cualquier programa que lo implante, en relación a las restricciones correspondientes. Una medida de un recurso abstracto deberá, por fuerza, ser independiente de los equipos, de los lenguajes de programación, del estilo de programación del programador y de todas aquellas cosas que dependan de una implantación específica.

Para un recurso `R`, la `R`-complejidad de un algoritmo se definirá como una medida de la cantidad de `R` que el algoritmo utiliza. En especial, se habla de complejidad temporal (espacial) cuando el el recurso en cuestión es el tiempo (espacio).

Analizar un algoritmo consiste en cuantificar los recursos -abstractos- que el algoritmo demanda. Como consecuencia, cada vez que se quiera analizar un algoritmo, se debe entender que lo que se quiere es medir su complejidad. En este capítulo se proponen medidas de complejidad para algoritmos, cuya utilidad está directamente relacionada con la posibilidad de medir las demandas de recursos reales de los posibles programas que los implanten.

1.1 PROBLEMAS Y SOLUCIONES ALGORÍTMICAS

En términos generales, un problema se plantea con una pregunta sobre un conjunto de datos. Puede haber más de una respuesta para una pregunta. Solucionar el problema consiste en presentar un método para encontrar respuestas para todas las posibles preguntas.

Por ejemplo, el problema de "encontrar un elemento en un arreglo" tiene como datos el arreglo y el elemento en cuestión. Como respuesta sirve cualquier índice en cuya posición en el arreglo se encuentre el elemento buscado. Un algoritmo de búsqueda lineal es una posible solución al problema.

Para efectos técnicos, un *problema* X se entenderá como un conjunto de parejas ordenadas $\langle p,r\rangle$, donde p es una *pregunta* y r una *respuesta* . P_X es el *conjunto de preguntas* de X, i.e., $P_X = \{p \mid \langle p,r\rangle \in X\}$. Tanto las preguntas como las respuestas son expresables en un lenguaje formal. Usualmente, un problema X se explica por comprensión, de la forma (D es un conjunto arbitrario: *datos*):

$$X = \{\langle p(x),r(x)\rangle \mid x \in D\}$$

Los problemas se pueden clasificar de acuerdo al carácter de la pregunta y la respuesta que se espera:

- *funcional* : para cada $p \in P_X$ existe un único r tal que $\langle p,r\rangle \in X$
- *de decisión* : las respuestas son "sí" o "no" (también: "no sé" e "indefinido")
- *de conteo* : las respuestas son números naturales
- *de búsqueda* : las preguntas plantean búsquedas en espacios determinados
- etc.

Un algoritmo A *soluciona* un problema X si, para cualquier $p \in P_X$, si A recibe como entrada p, produce como resultado un r, tal que $\langle p,r\rangle \in X$.

En la práctica no todas las preguntas de un problema son igualmente fáciles de resolver. Es decir, la dificultad de una pregunta es claramente dependiente de los datos. Para un problema X, el conjunto de preguntas P_X puede partirse en conjuntos disyuntos:

$$P_X = P_{X_0} \cup P_{X_1} \cup \ldots$$

Para una pregunta $p \in P_{X_i}$, se dirá que su *tamaño* es $|p| = i$. Se espera que la dificultad de resolver las preguntas sea proporcional a su tamaño. En el caso en que el problema se expresa por comprensión, la partición en clases de dificultad, o bien, el tamaño de las preguntas, está relacionado con la medida del parámetro que las determina.

Es conveniente utilizar una manera más coloquial -aunque igualmente rigurosa a la formal- para describir problemas. Por ejemplo:

BUSCAR UN ELEMENTO EN UN ARREGLO

Pregunta	:	Un arreglo `L: array[0..n-1] of D` Un elemento $x \in D$
Respuesta	:	Una variable $i \in 0..n-1$, tal que $L[i]=x$, o bien $i=n$, si $x \in L[0..n-1]$.

Es fácil observar que esta manera de plantear problemas está naturalmente emparentada con la de especificar problemas de programación, que deben ser resueltos con algoritmos. En la notación usual, el problema de programación se expresaría así:

BUSCAR UN ELEMENTO EN UN ARREGLO

```
Ctx  :   L: array[0..n-1] of  D, x:D
Pre  :   T
```

```
Pos  :    (i=n ∧ x∉L) cor (0≤i<n ∧ L[i]=x)
```

Cuando la precondición es **T**, se la puede omitir para abreviar la especificación.

Las preguntas para este problema están determinadas por el arreglo `L[0..n-1]` y el elemento `x`. En condiciones normales, el tamaño de una pregunta puede ser precisamente `n`, el tamaño del arreglo `L`. Es decir, un algoritmo que solucione este problema debe, en principio, enfrentar la misma dificultad para resolver dos preguntas que se refieran a arreglos de igual tamaño, sin importar su contenido o el elemento que se busca[1].

1.2 COMPLEJIDAD DE ALGORITMOS

La complejidad de un algoritmo es una medida de los recursos computacionales que el algoritmo demanda para su ejecución. Los ejemplos más representativos de recursos son el espacio y el tiempo, aunque deben ser entendidos de manera abstracta, en razón de que no se miden en términos de "bytes" o "segundos", porque esto no tiene sentido al nivel abstracto de un algoritmo. En general, los recursos que puede utilizar un algoritmo son los disponibles en la máquina abstracta sobre la que se ejecuta el algoritmo.

Supóngase un problema `X`, que se soluciona con un algoritmo `A`. Sea `R` un recurso computacional. Para una pregunta `p`, se define $R_A[p]$ como la cantidad de `R` que requiere `A` para resolver la pregunta `p`.

La partición de las preguntas en clases de dificultad tiene sentido si se cumple que

$$|p|=i,\ |q|=j,\ i<j \ \Rightarrow\ R_A[p] \le R_A[q]$$

i.e., las clases más complejas de resolver tienen subíndices más grandes. Usualmente esta condición se consigue fácilmente con un poco de sentido común.

Entonces se define, para `n∈`**`nat`**, la `R`-*complejidad de* `A` *en el peor caso* :

$$R_A(n) = (\sup p\ |\ |p|=n : R_A[p])$$

es decir, $R_A(n)$ es la mínima cota superior de la cantidad del recurso `R`, que el algoritmo `A` demanda en el peor caso .

