APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 1: ANÁLISIS DE ALGORITMOS

Por favor, reportar errores, omisiones y sugerencias a dfridlender@gmail.com
Consultar la página de la materia:
http://cs.famaf.unc.edu.ar/wiki/doku.php?id=algo2:main

Observaciones Preliminares

En las materias **Introducción a los Algoritmos** y **Algoritmos y Estructuras de Datos I** el énfasis estaba puesto en **qué** hace un programa. Se especificaba detalladamente las pre- y pos- condiciones que el programa debía satisfacer y se derivaba un programa que satisfaciera dicha especificación.

En **Algoritmos y Estructuras de Datos II**, el énfasis estará puesto en **cómo** resuelve el programa el problema especificado. Desarrollaremos instrumentos que nos permitan comparar diferentes programas que resuelven un mismo problema.

Uno de los aspectos que es importante considerar al comparar algoritmos es el referido a los recursos que el programa necesita para ejecutarse: tiempo de procesamiento, espacio de memoria, tiempo de utilización de un dispositivo. El estudio de la necesidad de recursos de un programa o algoritmo se llama **análisis del algoritmo** y lo que dicho análisis determina es la **eficiencia** del mismo.

Ejemplos motivadores. Considere las siguientes preguntas:

1. Un pintor demora una hora y media en pintar una pared de 3m de largo. ¿Cuánto demorará en pintar una de 5m de largo?
2. Un pintor demora una hora y media en pintar una pared cuadrada de 3m de lado. ¿Cuánto demorará en pintar una de 5m de lado?
3. Si lleva cinco horas inflar un globo aerostático esférico de 4m de diámetro, ¿cuánto llevará inflar uno de 8m de diámetro?
4. Un bibliotecario demora un día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto demorará en ordenar una con 2000 expedientes?

La respuesta a cada una de las primeras tres preguntas puede darse en forma precisa porque conocemos la relación entre el dato (longitud del lado o del diámetro) y la magnitud de la tarea. En efecto, en el primer caso sabemos que el trabajo que implica pintar una pared (de altura fija) es proporcional a su longitud. En el segundo caso, al tratarse de una pared cuadrada, el trabajo es proporcional a su superficie, que es el cuadrado del lado. En el tercero, el trabajo es proporcional al volúmen del globo, que a su vez es proporcional al cubo del diámetro.

La respuesta a la cuarta pregunta, en cambio, no parece tan fácil de responder. No conocemos cuánto más trabajo es ordenar 2000 expedientes que 1000, Existen numerosos métodos de ordenación que usamos cotidianamente sin preguntarnos realmente cuál es

el mejor. La respuesta a la pregunta **depende** del algoritmo de ordenación que esté utilizando el bibliotecario.

Ordenación por selección. Supongamos que está utilizando el algoritmo de **ordenación por selección** (selection sort). El mismo consiste en seleccionar repetidas veces. La primera vez se selecciona el mínimo del arreglo y se lo coloca en la primera celda.[1] Luego se selecciona el segundo mínimo (para ello alcanza con buscarlo a partir de la segunda celda, ya que en la primera acabamos de ubicar el mínimo del arreglo) y se lo coloca en la segunda celda, etc. Siguiendo de esta manera, obtenemos un ciclo en el que se cumplen las siguientes condiciones como invariante:

- el arreglo es una permutación del original,
- un segmento inicial del arreglo está ordenado, y
- dicho segmento contiene los elementos mínimos del arreglo.

```
{Pre: n ≥ 0 ∧ a = A}
proc selection_sort (in/out a: array[1..n] of T)
    var i, minp: nat
    i:= 1                          {Inv: a es permutación de A ∧ a[1,i) está ordenado ∧}
                                   {∧ los elementos de a[1,i) son menores o iguales a los de a[i,n]}
    do i < n → minp:= min_pos_from(a,i)
               swap(a,i,minp)
               i:= i+1
    od
end proc
{Post: a está ordenado y es permutación de A}
```

En la primera ejecución del ciclo, este algoritmo utiliza la función min_pos_from para encontrar la posición minp donde se encuentra el mínimo de todo el arreglo y luego ubica el mínimo encontrado en la primera posición del arreglo utilizando el procedimiento swap. En la segunda ejecución del ciclo, vuelve a utilizar la función min_pos_from para encontrar la posición del segundo mínimo y luego ubica el segundo mínimo en la segunda posición del arreglo. En general, una vez ubicado el (i-1)-ésimo mínimo del arreglo en la posición i-1, el algoritmo utiliza min_pos_from para encontrar la posición del i-ésimo mínimo del arreglo y luego el procedimiento swap para colocarlo en la posición i. Toda modificación del arreglo se realiza a través del procedimiento swap:

```
{Pre: a = A ∧ 1 ≤ i,j ≤ n }
proc swap (in/out a: array[1..n] of T, in i,j: nat)
    var tmp: T
    tmp:= a[i]
    a[i]:= a[j]
    a[j]:= tmp
end proc
{Post: a[i] = A[j] ∧ a[j] = A[i] ∧ ∀ k. k ∉{i,j} ⇒ a[k]=A[k]}
```

La postcondición del procedimiento swap implica que el mismo produce una permutación del arreglo que recibe como parámetro. Como el algoritmo selection_sort sólo

[1] Intercambiándolo con el elemento que se encuentra en ella para que no se pierda su valor.

modifica el arreglo a invocando al procedimiento swap reiteradamente, la condición que establece que el arreglo ordenado debe ser una permutación del inicial queda garantizada. Ésta es una observación interesante: **si uno se limita a modificar el arreglo sólo invocando al procedimiento swap, necesariamente se tendrá siempre una permutación del arreglo inicial**.

A continuación, detallamos el algoritmo utilizado para **seleccionar** el i-ésimo mínimo. Dado que el algoritmo selection_sort va ubicando los primeros i-1 mínimos en los primeros i-1 lugares del arreglo, para encontrar el i-ésimo mínimo de a basta con buscar el mínimo de a[i,n].

La función min_pos_from realiza esta tarea. Nuevamente es necesario realizar un ciclo para encontrar el mínimo. Como el objetivo es intercambiar el mínimo con el que se encuentra en la posición i del arreglo, es necesario devolver la **posición** donde el mínimo se encuentra. Para ello, se utiliza la variable minp, y el invariante del ciclo es que a[minp] es el mínimo del fragmento explorado de a[i,n].

```
{Pre: 0 < i ≤ n}
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    var j: nat
    minp:= i
    j:= i+1                                   {Inv: a[minp] es el mínimo de a[i,j)}
    do j ≤ n → if a[j] < a[minp] then minp:= j fi
               j:= j+1
    od
end fun
{Post: a[minp] es el mínimo de a[i,n]}
```

¿Por qué la función min_pos_from no devuelve el mínimo sino su posición?

¿Por qué dice i < n en vez de i ≤ n en la condición del **do** del procedimiento selection_sort?

¿Por qué se llama selection_sort?

El comando for. En el algoritmo presentado los dos ciclos **do** se utilizan para recorrer un arreglo desde una posición predeterminada hasta otra también predeterminada. Además, el índice utilizado para recorrer el arreglo (i en un caso y j en el otro) sólo son modificados al final del cuerpo de cada ciclo, cuando se los incrementa. En estas situaciones, utilizaremos una notación más compacta. En primer lugar, omitiremos declarar el índice. Además, en vez de escribir

```
k:= n
do k ≤ m → C
            k:= k+1
od
```

escribiremos

```
for k:= n to m do C od
```

Para que esta notación tenga sentido es fundamental que k no sea modificado en el cuerpo C del ciclo. Observar que esta notación hace más evidente que C se ejecuta una vez para cada valor de k desde n hasta m.

Utilizando ciclos **for** el procedimiento selection_sort puede escribirse

```
{Pre: n ≥ 0 ∧ a = A}
proc selection_sort (in/out a: array[1..n] of T)
    var minp: nat
    for i:= 1 to n-1 do        {Inv: a es permutación de A ∧ a[1,i) está ordenado ∧}
                               {∧ los elementos de a[1,i) son menores o iguales a los de a[i,n]}
        minp:= min_pos_from(a,i)
        swap(a,i,minp)
    od
end proc
{Post: a está ordenado y es permutación de A}
```

Asimismo, la función de **selección** min_pos_from se puede escribir

```
{Pre: 0 < i ≤ n}
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    minp:= i
    for j:= i+1 to n do                    {Inv: a[minp] es el mínimo de a[i,j)}
        if a[j] < a[minp] then minp:= j fi
    od
end fun
{Post: a[minp] es el mínimo de a[i,n]}
```

Ahora que tenemos un algoritmo de ordenación, podemos intentar responder la pregunta motivadora. Para ello, debemos analizar el algoritmo con el propósito de averiguar cuánto más trabajo implica ordenar 2000 expedientes que 1000 expedientes. Debemos hallar algún parámetro respecto del cual el trabajo sea proporcional. Con el propósito de medir el trabajo que realiza el algoritmo, volvemos nuestra atención sobre el mismo y observamos que contiene distintas tareas u operaciones: realiza asignaciones, sumas, comparaciones, llamadas a funciones y procedimientos, ejecuciones de ciclos, intercambios.