Si, para `n∈`**`nat`**, se cuenta con una distribución de probabilidad q_n sobre P_{Xn}, se define, la `R`-*complejidad de* `A` *en el caso promedio* :

$$\overline{R}_A(n) = (+p\ |\ |p|=n : q_n(p)R_A[p])$$

Es común suponer que q_n es una distribución uniforme.

El recurso *tiempo* se denota `T`, y el recurso *espacio* se denota `S`. Con ellos se da lugar a los conceptos de *complejidad temporal* y *complejidad espacial* . Así:

---

[1] Sin embargo, debe tenerse cuidado en esta clase de simplificaciones. Si, por ejemplo, el arreglo es de números naturales (y `x` también representa un natural), el tamaño de éstos es, en rigor, arbitrario. Al implantar el algoritmo en una máquina real, la manipulación de los números debe considerar tamaños artbitrarios. Por ejemplo, una comparación (i.e., una resta) puede ser una operación "difícil", si los números son muy grandes.

$T_A(n)$: tiempo utilizado por A, en el peor caso,
$\overline{S}_A(n)$: espacio utilizado por A, en el caso promedio,
etc.

1.2.1 Operaciones básicas

Supóngase un algoritmo A que resuelve un problema X. Cuando A responde una pregunta p, la máquina abstracta sobre la que se ejecuta, efectúa una secuencia fija de operaciones

$$op_1,\ op_2,\ldots,\ op_k$$

Al implantar el algoritmo en una máquina real, estas operaciones abstractas deben tener como contraparte operaciones reales. Así mismo, los recursos que se consideren para el algoritmo deben tener recursos reales que les correspondan en la implementación.

Sean M1, M2 dos máquinas sobre las que se implanta el algoritmo A. Las notaciones $R_A^{M_i}[p]$, $R_A^{M_i}(n)$, $\overline{R}_A^{M_i}(n)$ tienen significados análogos a los ya anotados: los superíndices denotan la dependencia de la máquina Mi en la medida correspondiente.

Ahora bien, si se supone que la implantación de cada operación op_j requiere, cada vez que se ejecuta en la máquina Mi, a lo sumo una cantidad fija del recurso R^{Mi}, es fácil comprobar que:

i. Existe un número real $c>0$, tal que, para todo A y para toda $p \in P_X$:
$$R_A^{M_1}[p] \le c\ R_A^{M_2}[p]$$

ii. Existe un número real $c>0$, tal que, para todo A y para todo $n \in \mathbf{nat}$:
$$R_A^{M_1}(n) \le c\ R_A^{M_2}(n)$$

iii. Si q_n es una distribución de probabilidad sobre P_{X_n}, existe un número real $c>0$, tal que, para todo A y para todo $n \in \mathbf{nat}$:
$$\overline{R}_A^{M_1}(n) \le c\ \overline{R}_A^{M_2}(n)$$

Las constantes c de que tratan las anteriores afirmaciones no dependen de los algoritmos ni de las preguntas, sino de las máquinas mismas. Por este motivo, estas conclusiones sustentan aun más la conveniencia -y la posibilidad- de independizar de las máquinas reales los análisis de complejidad, puesto que el cambio de una máquina a otra se puede establecer con una especie de constante de proporcionalidad[2]. Por otra parte, una de las máquinas consideradas puede ser la máquina abstracta sobre la que se ha diseñado el algoritmo A. De esta manera, la complejidad abstracta es proporcional a la real.

La R-complejidad abstracta se considera, para cada pregunta p, como la suma de lo demandado de R por cada una de las operaciones op_j. Es posible que sólo algunas de estas operaciones demanden significativamente el recurso R; éstas se denominarán *operaciones básicas*, puesto que la omisión de las restantes no distorsiona de manera importante la estimación de la complejidad. Las

[2] Las hipótesis que se utilizaron (siempre la misma secuencia de operaciones y cada operación con un gasto acotado por una constante) son más o menos evidentes para máquinas determinísticas, pero pueden generalizarse sin dificultad en casos de no-determinismo.

operaciones no-básicas se limitan a efectuar "teneduría de libros" con respecto al recurso que se mide.

La simplificación que representa el hecho de considerar únicamente operaciones básicas es aun más inocua cuando se consideran las complejidades en peor caso y en caso promedio.

Ejemplo

A continuación se da una lista de problemas y, para cada uno, una selección razonable de operaciones básicas:

Problema	***Operaciones***
Encontrar un elemento `x` en una lista	Comparación de `x` con un elemento de la lista
Multiplicar dos matrices reales	Multiplicación de dos números reales o multiplicación y adición entre reales Conteo separado o ponderado
Ordenar una lista de números	Comparación de dos elementos de la lista
Recorrido de un árbol binario	Viajar de un nodo a otro (recorrer un arco)

□

En realidad, esta forma de medir es muy flexible. Las operaciones básicas pueden incluir algunas de las consideradas "teneduría de libros" o incluso el conjunto de instrucciones de máquina para un computador particular. Variando la selección de las operaciones básicas se varía el grado de precisión y abstracción del análisis a conveniencia.

Cuando no se puede uniformizar el conjunto de operaciones básicas para el conjunto de algoritmos que soluciona un tipo de problema, se restringe el estudio a clases de algoritmos determinadas por las operaciones que se puedan realizar sobre sus datos.

1.2.2 Ejemplo: análisis de complejidad temporal

Considérese el problema ya mencionado de buscar un elemento en un arreglo. Planteado como especificación para un algoritmo, se trata de encontrar una solución para:

BUSCAR UN ELEMENTO EN UN ARREGLO

```
Ctx  :    L: array[0..n-1] of  D, x:D
Pos  :    (i=n ∧ x∉L) cor (0≤i<n ∧ L[i]=x)
```

Es decir, al final de la ejecución, la variable `i` debe informar la posición del elemento buscado; cuando no se encuentre, terminará con un valor fuera de rango.

El algoritmo `BLI` (búsqueda lineal con incertidumbre) es una solución posible:

```
[ Ctx :   L: array[0..n-1] of  D, x:D
  {Pos :   (i=n ∧ x∉L) cor (0≤i<n ∧ L[i]=x) }
   i:= 0;
```

```
    do  (i≠n cand L[i]≠x)  → i:= i+1  od
]
```

La complejidad temporal se va a estimar contando el número de comparaciones de elementos de D. Este conteo resulta fácil, puesto que la única comparación presente en el texto del algoritmo es la que se encuentra en la guarda del ciclo; se pueden distinguir los siguientes casos:

- Caso C_k : x aparece en la posición k de L, para $0 \le k < n$. Requiere $k+1$ comparaciones.
- Caso C_n : x no aparece en L. Requiere n comparaciones.