Una posibilidad sería contabilizar todas estas operaciones. Sería complicado ya que no todas ellas son equivalentes, no todas ellas requieren el mismo tiempo para realizarse. Deberíamos contabilizarlas por separado. Realizaremos un análisis más sencillo, que consiste en elegir una sola operación y contar cuántas veces se repite. Por supuesto que para que el análisis sea correcto, debemos ser criteriosos al elegir dicha operación: debe ser una que represente bien el trabajo que realiza el algoritmo. Para ello se requiere:

- debe ser constante (el trabajo que implica realizar dicha operación debe ser el mismo, independientemente del número de celdas del arreglo),
- "ninguna otra operación puede repetirse más que la elegida"; más precisamente, toda otra operación puede repetirse a lo sumo en forma proporcional al modo en que se repite la operación elegida.

Por ejemplo, en el caso que nos ocupa, el procedimiento selection_sort contiene un ciclo; debemos buscar dentro de él la tarea a elegir ya que allí se encuentran las operaciones que más se repiten. El **for** contiene implícitamente operaciones que se repiten: se incrementa el índice y se chequea la condición luego de cada ejecución del cuerpo del

mismo. Además, cada vez que se ejecuta el cuerpo se invoca a la función min_pos_from y al procedimiento swap. Cada una de estas operaciones se realiza una vez para cada valor de i, es decir, un total de n-1 veces.

Todas estas operaciones son constantes, con la sola excepción de la ejecución de la función min_pos_from. En efecto, la función min_pos_from contiene a su vez (además de la inicialización de minp) un ciclo **for** que requiere la repetición de ciertas operaciones. Como todo ciclo **for** contiene incrementos y chequeos de condición de salida implícitos luego de cada ejecución. Además, cada vez que se ejecuta el ciclo se realizan accesos al arreglo, una comparación entre celdas del mismo y posiblemente una asignación a minp. Todas estás operaciones (salvo la asignación a minp que requiere que la condición del **if** sea verdadera para ejecutarse) se realizan para cada valor de j, es decir, un total de n-i veces.

Cualquiera de estas operaciones es representativa del comportamiento del algoritmo. Además de ser constantes, son las que más se repiten porque fueron encontradas en el ciclo **for** de la función min_posfrom, que a su vez se invoca desde el ciclo **for** del procedimiento selection_sort. Elegimos, como es habitual al analizar algoritmos de ordenación, la comparación entre elementos del arreglo a[j] <a[minp] que se encuentra en la condición del **if**. Como ya observamos, la misma se realiza n-i veces cuando se invoca la función select con el parámetro i. También observamos que la función min_pos_from se invoca n-1 veces, cada vez con distintos valores de i $\in \{1 \ldots \text{n-1}\}$, el número total de veces que se realiza dicha comparación es n-1 + n-2 + ... + 1. Esto es, un total de $\frac{n(n-1)}{2}$ veces, o distribuyendo, $\frac{n^2}{2} - \frac{n}{2}$ veces.

Resolviendo el problema del bibliotecario. Este análisis nos permite concluir que el trabajo que realiza el procedimiento selection_sort para ordenar un arreglo de longitud n es proporcional a $\frac{n^2}{2} - \frac{n}{2}$. Para el problema del bibliotecario, entonces, tenemos que ordenar 1000 expedientes con este algoritmo involucra 499500 comparaciones mientras que ordenar 2000, involucra 1999000 comparaciones. Es decir, ordenar 2000 expedientes es aproximadamente 4 veces más trabajoso que ordenar 1000. Si tarda un día en ordenar 1000, tardará 4 en ordenar 2000.

Puede observar que la fórmula utilizada, $\frac{n^2}{2} - \frac{n}{2}$, es innecesariamente complicada para resolver el problema. El término que predomina en esta ecuación es el de grado 2. Podríamos decir que el trabajo de ssort es proporcional a $\frac{n^2}{2}$. Por lo tanto, también es proporcional a n^2.

Repetimos el cálculo con esta fórmula más sencilla: ordenar 1000 expedientes involucra 1 millón de comparaciones, y ordenar 2000 involucra 4 millones. Arribamos a la misma conclusión (que ordenar 2000 expedientes es 4 veces más trabajo que ordenar 1000) que con la fórmula innecesariamente complicada. Por esta razón es habitual simplificar la notación lo más posible, ignorando constantes multiplicativas (en nuestro caso, $\frac{1}{2}$) y términos de crecimiento despreciable comparado con otros (en nuestro caso, $\frac{n}{2}$ que crece más lentamente que $\frac{n^2}{2}$.

Número de operaciones de un comando. Frecuentemente vamos a querer contar o estimar el número de operaciones que se realizan al ejecutarse un comando determinado. Como no todas las operaciones son igualmente significativas para la performance de un

programa dado frecuentemente se cuentan sólo operaciones de un cierto tipo. La cuenta que se realice dependerá de las operaciones que se pretenden contar y del comando que se está analizando.

Si bien lo habitual es contabilizar las operaciones intuitivamente como hicimos en el caso de selection_sort, a continuación se da una descripción informal de cómo pueden contarse metódicamente las operaciones que se realizan durante la ejecución de un comando dado.

Si queremos contar el número de operaciones que se realizan al ejecutarse la secuencia de comandos $C_1;C_2;\ldots;C_n$, sumamos las operaciones que se realizan durante la ejecución de cada comando de la secuencia:

$$\text{ops}(C_1;C_2;\ldots;C_n) = \text{ops}(C_1) + \text{ops}(C_2) + \ldots + \text{ops}(C_n)$$

En particular, como **skip** representa una secuencia vacía de comandos, ops(**skip**) = 0.

El ciclo **for** k:= n **to** m **do** C(k) **od** puede verse intuitivamente como una abreviatura de la secuencia de comandos C(n);C(n+1);...;C(m).[2] Por ello, si queremos contar el número de operaciones de un ciclo **for**, sumamos las operaciones que se realizan en cada ejecución del cuerpo del mismo. Podemos utilizar por ejemplo la fórmula:

$$\text{ops}(\textbf{for k:= n to m do C(k) od}) = \sum_{\text{k}=n}^{m} \text{ops}(\text{C(k)})$$

Notar que en el lado derecho de la ecuación hemos utilizado n y m para denotar, respectivamente, los valores de las expresiones n y m.

Esta fórmula no tiene en cuenta las operaciones necesarias para evaluar n y m ni las necesarias para inicializar y modificar k y para compararlo con el valor de m. La fórmula:

$$\text{ops}(\textbf{for k:= n to m do C(k) od}) = \text{ops(n)} + \text{ops(m)} + \sum_{\text{k}=n}^{m} \text{ops}(\text{C(k)})$$

tiene en cuenta las operaciones necesarias para evaluar n y m, y la fórmula

$$\text{ops}(\textbf{for k:= n to m do C(k) od}) = (m - n + 2) * \text{ops}(\text{k}\leq m) + \sum_{\text{k}=n}^{m} \text{ops}(\text{C(k)})$$

no tiene en cuenta las operaciones necesarias para evaluar n y m pero sí las necesarias para comparar k con m. Observar que se contabilizan $m - n + 2$ comparaciones,[3] pero se utiliza m en vez de m dado que no se evalúa la expresión m cada vez que se ejecuta el cuerpo del **for** sino una sola vez para toda la ejecución del comando **for**.

Se deja como ejercicio proponer una fórmula que tenga en cuenta todas las operaciones que se requieren para ejecutar el **for**, incluso la inicialización y las modificaciones de k.

[2] En realidad, como n y m no necesariamente son constantes, el ciclo **for** no puede reemplazarse en el código por C(n);C(n+1);...;C(m). Por ejemplo, en el caso de la ordenación por selección, cada llamada a la función min_pos_from, es con un parámetro i diferente. Por lo tanto, el comando **for** en el cuerpo de dicha función itera, para cada i, un número diferente de veces.

[3] ¿Por qué se suma 2?

Si queremos contar el número de operaciones que se realizan al ejecutarse el comando condicional **if** b **then** c **else** d **fi**, debemos considerar dos casos: b verdadero ó b falso:

$$\text{ops}(\textbf{if b then C else D fi}) = \begin{cases} \text{ops(b)+ops(C)} & b = \text{verdadero} \\ \text{ops(b)+ops(D)} & b = \text{falso} \end{cases}$$

Nuevamente utilizamos b para referirnos al valor de la expresión b.

Si queremos contar el número de operaciones que se realizan al ejecutarse la asignación x:= e, tenemos 2 fórmulas dependiendo de si queremos o no contar la asignación en sí como una operación. En el primer caso tenemos ops(x:=e) = ops(e)+1 mientras que en el segundo tenemos simplemente ops(x:=e) = ops(e).

En los casos del comando condicional y de la asignación, ops(b) y ops(e) representan los números de operaciones necesarios para evaluar las expresiones b y e.