En otras palabras ($p \in C_k$ quiere decir que la pregunta se refiere al caso C_k) :

$$T_{BLI}[p] = \begin{cases} k+1 & \text{, si } p \in C_k, \text{ para } 0 \le k < n, \\ n & \text{, si } p \in C_n. \end{cases}$$

De este modo:

$T_{BLI}(n) = n$ (alcanzado cuando se da C_{n-1} o C_n).

Para un análisis de caso promedio, la definición de $\overline{T}_{BLI}(n)$ exige contar con una distribución de probabilidad q_n sobre las preguntas de tamaño n (i.e., sobre el conjunto de las preguntas p, con $|p|=n$). No obstante, no siempre es necesario conocer al detalle esta distribución; así, para este ejemplo, será suficiente conocer los valores $q_n(C_k)$ (i.e., la probabilidad de que ocurra uno de los casos C_k). Supóngase una distribución uniforme para todos los casos en que el elemento aparece en el arreglo. Es decir, para cierto a, $0 \le a \le 1$:

$$q_n(C_k) = \begin{cases} \frac{a}{n} & \text{, si } 0 \le k < n \\ 1-a & \text{, si } k = n \end{cases}$$

Entonces se puede estimar la complejidad temporal promedio:

$$\begin{aligned}
&\overline{T}_{BLI}(n) \\
=\ & \\
&(+p \mid |p|=n : q_n(p) T_{BLI}[p]) \\
=\ & \\
&(+k \mid 0 \le k < n : (+p \mid p \in C_k : q_n(p) T_{BLI}[p])) + (+p \mid p \in C_n : q_n(p) T_{BLI}[p]) \\
=\ & \\
&(+k \mid 0 \le k < n : (+p \mid p \in C_k : q_n(p) * (k+1))) + (+p \mid p \in C_n : q_n(p) * n) \\
=\ & \\
&(+k \mid 0 \le k < n : (+p \mid p \in C_k : q_n(p)) * (k+1)) + (+p \mid p \in C_n : q_n(p)) * n \\
=\ & \\
&(+k \mid 0 \le k < n : q_n(C_k) * (k+1)) + q_n(C_n) * n \\
=\ & \\
&(+k \mid 0 \le k < n : \tfrac{a}{n} * (k+1)) + (1-a) * n \\
=\ & \\
&\tfrac{a}{n} * (+k \mid 0 \le k < n : (k+1)) + (1-a) * n \\
=\ & \\
&\tfrac{a}{n} * \tfrac{n*(n+1)}{2} + (1-a) * n \\
=\ & \\
&n - a\,\tfrac{n-1}{2}
\end{aligned}$$

Situaciones interesantes:

- $x \in L$: es decir, $a=1$. Entonces: $\overline{T}_{BLI}(n) = \frac{n+1}{2} \approx \frac{1}{2}\, n$

- $a=\frac{1}{2}$: en este caso: $\overline{T}_{BLI}(n) \approx \frac{3}{4} n$
- $x \notin L$: es decir, $a=0$. Entonces: $\overline{T}_{BLI}(n) = n$.

1.2.3 Ejercicios

1 Compruebe las afirmaciones que se hicieron en el texto con respecto a la comparación de desempeño de programas implantados en diferentes máquinas. Los supuestos son:
- Cada pregunta p se resuelve con una secuencia determinada de operaciones básicas.
- La implantación de cada operación básica requiere, cada vez que se ejecuta en la máquina M_i, a lo sumo una cantidad fija del recurso R^{Mi}.

a Existe un número real $c>0$, tal que, para todo A y para toda $p \in P_X$:

$$R_A^{M_1}[p] \leq c\ R_A^{M_2}[p]$$

b Existe un número real $c>0$, tal que, para todo A y para todo $n \in$ **nat**:

$$R_A^{M_1}(n) \leq c\ R_A^{M_2}(n)$$

c Si q_n es una distribución de probabilidad sobre P_{Xn}, existe un número real $c>0$, tal que, para todo A y para todo $n \in$ **nat**:

$$\overline{R}_A^{M_1}(n) \leq c\ \overline{R}_A^{M_2}(n)$$

2 ¿Qué tan restrictivos son los supuestos anotados en el ejercicio anterior?

3 En el ejemplo de 1.2.2, suponga n par. Calcule $\overline{T}_{BLI}(n)$, si se sabe que x sí está en $L[0..n-1]$, que se encuentra en la primera mitad el 90% de las veces y que la probabilidad de que un elemento del arreglo sea x es uniforme dentro de cada mitad.

1.3 ORDENES DE COMPLEJIDAD

Una vez que se tiene la posibilidad de medir la complejidad de un algoritmo, es deseable poder comparar dos soluciones A_1, A_2 a un mismo problema X. Es claro que si, para cada pregunta p se tiene que $R_{A_1}[p] \leq R_{A_2}[p]$, el algoritmo $A1$ es mejor que el algoritmo $A2$. Algo parecido se podría afirmar si, para $n \in$ **nat**, se tiene que $R_{A_1}(n) \leq R_{A_2}(n)$, puesto que demanda menos recursos, en el peor caso. Sin embargo, afirmaciones universales de esta naturaleza se dan, en la práctica, en raras ocasiones. En cambio, lo que se usa es comparar el comportamiento asintótico de las complejidades de los algoritmos, cuando n, el tamaño de la pregunta, crece arbitrariamente.

Se dice que A_1 es (*asintóticamente*) R*-mejor* que A_2, si se cumple que

$$\lim_{n \to \infty} \frac{R_{A_1}(n)}{R_{A_2}(n)} = 0.$$

No debe olvidarse que, para problemas pequeños, puede suceder que A_2 tenga un mejor desempeño que el de A_1. Por ejemplo (cf. [Mel77]), supónganse cuatro algoritmos A,B,C,D que resuelven un problema P con complejidades temporales[3]

$$T_A(n) = 1000n, \quad T_B(n) = 200n \log n, \quad T_C(n) = 10n^2, \quad T_D(n) = 2^n.$$

[3] $\log n$ denota el logaritmo en base 2.

Las complejidades pueden contar, por ejemplo, el número de ciertas operaciones básicas que cada método efectúa. Entonces, el algoritmo `B` es el más rápido para `n=1`, el algoritmo `D` es el más rápido para $1<n\leq 9$, el algoritmo `C` es el más rápido para $10\leq n<100$ y el algoritmo `A` es el más rápido para $n\leq 100$.

Supóngase que cada operación básica demora un milisegundo en una máquina fija `M`, i.e.,

$$T_A^M(n) = 1000n \text{ ms}, \qquad T_B^M(n) = 200n \log n \text{ ms},$$
$$T_C^M(n) = 10n^2 \text{ ms}, \qquad T_D^M(n) = 2^n \text{ ms}$$

y que se dispone de una hora de cálculo para resolver el problema en cuestión. Utilizando el algoritmo `A` (`B`, `C`, `D`) se pueden resolver todas las preguntas hasta de tamaño `3600` (`1680`, `600`, `22`). Si estos tamaños máximos de problema resultaran insuficientes, habría dos posibilidades: conseguir una máquina más rápida o utilizar un algoritmo mejor.

Si se consigue una máquina `M1`, diez veces más veloz que `M`, las nuevas complejidades reales son:

$$T_A^{M_1}(n) = 100n \text{ ms}, \qquad T_B^{M_1}(n) = 20n \log n \text{ ms},$$
$$T_C^{M_1}(n) = n^2 \text{ ms}, \qquad T_D^{M_1}(n) = \frac{2^n}{10} \text{ ms},$$

de manera que ahora los algoritmos pueden resolver problemas de tamaños `36000`, `13155`, `1897` y `25`, en el mismo tiempo que en la primera máquina.

Es evidente que los incrementos en la velocidad conllevan incrementos muy pequeños en el rendimiento de algoritmos ineficientes (como el `D`), mientras que el cambio a un algoritmo asintóticamente mejor (de `D` a `C`, o de `C` a `A`) reporta mejores dividendos que el cambio de una máquina lenta por otra comparativamente más rápida. Este tipo de argumentos sustenta el hecho de llamar "mejor" a un algoritmo que asintóticamente lo es, aunque si el tamaño de los problemas no es arbitrariamente grande, es posible que se deban tener en cuenta otras consideraciones.

1.3.1 Las notaciones O y θ

En la práctica resulta usualmente complicado establecer con precisión una función de complejidad R_A. En cambio, y puesto que se está interesado en el comportamiento asintótico de los algoritmos, se prefiere estimar el orden de magnitud de la complejidad, con lo que se consigue una cota superior -para problemas grandes- de la complejidad, cualquiera que ella sea. Para esto es importante la notación `O`.

Notación

Para dos funciones $f: \mathbf{nat} \to \mathbf{R^+}$ y $g: \mathbf{nat} \to \mathbf{R^+}$, se dice que

- `f` es *de orden menor o igual que* `g`, y se escribe

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

  si y sólo si existen $c\geq 0$, $n_0 \in \mathbf{nat}$ tales que, para $n\geq n_0$: $f(n) \leq c\, g(n)$.

- `f` es *de orden igual a* `g`, y se escribe

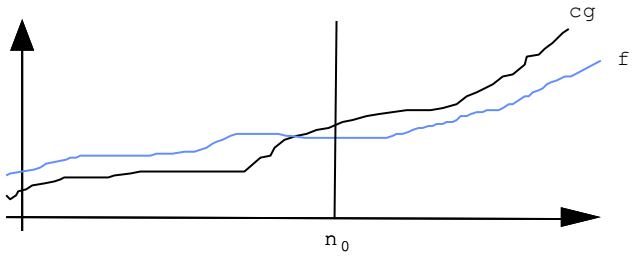
  $f = \theta(g)$ o bien $f(n) = \theta(g(n))$

  si y sólo si $f = O(g)$ y $g = O(f)$.

□

La relación $. = O(.)$ es un orden parcial sobre las funciones en $(\mathbf{nat} \to \mathbf{R}^+)$. Por su parte, $. = \theta(.)$ es una relación de equivalencia sobre el mismo conjunto de funciones.

Cuando $f = O(g)$, las gráficas de las funciones son de la forma:



Algunos ejemplos:

$\frac{5}{2} = O(n^2)$, $\frac{n^2}{2} = O(307n^2)$, $307n^2 = O(\frac{n^2}{2})$.

Nótese que, además, $\frac{n^2}{2} = \theta(307n^2)$. En cambio, es falso que $\frac{5}{2} = \theta(n^2)$.

Los siguientes teoremas establecen reglas útiles para el cálculo de órdenes:

Teorema A

Si f, g, r, s son funciones de $(\mathbf{nat} \to \mathbf{R}^+)$, entonces

- Para todo $c>0$: $cf = O(f)$
- *(Regla de sumas)* $f+g = O(\max(f,g))$
- *(Regla de productos)* Si $f = O(r)$ y $g = O(s)$, entonces $fg = O(rs)$

Demostración:
Se omite.

□

Teorema B

Si f, g funciones de $(\mathbf{nat} \to \mathbf{R}^+)$, para las que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

- Si $c \geq 0$, entonces $f = O(g)$
- Si $c>0$, entonces $f = \theta(g)$.

Demostración:
Se omite.

□

Algunos ejemplos del uso de las notaciones O y θ:

- $2n^5 = O(n^5)$

- $2n^5 + 4n = O(2n^5)$
- $2n^5 + 4n = O(n^5)$
- $(2n^5 + 4n)^2 = O(n^{10})$
- $(n+1)^2 = \theta(3n^2)$
- $\frac{n^2 + 5n + 7}{5n^3 + 7n + 2} = \theta(\frac{1}{n})$.

1.3.2 Complejidad asintótica de problemas

El álgebra de las notaciones O y θ facilita el manejo y comparación de complejidades asintóticas. Es corriente referirse más al orden de complejidad asintótica de los algoritmos que a la complejidad medida exactamente. Así, continuando con el ejemplo del principio de 1.3:

$$T_A(n) = O(n), \quad T_B(n) = O(n \log n), \quad T_C(n) = O(n^2), \quad T_D(n) = O(2^n).$$

Las conclusiones que afloraron de la discusión de utilizar una máquina más rápida se aprecian más sencillamente en términos de la notación O. Así, lo que se proponía entonces era una mejora que significaría la rebaja de los tiempos en un factor constante; ahora es claro que esto no afecta el orden asintótico de las complejidades.