Para contar el número de operaciones que se realizan al ejecutarse un ciclo **do** es necesario establecer el número de veces que se ejecutará el cuerpo del ciclo. No siempre es fácil obtener dicho número. Una vez determinado este número, llamémosle n, se obtiene una fórmula parecida a la del **for**,

$$\text{ops}(\textbf{do b} \rightarrow \text{C } \textbf{od}) = \text{ops(b)} + \sum_{\text{k=1}}^{n} d_k$$

donde d_k es el número de operaciones que realiza la k-ésima ejecución del cuerpo C del ciclo y la subsiguiente evaluación de la condición o guarda b.

Finalmente, se deja como ejercicio definir las ecuaciones correspondientes a operadores lógicos, relacionales y aritméticos, que se parecerán a las de la asignación explicada más arriba.

Número de comparaciones de la ordenación por selección. A modo de ejemplo, contemos el número de **comparaciones entre elementos del arreglo a** en el algoritmo de ordenación por selección:

$$\begin{aligned} \text{ops(selection\_sort(a))} &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \ \text{ops(min\_pos\_from(a,i))} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \ \sum_{\text{j=i+1}}^{\text{n}} \text{ops(a[j] < a[minp])} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \ \sum_{\text{j=i+1}}^{\text{n}} 1 \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{(n-i)} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{i} \\ &= \frac{\text{n*(n-1)}}{2} \\ &= \frac{\text{n}^2}{2} - \frac{\text{n}}{2} \end{aligned}$$

Esta expresión -idéntica a la obtenida apelando exclusivamente a nuestra intuición- indica que el número de comparaciones de la ordenación por selección **es del orden de** n^2, terminología que haremos más precisa pronto. Intuitivamente, el número de comparaciones de la ordenación por selección es proporcional a n^2.

Número de intercambios (swaps) de la ordenación por selección. Observemos que sólo se realizan swaps durante la ejecución del procedimiento selection_sort y no durante la de la función min_pos_from. Por cada valor de i se hace exactamente un intercambio (swap) entre a[i] y a[minp] al final del cuerpo del **for** de selection_sort. Como i toma $n-1$ valores, son $n-1$ intercambios.

Utilizando las fórmulas se obtiene lo mismo

$$\begin{aligned} \text{ops(ssort(a))} &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{ops(swap(a,i,minp))} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} 1 \\ &= \text{n-1} \end{aligned}$$

Es decir que el número de intercambios (swaps) de la ordenación por selección **es del orden de** n, es decir, es proporcional a n.

Contestando la pregunta 4 de la página 1. Asumiendo que el bibliotecario utiliza el método de ordenación por selección, hace del orden de n^2 comparaciones. Le llevó un día hacer 1.000.000 de ellas, por lo que le llevará aproximadamente 4 días hacer 4.000.000 de ellas.

Sin embargo, luego veremos algoritmos de ordenación mejores que le permitirían ordenar 2000 expedientes en poco más del doble del tiempo que le lleva ordenar 1000.

Otro ejemplo: ordenación por inserción. No siempre es posible calcular el número exacto de operaciones, dado que puede depender de cada input. El siguiente algoritmo de ordenación por inserción es un ejemplo de ello:

proc isort (**in**/**out** a: **array**[1..n] **of T**) $\{pre : n \geq 0 \wedge a = A\}$
  for i:= 1 **to** n **do** $\{inv : a[1, i)$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$
    insert(a,i)
  od
end proc $\{pos : a$ *está ordenado y es permutación de* $A\}$

Para escribir el invariante del procedimiento insert denotamos por $a[1\hat{..}i]$ la secuencia de celdas de a desde la posición 1 hasta la i salteando la j-ésima (para $1 \leq j \leq i$).

proc insert (**in**/**out** a: **array**[1..n] **of T**, **in** i: **int**) **ret** $\{pre : 0 < i \leq n \wedge a = A\}$
  j:= i $\{inv : a[1\hat{..}i]$ *y* $a[j, i]$ *están ordenados* $\wedge\ a$ *es permutación de* $A\}$
  do $j > 1 \wedge a[j] < a[j-1] \rightarrow$ swap(a,j-1,j)
    j:= j-1
  od
end proc $\{pos : a[1, i]$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$

Número de comparaciones de la ordenación por inserción. Nuevamente contamos **comparaciones entre elementos del arreglo a**, que en este caso son de la forma $a[j] < a[j-1]$ y sólo se realizan en el procedimiento insert. Intuitivamente, cuando i=1, no se realiza ninguna, cuando i=2 se realiza 1, cuando i=3 se realizan al menos 1 y a lo sumo 2, ..., cuando i=n se realizan al menos 1 y a lo sumo n-1. No podemos obtener el número exacto de comparaciones ya que éste depende del input. Pero podemos afirmar que en el **mejor caso** serán 0 + 1 + 1 + ... + 1 = n-1 comparaciones (es decir, del orden de n comparaciones) y en el **peor caso**, 0 + 1 + 2 + ... + (n-1) = $\frac{n^2}{2} - \frac{n}{2}$ comparaciones (es decir, del orden de n^2 comparaciones).

Obtener el número de comparaciones en el peor caso es importante pues establece una **cota**, una **garantía** sobre el comportamiento del algoritmo.

Obtener el número de comparaciones en el mejor caso no tiene la misma importancia, sólo establece una **posibilidad** sobre el comportamiento del algoritmo.

Sí es importante establecer el número de comparaciones en el **caso promedio** o **caso medio** ya que éste determina el comportamiento del algoritmo en la **práctica**. Para calcularlo se necesitan conocimientos de probabilidades. El número de comparaciones de la ordenación por inserción en el caso promedio es del orden de n^2.

Análisis de Algoritmos

El ejemplo ilustra varios aspectos del análisis del comportamiento de un algoritmo:

casos: no siempre es posible contar exactamente. Se distingue entre mejor caso, peor caso y caso promedio. El estudio del peor caso permite establecer una cota superior, una garantía de comportamiento. El caso medio revela el comportamiento en la práctica, pero es más difícil de establecer.

operación elemental: se cuenta el número de operaciones elementales. Ésta debe:

- ser de duración constante, su duración no debe depender de n,
- ser representativa del comportamiento, toda otra operación debe repetirse a lo sumo en forma proporcional a la operación elemental elegida. En la ordenación por selección, el intercambio (swap) no era una operación representativa.

aproximación: se ignoran constantes multiplicativas y los términos que resultan despreciables cuando n crece. Esto puede justificarse de varias maneras:

- la duración de la operación elemental depende del hardware, mejor hardware modificaría esas constantes.
- uno cuenta operaciones, pero no hace precisa la unidad de tiempo, no se sabe si cuenta segundos, días, microsegundos, años, etc. No estando determinada la unidad, las constantes pierden sentido.
- uno puede querer comparar 2 algoritmos cuyos análisis fueron hechos eligiendo operaciones elementales distintas en uno y en otro, sin saber a priori si dichas operaciones elementales tienen igual o diferente duración.

Pero hay que tener presente que se está haciendo una aproximación. Las constantes y términos que acabamos de llamar "despreciables" pueden ser significativos para valores suficientemente bajos de n, o en casos en que se pretende realizar un análisis más fino, más detallado, más preciso.

La Notación $\mathcal{O}$

Sean $c(n)$ el número de comparaciones de la ordenación por selección para arreglos de tamaño n, $s(n)$ el número de swaps del mismo algoritmo para arreglos del mismo tamaño. Por lo visto anteriormente, $c(n) = \frac{n^2}{2} - \frac{n}{2}$ y $s(n) = n - 1$. Observemos el crecimiento de estas funciones cuando n se duplica o triplica:

n	1.000	2.000	3.000
$c(n) = \frac{n^2}{2} - \frac{n}{2}$	499.500	1.999.000	4.498.500
n^2	1.000.000	4.000.000	9.000.000
$s(n) = n - 1$	999	1.999	2.999

¿Cuánto crece $c(n)$ cuando n se duplica o triplica? Puede observarse que los valores de $c(n)$ se cuadruplican o nonuplican. Lo mismo ocurre con n^2. Podemos decir que "a la larga, $c(n)$ crece al mismo ritmo que n^2" o que "a la larga, $c(n)$ es proporcional a n^2" o, como dijimos antes, que "$c(n)$ es del orden de n^2". De la misma manera, decimos que "a la larga, $s(n)$ crece al mismo ritmo que n" o que "a la larga, $s(n)$ es proporcional a n" o que "$s(n)$ es del orden de n".

Cuando lo que uno quiere expresar es el ritmo de crecimiento de una función $t(n)$ resulta conveniente utilizar funciones que tengan el mismo ritmo de crecimiento que $t(n)$ pero cuya expresión sea más simple. En el ejemplo anterior, si lo que se intenta comunicar es el ritmo de crecimiento, conviene decir que el número de comparaciones es proporcional a n^2 que decir que es exactamente igual a $\frac{n^2}{2} - \frac{n}{2}$. La primera expresión, más simple, me dice con la mayor sencillez posible que cuando n se duplica el número de comparaciones de la ordenación por selección se cuadruplica.

Pronto desarrollaremos una notación para expresar justamente que una cierta función crece a la larga al mismo ritmo que otra función. Antes de eso es conveniente introducir

una notación para un concepto más sencillo de definir: "a la larga, $t(n)$ crece **a lo sumo** al ritmo de $f(n)$", que denotaremos $t(n) \in \mathcal{O}(f(n))$:

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto

$$\mathcal{O}(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . t(n) \leq cf(n)\}$$

de todas las funciones que a la larga crecen a lo sumo al ritmo de $f(n)$.

La notación $\forall^\infty n \in \mathbb{N}.\mathcal{P}(n)$ expresa que $\mathcal{P}$ se cumple para casi todo $n \in \mathbb{N}$, es decir, se cumple salvo a lo sumo en una cantidad finita de números naturales. Otra forma de expresar lo mismo es escribiendo $\exists n_0 \in \mathbb{N} . \forall n \in \mathbb{N} . n \geq n_0 \Rightarrow \mathcal{P}(n)$. Preferimos la notación utilizada en la definición por ser más compacta y evitar mencionar n_0.

Otra observación es que sólo importa el comportamiento de las funciones para n suficientemente grande, por lo que uno podría debilitar la condición $t, f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, requiriendo sólo que $\forall^\infty n \in \mathbb{N}$. $t(n), f(n) \in \mathbb{R}^{\geq 0}$, es decir, permitiendo que $t(n)$ y $f(n)$ devuelvan valores negativos para una cantidad finita de n's.

Si $t(n) \in \mathcal{O}(f(n))$ decimos que $t(n)$ es (a lo sumo) **del orden de** $f(n)$.

En particular, para el caso de la ordenación por selección, el número de comparaciones $t(n) = \frac{n^2}{2} - \frac{n}{2}$ es del orden de n^2, dado que $t(n) \leq n^2$ para todo $n \in \mathbb{N}$.

Otro ejemplo interesante lo proporciona la función $t(n) = 5n^2 + 300n + 15$. A pesar de las constantes 5, 300 y 15, se puede comprobar fácilmente que $t(n)$ es del orden de n^2.

También podemos comprobar que la definición de $\mathcal{O}$ determina que las constantes multiplicativas sean ignoradas. Esta propiedad es esperada ya que si d_1 y d_2 son no nulas, $d_1 t(n)$ y $d_2 f(n)$ son proporcionales a $t(n)$ y $f(n)$ respectivamente:

Proposición: Si $t(n) \in \mathcal{O}(f(n))$, entonces $d_1 t(n) \in \mathcal{O}(d_2 f(n))$ para todo $d_1, d_2 \in \mathbb{R}^+$.

Demostración: Sea $c \in \mathbb{R}^+$ tal que $t(n) \leq cf(n)$ se cumple para casi todo $n \in \mathbb{N}$. Entonces, $d_1 t(n) \leq d_1 cf(n) = (\frac{d_1 c}{d_2}) d_2 f(n)$.

Otra propiedad que expresa la insignificancia de las constantes multiplicativas es que $df(n) \in \mathcal{O}(f(n))$ para todo $d > 0$. Sigue de la proposición anterior y reflexividad.

Proposición: La relación "es a lo sumo del orden de" entre funciones de $\mathbb{N}$ en $\mathbb{R}^{\geq 0}$ es reflexiva y transitiva.

Demostración: Reflexividad: como $\forall n \in \mathbb{N}, f(n) \leq f(n)$ trivialmente, tomando $c = 1$ tenemos $f(n) \in \mathcal{O}(f(n))$.

Transitividad: Sean $c_1, c_2 \in \mathbb{R}^+$ tales que $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n)$ y $\forall^\infty n \in \mathbb{N}, g(n) \leq c_2 h(n)$. Los naturales que satisfacen ambas desigualdades siguen siendo todos salvo a lo sumo un número finito. Por lo tanto, $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n) \leq (c_1 c_2) h(n)$.

Corolario: $g(n) \in \mathcal{O}(h(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$.

Demostración: El "sólo si" se cumple por transitividad, y el "si" por reflexividad.

Corolario: $f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$.

Observar que esto no significa que la relación "es a lo sumo del orden de" sea antisimétrica, de hecho no lo es. Antisimetría debería establecer que $f = g$.

Lo siguiente establece que para las funciones "interesantes"las constantes aditivas no importan.

Proposición: Si $f(n)$ está acotado inferiormente por un número positivo, entonces $g(n) \in \mathcal{O}(f(n))$ implica $g(n) + d \in \mathcal{O}(f(n))$ para todo $d \geq 0$.

Demostración: Sean $k, c > 0$ tales que $\forall^\infty n.f(n) > k$ y $\forall^\infty n \in \mathbb{N}.g(n) \leq cf(n)$. Sea $c' = c + d/k$. Ahora $\forall^\infty n \in \mathbb{N}.g(n) + d \leq cf(n) + \frac{d}{k}k \leq cf(n) + \frac{d}{k}f(n) \leq c'f(n)$.

Proposición: Para todo $a, b > 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.

Demostración: Para todo $n \in \mathbb{N}^+$, $\log_a n = \log_a b \ \log_b n$. Luego, $\log_a b$ es la constante (positiva por $a, b > 1$) que prueba que $\log_a n \in \mathcal{O}(\log_b n)$. Igual para $\log_b n \in \mathcal{O}(\log_a n)$.

Esto demuestra que en este contexto la base del logaritmo es irrelevante y puede ignorarse.

Regla del Límite

La siguiente regla es útil para demostrar cuándo una función es de un cierto orden, y cuándo no lo es.

Proposición (Regla de Límite): $\forall f, g \in \mathbb{N} \to \mathbb{R}^{\geq 0}$, si $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existe los siguientes enunciados se cumplen:

1. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow \mathcal{O}(f(n)) = \mathcal{O}(g(n))$.
2. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.
3. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = +\infty \Rightarrow \mathcal{O}(g(n)) \subset \mathcal{O}(f(n))$.

Observar que en los dos últimos casos, la inclusión es estricta.

Demostración:

1. Alcanza con comprobar que $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$, dado que si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in \mathbb{R}^+$ entonces $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{l} \in \mathbb{R}^+$. Veamos entonces que $f(n) \in \mathcal{O}(g(n))$. Sea $\varepsilon \in \mathcal{R}^+$, se tiene que $\forall^\infty n \in \mathbb{N}.\frac{f(n)}{g(n)} - l < \varepsilon$, luego $f(n) < (\varepsilon + l)g(n)$.
2. El razonamiento anterior funciona incluso si $l = 0$. Tenemos pues $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Veamos que esta inclusión es estricta verificando que $g(n) \notin \mathcal{O}(f(n))$. Si $g(n) \in \mathcal{O}(f(n))$, entonces hay una constante $c \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, g(n) \leq cf(n)$. Entonces $\forall^\infty n \in \mathbb{N}.\frac{f(n)}{g(n)} \geq \frac{1}{c}$, con ello $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no podría ser menor que $\frac{1}{c}$. Luego $\mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.
3. Como $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ implica que $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, vale por (2).

Corolario:

1. $x < y \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(n^y)$,
2. $x \in \mathbb{R}^+ \Rightarrow \mathcal{O}(\log n) \subset \mathcal{O}(n^x)$,
3. $x \in \mathbb{R}^{\leq 0} \Rightarrow \mathcal{O}(n^x) \subset O(\log n)$,
4. $c > 1 \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(c^n)$.
5. $0 \leq c < 1 \Rightarrow \mathcal{O}(c^n) \subset \mathcal{O}(n^x)$.
6. $c > d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(c^n)$.
7. $d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

Los incisos 3 y 5 son de poco interés ya que no es esperable tener programas cuyo número de comparaciones decrece a medida que n crece. Se enuncian para proporcionar una visión un poco más completa de la jerarquía de funciones determinada por la notación $\mathcal{O}$.