Con base en las definiciones de complejidad de algoritmos se pueden establecer nociones interesantes para los problemas. Se dice que un problema X tiene orden de R-complejidad $O(f)$, si se puede mostrar un algoritmo A que lo resuelve, para el que $f = O(R_A)$. Es decir, se establece una cota superior para la complejidad de los algoritmos que solucionan el problema.

Se dice que un problema X tiene orden de R-complejidad $\Omega(f)$, si para cualquier algoritmo A que lo solucione, se tiene que $R_A = O(f)$. Esta clase de afirmaciones es difícil de establecer, porque se trata de aserciones universales sobre los posibles algoritmos que solucionen el problema. El establecimiento de tales cotas inferiores para el orden de complejidad de los algoritmos que resuelven un problema se ha logrado, no obstante, para algunas clases importantes de problemas.

1.3.3 Taxonomía de órdenes de complejidad

Tanto para algoritmos como para problemas, en la literatura se distinguen clases de órdenes de complejidad. Así, se dice que un complejidad R_A es (de orden)

- *Constante* : si $R_A = O(1)$
- *Logarítmica* : si $R_A = O(\log n)$
- *Polilogarítmica* : si existe una constante $k>0$, tal que $R_A = O(\log^k n)$
- *Lineal* : si $R_A = O(n)$
- *Polinomial* : si existe una constante $k>0$, tal que $R_A = O(n^k)$
- *Exponencial* : si existe una constante $k>0$, tal que $R_A = O(2^{n^k})$

Nótese que esta clasificación no es, de ninguna manera, exhaustiva para hacer las distinciones teóricas que las definiciones sí posibilitan (v.gr., cómo clasificar $R_A = O(\log \log n)$?). Obsérvese que las clases están contenidas unas en otras, en el orden de la enumeración, i.e., toda complejidad constante es logarítmica, ésta es polilogarítmica, etc. En otras palabras, el orden de enumeración es compatible con el orden parcial inducido por la notación O. Sin embargo, como clases de equivalencia de θ, son disyuntas. Por ejemplo, como $\log n = O(1)$ es falso, la clase logarítmica no coincide con la constante.

1.3.4 Ejercicios

1 Describa gráficamente el comportamiento de f y g, cuando $f = \theta(g)$.

2 Demuestre los teoremas A y B de 1.3.1.

3 Las siguientes proposiciones puede ser verdaderas o falsas. Júzguelas y explique sus respuestas:

a $(n^2 + 3n + 1)^3 = \theta(n^6)$ **b** $e^{1/n} = O(1)$

c $n^3 (\log \log n)^2 = O(n^3 \log n)$ **d** $(+j \mid 0 \le j \le n : 1) = \theta(n)$

e $(+j \mid 1 \le j \le n : \frac{1}{n^2}) = \theta(n)$ **f** $(+j \mid 1 \le j \le n : \frac{1}{j^2}) = \theta(1)$

4 Pruebe que si $f = O(g)$ y $g = O(h)$, entonces $f = O(h)$.

5 Sean a,b números reales, tales que $0<a<b$. Pruebe que $n^a = O(n^b)$, pero que es falso que $n^b = O(n^a)$.

6 Sea P un polinomio sobre n, con coeficientes reales, de grado k. Muestre que $P = O(n^k)$.

7 Establezca clases de equivalencia, según θ, para las siguientes funciones. Ordene las clases de menor a mayor complejidad, según O:

n, $\sqrt{n} + \log n$, n^3, 2^n, $\log n$, $n \log n$,
$n - n^3 + 7n^5$, $\log^2 n$, $n^2 + \log n$, $n!$, $5n^2$.

¿Puede clasificar cada función según la taxonomía de 1.3.3?
AYUDA: Para efectos de comparar $n!$, utilice la aproximación de Stirling:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + \frac{1}{288\,n^2} + O\left(\frac{1}{n^3}\right)\right] \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

1.4 MODELOS DE MÁQUINAS

Aunque las medidas de complejidad que se han introducido pretenden ser independientes de las máquinas que implementarán los programas, los algoritmos se suponen ejecutables sobre máquinas abstractas que, aunque ideales, de unas a otras pueden tener diferencias esenciales de potencialidad computacional. Es el caso de comparar máquinas secuenciales determinísticas con otras que permitan paralelismo o no-determinismo.

La siguiente clasificación sigue fundamentalmente a [Joh90]. Se dará una descripción *grosso modo* de las componentes de cada modelo sus capacidades de cómputo.

- *Máquinas de Turing determinísticas (MTD)*
  Una MTD es una máquina con tres cintas semi-infinitas, i.e., de almacenamiento secuencial acotado por un extremo: una cinta de *entrada* (sólo lectura), una de *salida* (sólo escritura) y una de *trabajo*. Cada cinta tiene celdas subindicadas con números naturales y la máquina tiene una cabeza sobre cada cinta que puede moverse hacia adelante una celda en las cintas de entrada y de salida, mientras que en la de trabajo puede, además retroceder una celda. De acuerdo a lo

que esté observando con las cabezas en las cintas de entrada o trabajo, la máquina toma una decisión como mover alguna cabeza o escribir algo.

La máquina resuelve un problema cuando una descripción de la pregunta se le plantea en las celdas más a la izquierda de la cinta de entrada y, comenzando la computación con todas las demás celdas en blanco, la máquina se mueve y se detiene eventualmente con una descripción de la respuesta en la parte más izquierda de la cinta de salida. El *tiempo* de la computación es el número de pasos que la máquina debe dar antes de detenerse con la respuesta. El *espacio* es el número de celdas en la cinta de trabajo que fueron visitadas por la cabeza respectiva durante la computación.

Obsérvese que el espacio demandado es el adicional a las descripciones de la pregunta y de la respuesta. Por otra parte, el tiempo está siempre acotado inferiomente por el tamaño de la entrada.