Demostración:

1. $\lim_{n\to\infty} \frac{n^x}{n^y} = \lim_{n\to\infty} \frac{1}{n^{y-x}} = 0$.
2. Como $\lim_{n\to\infty} \log n = +\infty = \lim_{n\to\infty} n^x$, por la Regla de L'Hôpital tenemos $\lim_{n\to\infty} \frac{\log n}{n^x} = \lim_{n\to\infty} \frac{\frac{1}{n}}{xn^{x-1}} = \lim_{n\to\infty} \frac{1}{xn^x} = 0$.
3. $\lim_{n\to\infty} \frac{\log n}{n^0} = \lim_{n\to\infty} \frac{\log n}{1} = \lim_{n\to\infty} \log n = +\infty$. Luego, $\mathcal{O}(n^0) \subset O(\log n)$. Para todo $x \in \mathbb{R}^-$, $\mathcal{O}(n^x) \subset \mathcal{O}(n^0) \subset O(\log n)$.
4. Demostraremos por inducción que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Para $k = 0$ la prueba es trivial. Asumimos como hipótesis inductiva que $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por la Regla de L'Hôpital, $\lim_{n\to\infty} \frac{n^{k+1}}{c^n} = \lim_{n\to\infty} \frac{(k+1)n^k}{c^n \log_e c} = \frac{k+1}{\log_e c} \lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por lo tanto, para todo $k \in \mathbb{N}$, $\mathcal{O}(n^k) \subset \mathcal{O}(c^n)$. Sea $x \in \mathbb{R}$. Sea $k \in \mathbb{N}$ tal que $x \leq k$. Se obtiene $\mathcal{O}(n^x) \subseteq \mathcal{O}(n^k) \subset \mathcal{O}(c^n)$.
5. Similar al anterior, demostrando que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = +\infty$.
6. Sigue de $\lim_{n\to\infty} \frac{c^n}{d^n} = \lim_{n\to\infty} (\frac{c}{d})^n = +\infty$, que vale pues $\frac{c}{d} > 1$.
7. Sea $c > d$. Para todo $n \geq 2c^2$ se puede ver que $n! \geq n(n-1)\dots(n - \lfloor \frac{n}{2} \rfloor) \geq \lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil} \geq c^{2\lceil \frac{n}{2} \rceil} \geq c^n$. Por lo tanto, $c^n \in \mathcal{O}(n!)$ que implica $\mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$. Por el inciso anterior, tenemos $\mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

Proposición: $g(n) \in \mathcal{O}(f(n)) \Rightarrow \mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$.
Demostración: Para $c \in \mathbb{R}^+$, $\forall^\infty n \in \mathbb{N}. g(n) \leq cf(n)$. Luego, $f(n)+g(n) \leq (1+c)f(n)$.
Corolario: $\mathcal{O}(a_k n^k + \ldots + a_1 n + a_0) = \mathcal{O}(n^k)$, si $a_k \neq 0$.
Demostración: Usando la proposición anterior reiteradamente.
Proposición: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\text{máx}(f(n), g(n)))$.
Demostración: Fácil pues $f(n)+g(n) \leq 2\,\text{máx}(f(n), g(n))$ y $\text{máx}(f(n), g(n)) \leq f(n)+g(n)$.
Proposición: Si $\forall^\infty n \in \mathbb{N}. f(n) > 0$, entonces $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$ sii $\mathcal{O}(f(n)g(n)) \subseteq \mathcal{O}(f(n)h(n))$.
Demostración: Fácil pues $\forall^\infty n \in \mathbb{N}$, se verifica $g(n) \leq ch(n)$ sii $f(n)g(n) \leq cf(n)h(n)$.
Proposición: Si $f(n) \in \mathcal{O}(g(n))$ y $\lim_{n\to\infty} h(n) = \infty$, entonces $f(h(n)) \in \mathcal{O}(g(h(n)))$.
Demostración: Si $\forall^\infty n \in \mathbb{N}. f(n) \leq cg(n)$, como a partir de cierto n, $h(n)$ es suficientemente grande, $\forall^\infty n \in \mathbb{N}. f(h(n)) \leq cg(h(n))$.
Ejemplo: Como $n^{1/2} \in \mathcal{O}(n)$ y $log(n)$ tiende a infinito, $\sqrt{\log n} = \log^{1/2} n \in \mathcal{O}(\log n)$.
Definición: Si $\mathcal{O}(t(n)) = \mathcal{O}(1)$, t se dice **constante**. Si $\mathcal{O}(t(n)) = \mathcal{O}(\log n)$, t se dice **logarítmico**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n)$, t se dice **lineal**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^2)$, t se dice **cuadrática**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^3)$, t se dice **cúbica**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^k)$ para algún $k \in \mathbb{N}$, t se dice **polinomial**. Si $\mathcal{O}(t(n)) = \mathcal{O}(c^n)$, para algún $c > 1$, t se dice **exponencial**. Si $t(n)$ expresa la eficiencia de un algoritmo, éste se dice, respectivamente, constante, logarítimico, lineal, cuadrático, cúbico, polinomial, exponencial. Si $t(n)$ expresa la eficiencia del algoritmo más eficiente posible para resolver un determinado problema, también se habla de problema constante, logarítmico, lineal, cuadrático, polinomial, exponencial.
Notación Complementaria. La notación $\mathcal{O}$ está destinada especialmente a establecer cotas superiores a la eficiencia de un programa, es decir, para afirmar que una cierta función a la larga crece a lo sumo al ritmo de otra. También suele ser útil establecer cotas inferiores, para expresar que una cierta función a la larga crece **por lo menos** al

ritmo de otra. Y como anunciamos en la página 10, también pretendemos definir una notación que exprese que una cierta función a la larga crece al mismo ritmo que otra, es decir, que una cierta cota es ajustada.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto

$$\Omega(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists d \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . t(n) \geq df(n)\}.$$

Así, $t(n) \in \Omega(f(n))$ dice que a la larga $t(n)$ crece como mínimo al ritmo de $f(n)$. Se puede comprobar inmediatamente que $g(n) \in \Omega(f(n))$ sii $f(n) \in \mathcal{O}(g(n))$.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto, $\Theta(f(n)) = \mathcal{O}(f(n)) \bigcap \Omega(f(n))$.

Así, $t(n) \in \Theta(f(n))$ dice que a la larga $t(n)$ crece al mismo ritmo que $f(n)$.

Proposición:

1. $f(n) \in \Omega(g(n))$ sii $g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$ sii $\Omega(f(n)) \subseteq \Omega(g(n))$
2. $f(n) \in \Theta(g(n))$ sii $g(n) \in \Theta(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ sii $\Omega(f(n)) = \Omega(g(n))$

Ejemplos. A continuación se ven dos ejemplos de algoritmos de búsqueda: búsqueda lineal y búsqueda binaria.

El de búsqueda lineal es el siguiente, que recorre el arreglo de izquierda a derecha buscando la primer ocurrencia de x.

fun lsearch (a: **array**[1..n] **of T**, x:**T**) **ret** i:**int** $\{pre : n \geq 0\}$
  i:= 1 $\{inv : x \text{ no está en } a[1, i)\}$
  do i $\leq$ n $\wedge$ a[i] $\neq$ x $\rightarrow$ i:= i+1 **od**
end fun $\{pos : x \text{ está en } a \text{ sii } i \leq n \wedge x = a[i]\}$

Para este algoritmo es sencillo analizar el mejor caso, el peor caso y el caso medio. Sean $t_1(n)$, $t_2(n)$ y $t_3(n)$ las funciones que cuentan la cantidad de comparaciones con x en el mejor caso, peor caso y caso medio respectivamente.

mejor caso: Ocurre cuando x se encuentra en la primera posición del arreglo. Se realiza una comparación: $t_1(n) = 1 \in \Theta(1)$.

peor caso: Ocurre cuando x no se encuentra en el arreglo. Se realizan n comparaciones: $t_2(n) = n \in \Theta(n)$.

caso medio: A priori, depende de la probabilidad de que x esté en el arreglo y, en caso de que esté, la probabilidad de que esté en cada posición. Si consideramos el caso en que está, y si consideramos equiprobable que esté en una u otra posición, el promedio del número de comparaciones que hará falta en cada caso es $t_3(n) = \frac{1+2+...+(n-1)+n}{n}$. Simplificando: $t_3(n) = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in \Theta(n)$.

Ejercicio: ¿Cómo puede modificarse la búsqueda lineal si se asume que el arreglo está ordenado? ¿De qué orden serían $t_1(n), t_2(n)$ y $t_3(n)$ en ese caso?

El segundo ejemplo es el de búsqueda binaria, que requiere que el arreglo esté ordenado de menor a mayor. Es similar a cuando uno busca una palabra en un diccionario: uno lo abre al medio y (a menos que tenga la suerte de encontrarla justo donde abrió el diccionario, en cuyo caso la búsqueda termina) si la palabra que uno busca es anterior a las que se ven donde se abrió el diccionario uno limita la búsqueda a la parte izquierda (abriendo nuevamente al medio, etc), si en cambio la palabra que uno busca es posterior a las que se ven, uno limita la búsqueda a la parte derecha, etc.

fun bsearch (a: **array**[1..n] **of T**, x:**T**) **ret** i:**int** $\{pre : n > 0\}$
  var izq,med,der: **int**

izq:= 1
der:= n
i:= 0 $\{inv : (x \text{ está en } a \text{ sii } x \text{ está en } a[izq, der]) \wedge (i \neq 0 \Rightarrow x = a[i])\}$
do izq $\leq$ der $\wedge$ i = 0 $\rightarrow$
med:= (izq+der) $\div$ 2
if x < a[med] $\rightarrow$ der:= med-1
x = a[med] $\rightarrow$ i:= med
x > a[med] $\rightarrow$ izq:= med+1
fi
od
end fun $\{pos : x \text{ está en } a \text{ sii } i \neq 0 \ (\ \wedge x = a[i])\}$

La complejidad del algoritmo está dada por la cantidad de veces que se ejecuta el ciclo, dado que cada ejecución del ciclo insume tiempo constante (a lo sumo 2 comparaciones). En cada ejecución del ciclo, o bien se encuentra x, o bien el espacio de búsqueda se reduce a la mitad. En efecto, si izq_i y der_i denotan los valores de izq y der después de la i-ésima ejecución del ciclo, sabemos que $d_i = \text{der}_i - \text{izq}_i + 1$ son la cantidad de celdas que nos quedan por explorar para encontrar x. Se puede ver que si $d_i > 1$, entonces $d_{i+1} \leq d_i/2$. Como $d_0 =$ n y el ciclo termina cuando $d_i = 1$, esto pasará -en el peor caso- cuando $i = \lceil \log_2 n \rceil$. Por lo tanto, $t(n) \in \mathcal{O}(\log n)$, donde $t(n)$ es el número de comparaciones que se realizan en el peor caso.

De la misma forma puede obtenerse una cota inferior de $t(n)$ (es decir, del peor caso), por ejemplo, demostrando que $t(n) \geq \lfloor \log_4 n \rfloor$ partiendo de que $d_{i+1} \geq d_i/4$. Por ello, se tiene que $t(n) \in \Omega(\log n)$ y por lo tanto $t(n) \in \Theta(\log n)$.

Puede notarse que este algoritmo se comporta mucho mejor que el anterior para grandes valores de n. Supongamos una búsqueda lineal sobre un arreglo de tamaño 1.000.000. Si ahora repitiéramos la búsqueda lineal sobre uno de tamaño 2.000.000, dado que $t_2(n) \in \mathcal{O}(n)$, nos llevaría el doble de tiempo. En cambio, si estuviéramos utilizando búsqueda binaria, como $t(n) \in \mathcal{O}(\log n)$, la diferencia de tardanza sería casi imperceptible, dado que $\log(2n) = \log 2 + \log n = 1 + \log n$. En efecto, se puede observar que el algoritmo hace sólo una comparación más.

Versión recursiva de la búsqueda binaria: el algoritmo que acabamos de ver se puede escribir recursivamente como sigue:

fun bsearch (a: **array**[1..n] **of T**, x:**T**, izq, der : **int**) **ret** i:**int** $\{pre : 0 \leq izq \leq der \leq n\}$
var med: **int**
if izq > der $\rightarrow$ i = 0
izq $\leq$ der $\rightarrow$ med:= (izq+der) $\div$ 2
if x < a[med] $\rightarrow$ i:= bsearch(a, x, izq, med-1)
x = a[med] $\rightarrow$ i:= med
x > a[med] $\rightarrow$ i:= bsearch(a, x, med+1,der)
fi
fi
end fun $\{pos : x \text{ está en } a[izq, der] \text{ sii } i \neq 0 \ (\ \wedge x = a[i])\}$

Hace falta una función principal:

fun search (a: **array**[1..n] **of T**, x:T) **ret** i:**int** $\{pre : n > 0\}$

```
    i:= bsearch(a, x, 1, n)
end fun                                   {pos : x está en a sii i ≠ 0 ( ∧ x = a[i])}
```

1. Recurrencias

Recordemos el problema formulado en una clase anterior:

> **Pregunta 2:** Un bibliotecario demora 1 día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto demorará en ordenar una con 2000 expedientes?

Le proponemos al bibliotecario que ordene la primera mitad de la biblioteca (tarea A $\approx$ 1.000.000 de comparaciones = 1 día de trabajo), luego la segunda mitad (tarea B $\approx$ 1.000.000 de comparaciones = 1 día de trabajo), y luego efectúe la intercalación obvia entre las 2 bibliotecas ordenadas (tarea C $\approx$ 2.000 comparaciones = unos minutos de trabajo). Esto le llevará mucho menos de los 4 días que le llevaría hacer todo con el algoritmo de ordenación por selección.

Esto a su vez puede mejorarse: la tarea A puede desdoblarse en ordenar 500 expedientes (tarea AA $\approx$ 250.000 comparaciones = 1/4 día de trabajo), ordenar los otros 500 expedientes (tarea AB $\approx$ 250.000 comparaciones = 1/4 día de trabajo), e intercalar (tarea AC $\approx$ 1.000 comparaciones). Similarmente para la tarea B. En total, tendríamos ahora $\approx$ 1.004.000 comparaciones, poco más de 1 día. Esta idea puede iterarse: en vez de ordenar grupos de 500, serán grupos de 250 ó 125 ó 68 ó 34 ó . . . ¿cuán chicos pueden ser estos grupos? Una biblioteca de 1 solo expediente es trivial, no requiere ordenación. Una de 2 expedientes ya puede subdivirse en 2 partes de un expediente cada una, ordenar cada parte (trivial) e intercalar.

De esta manera, la utilización de la ordenación por selección finalmente queda eliminada: toda la ordenación se realiza dividiendo la biblioteca en 2 ordenando (utilizando recursivamente esta idea) e intercalando.

El algoritmo de ordenación que acabamos de explicar se llama mergeSort (ordenación por intercalación). En un estilo funcional se puede escribir así:

```
msort :: [T] → [T]
msort [] = []
msort [t] = [t]
msort ts = merge sts1 sts2
           where
               sts1 = msort ts1
               sts2 = msort ts2
               (ts1,ts2) = split ts

split :: [T] → ([T],[T])
split ts = (take n ts, drop n ts)
           where n = length ts ÷ 2

merge :: [T] → [T] → [T]
merge [] sts2 = sts2
merge sts1 [] = sts1
```

merge (t1:sts1) (t2:sts2) = **if** t1 $\leq$ t2
then t1:merge sts1 (t2:sts2)
else t2:merge (t1:sts1) sts2

donde split divide ts en 2 partes de igual longitud ($\pm$ 1) y merge realiza la intercalación. La administración de la memoria en estilo funcional se deja en manos del compilador. Eso facilita su definición. En estilo imperativo la principal dificultad es justamente la de realizar la intercalación: o bien es necesario complicar significativamente el algoritmo, o bien utilizar arreglos auxiliares para realizarla. Volviendo a nuestro ejemplo motivador, se puede ver que no es fácil intercalar 2 bibliotecas (cada una de ellas ordenada) llenas de expedientes si el resultado de la intercalación debe quedar en las mismas bibliotecas, a menos que se cuente con una biblioteca vacía donde poner los expediente temporariamente. Tomando b y c como arreglos temporarios para realizar la intercalación (en realidad sólo uno es necesario, incluimos el otro para simplificar el algoritmo de intercalación) se lo puede escribir de la siguiente manera:

var a: **array**[1..n] **of T**
var b,c: **array**[1..n÷2+1] **of T**
proc msort (**in** izq,der: **int**) $\{pre : n \geq der \geq izq > 0 \wedge a = A\}$
var med: **int**
if der > izq $\rightarrow$ med:= (der+izq) ÷ 2
msort(izq,med)
$\{a[izq, med]$ *permutación ordenada de* $A[izq, med]\}$
msort(med+1,der)
$\{a[med + 1, der]$ *permutación ordenada de* $A[med + 1, der]\}$
for i:= izq **to** med **do** b[i-izq+1]:=a[izq] **od**
$\{b[1, med - izq + 1] = a[izq, med]\}$
for j:= med+1 **to** der **do** c[j-med]:=a[j] **od**
$\{c[1, der - med] = a[med + 1, der]\}$
intercalar(izq,med,der)
$\{a[izq, der]$ *permutación ordenada de* $A[izq, der]\}$
fi
end proc $\{a$ *permutación de* $A \wedge a[izq, der]$ *permutación ordenada de* $A[izq, der]\}$

Asimismo, el procedimiento intercalar puede escribirse de la siguiente manera.

proc intercalar (**in** izq,med,der: **int**)
$\{pre : n \geq der > med \geq izq > 0 \wedge a = A \wedge b = B \wedge c = C\}$
$\{pre : B[1, med - izq + 1] = A[izq, med] \wedge C[1, der - med] = A[med + 1, der]\}$
$\{pre : B[1, med - izq + 1]$ *y* $C[1, der - med]$ *ordenados*$\}$
var i,j,maxi,maxj: **int**
i:= 1
j:= 1
maxi:= med-izq+1
maxj:= der-med
$\{inv : b = B \wedge c = C \wedge a[1, izq) = A[1, izq) \wedge a[k, n] = A[k, n]\}$

$\{inv : a[izq, k) = \ intercalación\ de\ b[1, i)\ con\ c[1, j)\}$

```
    for k:= izq to der do
        if i ≤ maxi ∧ (j > maxj ∨ b[i] ≤ c[j]) then a[k]:= b[i]
                                                    i:=i+1
        else a[k]:= c[j]
             j:=j+1
        fi
    od
end proc
```

$\{a\ permutación\ de\ A \wedge a[izq, der]\ permutación\ ordenada\ de\ A[izq, der]\}$

Finalmente, la ejecución de la ordenación por intercalación comienza llamando al procedimiento con izq igual a 1 y der igual a n:

msort(1,n)

Esta técnica para resolver un problema, consistente en dividir el problema en problemas menores (de idéntica naturaleza pero de menor tamaño), asumir los problemas menores resueltos y utilizar dichos resultados para resolver el problema original, se conoce por "divide and conquer", es decir, "divide y vencerás". Justamente el hecho de que los problemas menores sean de igual naturaleza que el original, es lo que permite que el mismo algoritmo puede aplicarse recursivamente para resolver los problemas menores. Los casos más sencillos (como el del fragmento de arreglo de longitud menor o igual que 1, para el problema de ordenación) deben resolverse aparte. Estos casos frecuentemente son triviales. Usualmente el término "divide y vencerás" se aplica a aquellos casos en que el problema se subdivide en problemas menores "fraccionando" el tamaño de la entrada.