- *Máquinas de Turing no-determinísticas (MTND)*
  Una MTND es igual a una MTD, con la posibilidad suplementaria de tomar más de una decisión -mover una cabeza o escribir algo- a partir de una misma configuración de sus componentes. El número de selecciones factibles es, no obstante, finito. La decisión se toma de manera no-determinística, i.e., arbitrariamente. Si una MTND se enfrenta en dos ocasiones a una misma configuración, la secuencia de operaciones efectuadas puede diferir de una a otra ejecución.

  Las MTNDs se utilizan sólo para resolver problemas de decisión, de suerte que la cinta de respuesta puede limitarse a una celda. Si la cinta de respuesta contiene un "sí", debe entenderse que hay una forma de calcular la respuesta "adivinando" las decisiones correctas en las selecciones no-determinísticas. Si la respuesta es "no", significa que no hay ninguna posible computación que termine en una respuesta positiva. Es fácil generalizar estas definiciones, si se pretende considerar otra clase de posibles respuestas, como "indefinido".

  Una computación puede visualizarse como un árbol de ramificación finita, cuyos nodos son configuraciones o estados de computación y cuyos arcos representan las decisiones arbitrarias que pueden tomarse en cada configuración. Como *tiempo* se considera la altura de este árbol, que equivale a estimar el tiempo por el de la posible ejecución más larga. Como *espacio*, la cantidad máxima de espacio de la cinta de trabajo que una posible ejecución, i.e., una rama del árbol, puede utilizar.

- *Máquinas de Turing con otros criterios de aceptación*
  Hay una variedad de convenciones de salida para máquinas de Turing que producen árboles de ejecución como computaciones.

  La respuesta de una *máquina de Turing probabilística* se considera afirmativa, si por lo menos la mitad de las configuraciones de parada han escrito "sí" en la cinta de salida; se considera negativa, si ninguna de las configuraciones de parada tiene "sí" en la salida; y, en otro caso, se considera indefinida.

  Una *máquina de Turing contadora* es aplicable a problemas de conteo y da como respuesta el número de configuraciones de parada que han escrito "sí" en la salida.

- *Máquinas de Turing con oráculo (MTO)*
  Son máquinas de Turing (MTD, MTND) con una cinta adicional semi-infinita, la *cinta del oráculo* , que alterna entre modos de sólo-lectura y sólo-escritura. Asociado con una TMO hay un problema Y que el oráculo puede resolver en un paso de trabajo. Cuando el oráculo está en modo de sólo-escritura, la MTD puede entrar, en cualquier momento, en un estado de pregunta, en cuyo caso el contenido y de la cinta del oráculo es automáticamente remplazado por un valor

b, de modo que $\langle y,b\rangle \in Y$, y la cinta se cambia a modo de sólo-lectura. Una vez que su contenido es leído, la máquina puede, en un solo paso, borrar su contenido y volver al modo de sólo-escritura. Preguntar a un oráculo es equivalente a llamar una subrutina que resuelve el subproblema Y, excepto por el hecho de que no se cuentan los recursos consumidos por la subrutina.

Para estimar el *espacio*, debe adicionarse a lo consumido por la cinta de trabajo, el máximo espacio utilizado en la cinta del oráculo. Un recurso interesante de cuantificar es el número de llamadas al oráculo, sobre todas las posibles computaciones de un algoritmo.

- *Máquinas paralelas de acceso aleatorio (PRAM)*
  Una PRAM (del inglés: *parallel random access machine*) es una máquina con un número arbitrario -pero finito- de procesadores, que además está equipada con tres conjuntos adicionales de registros de memoria: uno de celdas de sólo-lectura (entrada), uno de celdas de sólo-escritura (salida) y uno de celdas de lectura-escritura (trabajo). Los requisitos de entrada/salida son análogos a los de las MTDs. El paralelismo está restringido de manera que, aunque dos procesadores pueden leer simultáneamente el contenido de una misma celda, no está permitido que escriban al tiempo sobre una misma celda.

  El *espacio* se mide numerando los registros de trabajo r_1, r_2, ..., y determinando el máximo k, tal que durante la computación algún procesador utilice el registro r_k.

  El *tiempo* se estima de una forma más complicada. Se supone que los procesadores trabajan sincrónicamente, un paso cada vez. El tiempo, para un paso dado, es el máximo, sobre todos los procesadores, del costo semilogarítmico de la operación realizada por cada procesador. Más exactamente, el costo semilogarítmico de una operación que coloca un número N en el registro i, cuando la entrada tiene tamaño n, es $\lceil 1+ \frac{\log N + \log i}{\log n} \rceil$.
  Es importante considerar como recurso el número de procesadores utilizados.

- *Máquinas de acceso aleatorio (RAM)*
  Una RAM (del inglés: *random access machine*) es una PRAM con un solo procesador. Es un modelo de máquina muy utilizado en la literatura y de fácil implantación sobre máquinas reales. En ocasiones se distinguen los registros acumuladores y la memoria principal[4] y el tiempo de acceso a las dos clases de memoria se estima en forma diferente.

En lo sucesivo, se podrá suponer que se razona para máquinas RAM, secuenciales, determinísticas, con un procesador. Como ya se anotó, una MTD es fácilmente asimilable a una RAM.

1.5 CÁLCULO DE COMPLEJIDAD DE ALGORITMOS

El lenguaje que se utilice para expresar los algoritmos no debe ser en ningún sentido especial o importante. De hecho, lo deseable es contar con un lenguaje algorítmico de propósito general, tanto fácilmente comprensible como naturalmente traducible a un lenguaje de programación real, i.e., implantable en máquinas reales.

Con los anteriores propósitos en mente, utilizaremos a GCL (cf. [Car91]) como lenguaje algorítmico cuando se quiera ser formal en las descripciones. No obstante, en ocasiones una clara explicación en lenguaje natural establece con la precisión deseable -es decir, que facilite los análisis de complejidad- los métodos que se deseen describir.

[4] También llamada memoria RAM, pero esta vez, del inglés: *random access memory*.

Como ya se anotaba en apartados anteriores, usualmente es suficiente calcular el orden asintótico de la complejidad de un algoritmo, en lugar de establecer la complejidad como tal con exactitud. De hecho, a menos que se diga explícitamente lo contrario, en adelante se entenderá, al pretender calcular complejidades, que se espera establecer una clase de complejidad, en el sentido de 1.3.3, a la que pertenezca la función de complejidad en cuestión.

En GCL se pueden considerar las complejidades espacial y temporal. La estimación de la demanda de espacio de un algoritmo se puede establecer a partir de las declaraciones de variables y, en general, de estucturas de datos. Como hay estructuras que pueden variar de tamaño con la ejecución, esta eventualidad debe tenerse en cuenta si se cuenta el número de veces que se ejecuten instrucciones de pedido y/o liberación de espacio. Tales conteos están directamente relacionados con la complejidad temporal del programa o de las partes de éste que incluyen los cambios espaciales anotados.

1.5.1 Complejidad temporal de instrucciones GCL

A continuación se establece una forma de estimar la complejidad temporal `T(I)` para instrucciones `I` expresables en GCL. Se han incluido algunas instrucciones que usualmente no están en el repertorio de GCL, aunque son fácilmente implantables en él.

Las estimaciones corresponden al análisis del peor caso. Cuando la estimación corresponde a una suma de funciones, pueden establecerse cotas más sencillas, aunque a veces "demasiado grandes", aplicando la regla de sumas del Teorema A de 1.3.1.

- *Acción nula*
  $T(\textbf{skip}) = 0$

- *Abortar (terminación anormal)*
  $T(\textbf{abort}) = 0$

- *Asignación*
  $T(x := e) = O(T(e))$
  Se entiende que $T(e)$ es una estimación del tiempo de la evaluación de la expresión e. Normalmente es acotado por una constante, i.e., $T(e) = O(1)$.

- *Secuenciación*
  $T(S_1;S_2) = O(T(S_1) + T(S_2))$
  Si se usa la regla de sumas:
  $T(S_1;S_2) = O(\max(T(S_1), T(S_2))$.

- *Condicional múltiple*
  $IF \cong \textbf{if}\ B_1 \to S_1\ [] \ \ldots\ [] \ B_r \to S_r\ \textbf{fi}$
  La semántica operacional que se tiene en mente consiste en evaluar todas las condiciones y elegir un comando guardado por una de las que son verdaderas. Entonces:
  $T(IF) = O((+i|\ 1 \le i \le r:\ T(B_i)) + (\max i|\ 1 \le i \le r:\ T(S_i)))$
  La regla de sumas permite afirmar que, también:
  $T(IF) = O((\max i|\ 1 \le i \le r:\ T(B_i)) + (\max i|\ 1 \le i \le r:\ T(S_i)))$

- *Condicional simple*
  $ITE \cong \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
  En términos de efectos netos, i.e., no necesariamente de una semántica operacional, `ITE` es simulable en GCL:
  $ITE \cong \textbf{if}\ B \to S_1\ []\ \neg B \to S_2\ \textbf{fi}$
  Con la estimación anterior, se tendría que (suponiendo que $T(B) = T(\neg B)$):

$T(ITE) = O(\max(T(B)+T(S_1), T(B)+T(S_2))$
y por la regla de sumas:
$T(ITE) = O(\max(T(B), T(S_1), T(S_2)))$.
En una implantación real no es necesario evaluar $\neg B$, pero esta consideración no afecta el orden de complejidad de la cota señalada, puesto que, en rigor, debería escribirse:
$T(ITE) = O(T(B)+\max(T(S_1), T(S_2))$

- *Iteración simple*
  $WHILE[B,S] \cong$ **do** $B \rightarrow S$ **od**
  Si t es una cota para el número de iteraciones que se efectúan al ejecutar $WHILE[B,S]$:
  $T(WHILE[B,S]) = O(t * \max(T(B), T(S))$

- *Iteración múltiple*
  $DO \cong$ **do** $B_1 \rightarrow S_1$ [] ... [] $B_r \rightarrow S_r$ **od**
  Es conocido que
  $DO \cong$ **do** $BB \rightarrow IF$ **od** $\cong WHILE[BB,IF]$
  donde $BB \equiv (\vee i \mid 1 \leq i \leq r: B_i)$,
  $IF \cong$ **if** $B_1 \rightarrow S_1$ [] ... [] $B_r \rightarrow S_r$ **fi**
  Si t es una cota para el número de iteraciones que se efectúan al ejecutar $WHILE[BB,IF]$:
  $T(DO) = O(t * (+i: 1 \leq i \leq r: T(B_i)+T(S_i)))$
  o bien, por la regla de sumas:
  $T(DO) = O(t * (\max i: 1 \leq i \leq r: T(B_i)+T(S_i)))$

- *Iteración controlada*
  $FOR \cong$ **for** $x \in C \rightarrow S(x)$ **rof**
  Se espera que C sea un conjunto enumerable, i.e., $C = \{c_1, c_2, \ldots\}$. Usualmente, C es un intervalo de números enteros, para el que se establece un orden de recorrido, v.gr.,
  for $i := 1$ **to** $r \rightarrow S(i)$ **rof**, o bien, **for** $i := 1$ **downto** $r \rightarrow S(i)$ **rof**.
  FOR puede simularse:
  $FOR \cong i := 1;$ **do** $i \neq |C|+1 \rightarrow S(c_i); i := i+1$ **od**
  Entonces:
  $T(FOR) = O((+i: 1 \leq i \leq r: T(S(c_i)))$
  Con la regla de sumas:
  $T(FOR) = O(|C| * (\max i: 1 \leq i \leq r: T(S(c_i)))$

- *Llamada a procedimiento*
  Para un procedimiento declarado:

  proc p (**x; var y**) $\lceil \ldots \lceil B \rfloor$
  $T(p(a,b)) = O(\max(T(a), T(b), T(B))$.

- *Llamada a una función*
  Para una función declarada:

  funct f (**x**) $\lceil \ldots \lceil B \rfloor$
  y una instrucción $L(f(\mathbf{a}))$, que menciona una llamada $f(\mathbf{a})$ a la función f. Si el costo de una instrucción $L(\xi)$, donde ξ es una constante, se estima con $O(T_L)$, se tendrá que:
  $T(L(f(\mathbf{a}))) = O((\max(T(a), T(B), T_L)))$.