Número de Comparaciones. Sea $t(n)$ el número de comparaciones entre elementos de **T** que realiza el procedimiento msort con una entrada de tamaño n en el peor caso. Cuando la entrada es de tamaño 1, no hace ninguna comparación, $t(1) = 0$. Cuando la entrada es de tamaño mayor a 1, hace todas las comparaciones que hace la primera llamada recursiva al procedimiento msort, esto es, $t(\lceil n/2 \rceil)$, más todas las comparaciones que hace la segunda llamada recursiva al procedimiento msort, esto es, $t(\lfloor n/2 \rfloor)$, más todas las comparaciones que hace el proceso de intercalación en el peor caso, esto es, $n - 1$. Tenemos entonces:

$$t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1.$$

Esto es una recurrencia, es decir, se define la función t en términos de sí misma. ¿Cómo obtener explícitamente el orden de $t(n)$? Usaremos que $t(n)$ está acotada:

$$\begin{array}{rcl} t(n) & \leq & t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n \\ t(n) & \geq & t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \frac{n}{2} \end{array}$$

Para simplificar el cálculo, consideramos primero aquellos valores de n para los que las operaciones $\lceil\ \rceil$ y $\lfloor\ \rfloor$ pueden ser ignoradas: potencias de 2.

Sea entonces $n = 2^m$. A partir de la primera desigualdad, para $m > 0$ tenemos $t(2^m) \leq t(2^{m-1}) + t(2^{m-1}) + 2^m = 2t(2^{m-1}) + 2^m$ de donde se obtiene dividiendo por

2^m,

$$\frac{t(2^m)}{2^m} \leq \frac{t(2^{m-1})}{2^{m-1}} + 1.$$

Iterando este proceso, se llega a

$$\frac{t(2^m)}{2^m} \leq \frac{t(2^{m-1})}{2^{m-1}}+1 \leq \frac{t(2^{m-2})}{2^{m-2}}+2 \leq \frac{t(2^{m-k})}{2^{m-k}}+k \leq \frac{t(2^0)}{2^0}+m = \frac{t(1)}{1}+m = 0+m = m.$$

Despejando, $t(2^m) \leq 2^m m$. Como $n = 2^m$, sabemos que $m = \lg n$. Reemplazando queda $t(n) \leq n \lg n$, para todo n que sea potencia de 2. Podemos concluir entonces que $t(n) \in \mathcal{O}(n \lg n | n$ potencia de 2).

Análogamente, usando la segunda desigualdad obtenemos $t(n) \geq \frac{1}{2} n \lg n$ y concluimos que $t(n) \in \Omega(n \lg n | n$ potencia de 2) y finalmente de ambas conclusiones, obtenemos que $t(n) \in \Theta(n \lg n | n$ potencia de 2).

Resta ver cuál es la cantidad de comparaciones para el resto de los valores de n.

Primero se puede ver que $t(n)$ es creciente. Por inducción en n, $t(n+1) > t(n)$. El caso base es sencillo, $t(2) = t(1) + t(1) + 2 - 1 = 1 > 0 = t(1)$. Supongamos que para todo $1 \leq k < n, t(k+1) > t(k)$. Entonces, $t(n+1) = t(\lceil (n+1)/2 \rceil) + t(\lfloor (n+1)/2 \rfloor) + n + 1 - 1 > t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1 = t(n)$.

Para n suficientemente grande, sea $k \geq 1$ tal que $2^k \leq n < 2^{k+1}$. Esto implica que $k \leq \lg n < k + 1$. Como t es creciente, tenemos $t(n) < t(2^{k+1}) \leq 2^{k+1}(k + 1) = 2(2^k k + 2^k) \leq 2(2^k k + 2^k k) = 4\, 2^k k \leq 4\, 2^{\lg n} \lg n = 4n \lg n$. Por lo tanto $t(n) \in \mathcal{O}(n \lg n)$. También por t creciente, se tiene que $t(n) \geq t(2^k) \geq \frac{1}{2} 2^k k \geq \frac{1}{8} 2^{k+1}(k+1) > \frac{1}{8} 2^{\lg n} \lg n = \frac{1}{8} n \lg n$. Entonces $t(n) \in \Omega(n \lg n)$, o sea, $t(n) \in \Theta(n \lg n)$.

Al analizar el procedimiento msort utilizamos intuitivamente la siguiente notación.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se definen los conjuntos

$$\begin{aligned} \mathcal{O}(f(n)|P(n)) &= \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . P(n) \Rightarrow t(n) \leq c f(n)\} \\ \Omega(f(n)|P(n)) &= \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . P(n) \Rightarrow t(n) \geq c f(n)\} \\ \mathcal{O}(f(n)|P(n)) &= \mathcal{O}(f(n)|P(n)) \bigcap \Omega(f(n)|P(n)) \end{aligned}$$

El razonamiento hecho para el análisis del procedimiento msort puede generalizarse:

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se dice que f es **asintóticamente** o **eventualmente no decreciente** si $\forall^\infty n \in \mathbb{N}, f(n) \leq f(n+1)$. Sea $i \in \mathbb{N}^{\geq 2}$, f es **i-suave** o **i-uniforme** si es eventualmente no decreciente y $\exists d \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, f(in) \leq d f(n)$. Finalmente, f es **suave** o **uniforme** si es i-suave para todo $i \in \mathbb{N}^{\geq 2}$.

Ejemplos: En el análisis del procedimiento msort, $t(n)$ es eventualmente no decreciente. La función $n \lg n$ es eventualmente no decreciente (ya que es producto de dos funciones que lo son). También es 2-suave ya que para todo $n \geq 2$ se cumple $2n \lg(2n) = 2n(\lg 2 + \lg n) = 2n + 2n \lg n \leq 2n \lg n + 2n \lg n = 4n \lg n$ (en realidad, también podemos concluir que es 2-suave por ser producto de dos funciones que son 2-suave). Del resultado que sigue se desprende que $n \lg n$ es también suave.

Lema: f es i-suave si y sólo si f es suave.

Demostración: El "si" es trivial, demostramos el "sólo si" suponiendo que f es i-suave y demostrando que entonces también es j-suave. Sea n suficientemente grande,

$$\begin{aligned} f(jn) &\leq f(i^{\lceil \log_i j \rceil} n) && f \text{ es eventualmente no decreciente y } jn \leq i^{\lceil \log_i j \rceil} n \\ &\leq d^{\lceil \log_i j \rceil} f(n) && f \text{ es } i\text{-suave} \end{aligned}$$

por lo tanto, f es j-suave (con constante $d^{\lceil \log_i j \rceil}$).

Los siguientes son ejemplos de funciones suaves: n^k, $\log n$, $n^k \log n$, mientras que $n^{\log n}$, 2^n o $n!$ son ejemplos de funciones no suaves.

Regla de la suavidad o uniformidad: Sea $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ suave y $t : \mathbb{N} \to R^{\geq 0}$ eventualmente no decreciente, si $t(n) \in \mathcal{O}(f(n)|n$ potencia de $b)$, entonces $t(n) \in \mathcal{O}(f(n))$. Análogamente para Θ y Ω.

Demostración: Sea n suficientemente grande, y sea m tal que $b^m \leq n < b^{m+1}$. Entonces

$$\begin{array}{rcll} t(n) & \leq & t(b^{m+1}) & t \text{ es eventualmente no decreciente} \\ & \leq & cf(b^{m+1}) & t(n) \in \mathcal{O}(f(n)|n \text{ potencia de } b) \\ & \leq & cdf(b^m) & f \text{ es } b\text{-suave} \\ & \leq & cdf(n) & f \text{ es eventualmente no decreciente} \end{array}$$

por lo tanto $t(n) \in \mathcal{O}(f(n))$.

Ejemplo: sea $t(n)$ el número de comparaciones que realiza el procedimiento msort: $t(n)$ es eventualmente no decreciente, $t(n) \in \Theta(n \log n|n$ potencia de $2)$ y $n \log n$ es suave. Luego $t(n) \in \Theta(n \log n)$.

Al estudiar las recurrencias en forma general, se desarrollarán técnicas generales para la resolución de recurrencias, de manera de no verse uno siempre obligado a realizar una prueba detallada como la que hicimos en el caso del mergeSort. Este tema se dicta siguiendo la presentación de los libros "Fundamentos de Algoritmia" de Brassard y Bratley, páginas 135 a 148 e "Introduction to Algorithms: A Creative Approach" de Manber, páginas 50 a 55. Sólo se reproduce acá una breve descripción del método.