1.5.2 Ejemplo: complejidad temporal de un ordenamiento burbuja

El *ordenamiento burbuja* (inglés: *bubble sort*) es un algoritmo[5] (las etiquetas de las instrucciones se incluyen para facilitar la explicación del análisis):

```
[Ctx: a: array[0..n-1] of int
   {Pre: a = A}
   {Pos: a = ord_(A)}        /* ord_(x) : x ordenado ascendentemente */
   S1:    for i:=0 to n-2 →
              S2:  for j:= n-1 downto i+1 →
                       S3:  if a[j-1]] > a[j]  →   S4:  temp:= a[j];
                                                   S5:  a[j]:=a[j-1];
                                                   S6:  a[j-1]:= temp
                            [] a[j-1]] ≤ a[j]  →   skip
                            fi
                   rof
          rof
]
```

Supóngase que las asignaciones y las comparaciones tienen costo `O(1)`.

El análisis se trabaja "de dentro hacia fuera". En primer lugar, las tres asignaciones más internas tienen un costo constante, es decir:

$T(S4; S5; S6) = O(1)$

El costo de la instrucción condicional también puede acotarse por una constante:

$T(S3) = O(1)$

El `FOR` interior ejecuta `n-1-i` veces su cuerpo `S3`, de modo que:

$T(S2) = O(n-1-i)$

Finalmente, el costo del `FOR` exterior puede estimarse por

$$T(S1) = O((+i \mid 0 \le i \le n-2 : n-1-i))$$
$$= O((+i \mid 1 \le j \le n-1 : j))$$
$$= O(\frac{n(n-1)}{2})$$
$$= O(n^2).$$

1.5.3 Complejidad de algoritmos recurrentes

Cuando se tienen procedimientos o funciones recurrentes, las estimaciones de costos de llamadas derivadas en 1.5.2 expresan, en general, ecuaciones de recurrencia. Por ejemplo, supongamos una definición para la función factorial, de la forma:

[5] Mientras no importe la complejidad espacial, se omite la declaración detallada de las variables utilizadas, en aras de la simplicidad.

```
funct fact (r: nat): nat
{Pos: fact = r!}
⌈  if     r=0  →   fact:= 1
   ▯      r>0  →   fact:= r*fact(r-1)
   fi
⌋
```

¿Qué complejidad temporal tiene una llamada `fact(n)` ?

Si denotamos con $T_f(n)$ el costo de una tal llamada, y se cuentan únicamente las multiplicaciones que se efectúan, deberá cumplirse que:

$$T_f(n) = 0 \quad , \text{ si } n=0$$
$$= 1 + T_f(n-1) \quad , \text{ si } n>0$$

La relación anterior establece una recurrencia para $T_f(n)$, puesto que expresa su valor en términos del valor de $T_f(n-1)$. En rigor, esto basta como respuesta a la pregunta formulada, porque, para un n arbitrario pero fijo, tal información se puede utilizar -recurrentemente!- un número finito de veces, hasta encontrar explícitamente la respuesta particular.

Si se calculan los primeros valores de $T_f(n)$, se observa una regularidad evidente:

n	$T_f(n)$
0	0
1	1
2	2
...	...

Es fácil comprobar, por inducción, que:

$$T_f(n) = n, \quad \text{para } n \geq 0.$$

Esta formulación "cerrada" de $T_f(n)$ es preferible a la recurrente, ya que permite un cálculo directo de la complejidad en términos de los argumentos de la llamada. Sin embargo, no siempre se tiene la suerte de poder intuir, a partir de algunos ejemplos, la formulación cerrada que, además, se deba probar por métodos inductivos. En el capítulo 2 se exponen algunos métodos algebraicos para resolver algunas clases de ecuaciones de recurrencia, i.e., encontrar formulaciones no recurrentes para las funciones implícitamente entendidas con la recurrencia.

1.5.4 Ejercicios

1 Suponga que GCL se extiende con una instrucción `REPEAT`, operacionalmente equivalente a una instrucción de la forma

repeat S **until** B

i.e., "repetir la instrucción `S` hasta que se cumpla la condición `B`". Estime el orden de complejidad de `T(REPEAT)`.

2 Considere el siguiente algoritmo para evaluación de polinomios:

```
[ Ctx: a: array[0..n] of int; x: int;
  {Pre: (∀j| 0≤j≤n: a[j] = A_j) }
  y,xi:= a[0],1;
  for i:= 1 to n →
    xi:= x*xi;
    y:= y + a[i]*xi
  rof
  {Pos: y = (+j| 0≤j≤n: A_j*x^j⟩ }
]
```

Considere como operaciones básicas la suma y la multiplicación de números enteros. Suponga constantes (aunque no necesariamente iguales) los costos de estas operaciones.

a Calcule la complejidad temporal exacta del algoritmo en términos de n (el grado del polinomio).

b Analice la complejidad temporal del algoritmo en el caso promedio, con los siguientes supuestos adicionales:
- a lo sumo uno de los coeficientes A_j es 0,
- las multiplicaciones por 0 no cuestan (las demás: costo constante),
- la probabilidad de que algún coeficiente A_j sea 0 y los demás no nulos es p.

3 **a** Diseñe un algoritmo que calcule el mínimo de un arreglo de n números y estime su complejidad temporal en el peor caso.

b Diseñe un algoritmo que ordene ascendentemente un arreglo de n números, utilizando el algoritmo de **a** (i.e., ordenamiento por mínimos). Estime la complejidad temporal en el peor caso.

4 Considere la siguiente función recurrente para calcular cuadrados de números naturales:

```
funct cuad (n: nat): nat
{Pos: cuad = n^2 }
⌈ if   n=0            →   cuad:= 0
  []   impar(n)       →   cuad:= cuad(n-1) + 2*n - 1
  []   n>0 ∧ par(n)   →   cuad:= cuad(n-2) + 4*n - 4
  fi
⌋
```

Analice la complejidad temporal para una llamada `cuad(k)`. Considere como operaciones básicas:

a Asignaciones.

b Sumas y restas.

BIBLIOGRAFÍA

[Car91] Cardoso, R., *Verificación y desarrollo de programas, Ediciones Uniandes -Ecoe,* 2a. impresión revisada, 1993.

[Cor90] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Introduction to algorithms, MIT Press, 1990.

[Joh90] Johnson, D.S., “A Catalog of Complexity Classes”, Cap. 2 en Handbook of Theoretical Computer Science, J. van Leeuwen (editor), Elsevier Science Publishers B.V., 1990.

[Mel77] Melhorn, K., Effiiziente Algorithmen, Teubner Studienbücher - Informatik, 1977.

[Par95] Parberry, I., Problems on algorithms, Prentice-Hall, 1995.