Recurrencias/relaciones "divide y vencerás" (divide and conquer). (Introduction to Algorithms: A Creative Approach, páginas 50 y 51)

1. comprobar que $t(n)$ es eventualmente no decreciente.
2. llevar la recurrencia a una ecuación de la forma $t(n) = at(n/b) + g(n)$ para $b \in \mathbb{N}^{\geq 2}$, $g(n) \in \Theta(n^k)$, $k \in \mathbb{N}$ y n **potencia de** b. Observar que $t(n)$ puede tener una expresión general más compleja, pero para el caso de n potencia de b, puede reducirse como la ecuación mencionada.
3. según sea $a > b^k$, o $a = b^k$ o $a < b^k$, se obtienen los siguientes resultados para n potencia de b:

$$t(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{si} \quad a > b^k \\ \Theta(n^k \log n) & \text{si} \quad a = b^k \\ \Theta(n^k) & \text{si} \quad a < b^k \end{cases}$$

4. Si la ecuación inicial no contiene una igualdad sino sólo una cota: $t(n) \leq at(n/b)+g(n)$ (resp. $t(n) \geq at(n/b)+g(n)$) para n **potencia de** b, $g(n) \in \mathcal{O}(n^k)$ (resp. $g(n) \in \Omega(n^k)$) se obtiene el mismo resultado en los 3 casos, sólo que escribiendo $\mathcal{O}$ (resp. Ω) en vez de Θ.

Ejemplos de recurrencias "divide y vencerás" son el número de comparaciones que realiza el procedimiento msort en el peor caso, o el número de comparaciones que realiza la búsqueda binaria en el peor caso. El primer ejemplo fue desarrollado en detalle recientemente.

Para la búsqueda binaria hicimos las cuentas detalladamente, pero ahora podemos volver a hacerla utilizando recurrencias. Si pensamos en el número de comparaciones que hacen falta para buscar en un arreglo de longitud n, observamos que es 1 más el número de comparaciones que hacen falta para buscar en un arreglo de longitud $\lfloor n/2 \rfloor$. Eso nos dá $t(n) = t(\lfloor n/2 \rfloor) + 1$. Aplicando el método presentado más arriba, como $t(n)$ es eventualmente no decreciente y $a = 1$, $b = 2$ y $k = 0$, tenemos $a = b^k$ y por ello $t(n) \in \Theta(\log n)$.

Recurrencias lineales homogéneas. (Fundamentos de Algoritmia, páginas 135 a 140)

1. llevar la recurrencia a una **ecuación característica** de la forma $a_k t_n + \ldots + a_0 t_{n-k} = 0$,
2. considerar el **polinomio característico asociado** $a_k x^k + \ldots + a_0$,
3. determinar las raíces $r_1, \ldots, r_j$ del polinomio característico, de multiplicidad $m_1, \ldots, m_j$ respectivamente (se tiene $m_i \geq 1$ y $m_1 + \ldots + m_j = k$),
4. considerar la forma general de las soluciones de la ecuación característica:

$$\begin{array}{rcl} t(n) & = & c_1 r_1^n + c_2 n r_1^n + \ldots + c_{m_1} n^{m_1-1} r_1^n + \\ & + & c_{m_1+1} r_2^n + c_{m_1+2} n r_2^n + \ldots + c_{m_1+m_2} n^{m_2-1} r_2^n + \\ \vdots & \vdots & \vdots \\ & + & c_{m_1+\ldots+m_{j-1}+1} r_j^n + c_{m_1+\ldots+m_{j-1}+2} n r_j^n + \ldots + c_{m_1+\ldots+m_j} n^{m_j-1} r_j^n \end{array}$$

   como $m_1 + \ldots + m_j = k$, tenemos k incógnitas: $c_1, \ldots, c_k$,
5. con las k **condiciones iniciales** $t_{n_0}, \ldots, t_{n_0+k-1}$ (n_0 es usualmente 0 ó 1) plantear un sistema de k ecuaciones con k incógnitas:

$$\begin{array}{rcl} t(n_0) & = & t_{n_0} \\ t(n_0+1) & = & t_{n_0+1} \\ \vdots & \vdots & \vdots \\ t(n_0+k-1) & = & t_{n_0+k-1} \end{array}$$

6. obtener de este sistema los valores de $c_1, \ldots, c_k$,
7. escribir la **solución final** de la forma $t_n = t'(n)$, donde $t'(n)$ se obtiene a partir de $t(n)$ reemplazando c_i y r_i por sus valores y simplificando la expresión final.
8. **corroborar** que efectivamente $t(n_0 + k) = t_{n_0+k}$, donde t_{n_0+k} puede obtenerse utilizando la ecuación característica.

Un ejemplo puede darse contando t_n, el número de veces que se ejecuta la acción A en el siguiente procedimiento cuando se llama con parámetro n:

```
proc p (in n: int)                                        {pre : n ≥ 0}
    if n = 0 → skip
       n = 1 → A
       n > 1 → p(n-1)
               p(n-2)
    fi
end proc
```

Al calcular t_n, obtenemos

$$t_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ t_{n-1} + t_{n-2} & \end{cases}$$

que es la secuencia de fibonacci.

Apliquemos el método presentado para las recurrencias homogéneas:

1. ecuación característica $t_n - t_{n-1} - t_{n-2} = 0$, $k = 2$.
2. polinomio característico asociado $x^2 - x - 1$.

3. raíces $r_1 = \frac{1+\sqrt{5}}{2}$ de multiplicidad $m_1 = 1$ y $r_2 = \frac{1-\sqrt{5}}{2}$ de multiplicidad $m_2 = 1$.
4. forma general $t(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n$.
5. condiciones iniciales $t_0 = 0$ y $t_1 = 1$. Sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ c_1(\frac{1+\sqrt{5}}{2}) + c_2(\frac{1-\sqrt{5}}{2}) & = & 1 & (t(1) = t_1) \end{array}$$

6. despejando, $c_1 = \frac{1}{\sqrt{5}}$ y $c_2 = -\frac{1}{\sqrt{5}}$.
7. solución final $t_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$.
8. efectivamente, con la solución final $t_2 = 1$ coincidiendo con el resultado obtenido para calcular t_2 usando la recurrencia original.

El siguiente ejemplo no proviene de un algoritmo. Calcular explícitamente t_n donde

$$t_n = \begin{cases} n & n = 0, 1, 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \end{cases}$$

usando la técnica presentada.

1. ecuación característica $t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$, $k = 3$.
2. polinomio característico asociado $x^3 - 5x^2 + 8x - 4$.
3. raíces $r_1 = 1$ de multiplicidad $m_1 = 1$ y $r_2 = 2$ de multiplicidad $m_2 = 2$.
4. forma general $t(n) = c_1 1^n + c_2 2^n + c_3 n 2^n$, simplificando $t(n) = c_1 + c_2 2^n + c_3 n 2^n$.
5. condiciones iniciales $t_0 = 0$, $t_1 = 1$ y $t_2 = 2$. Sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ c_1 + 2c_2 + 2c_3 & = & 1 & (t(1) = t_1) \\ c_1 + 4c_2 + 8c_3 & = & 2 & (t(2) = t_2) \end{array}$$

6. despejando, $c_1 = -2$, $c_2 = 2$ y $c_3 = -1/2$.
7. solución final $t_n = -2 + 2 * 2^n - \frac{1}{2}n2^n$, simplificando $t_n = 2^{n+1} - n2^{n-1} - 2$.
8. efectivamente, con la solución final $t_3 = 2$ coincidiendo con el resultado obtenido para calcular t_3 usando la recurrencia original.

En clase resolvimos también el siguiente ejemplo:

$$t_n = \begin{cases} 0 & n = 0, 1, 2 \\ 8 & n = 3 \\ 7t_{n-1} - 18t_{n-2} + 20t_{n-3} - 8t_{n-4} & \end{cases}$$

Recurrencias no homogéneas. (Fundamentos de Algoritmia, páginas 140 a 148)

1. llevar la recurrencia a una ecuación característica de la forma $a_k t_n + \ldots + a_0 t_{n-k} = b^n p(n)$, donde $p(n)$ es un polinomio no nulo de grado d.
2. considerar el polinomio característico asociado $(a_k x^k + \ldots + a_0)(x-b)^{d+1}$,
3. determinar las raíces $r_1, \ldots, r_j$ del polinomio característico, de multiplicidad $m_1, \ldots, m_j$ respectivamente (se tiene $m_i \geq 1$ y $m_1 + \ldots + m_j = k + d + 1$),
4. considerar la forma general de las soluciones de la ecuación característica:

$$\begin{array}{rcl} t(n) & = & c_1 r_1^n + c_2 n r_1^n + \ldots + c_{m_1} n^{m_1-1} r_1^n + \\ & + & c_{m_1+1} r_2^n + c_{m_1+2} n r_2^n + \ldots + c_{m_1+m_2} n^{m_2-1} r_2^n + \\ \vdots & \vdots & \vdots \\ & + & c_{m_1+\ldots+m_{j-1}+1} r_j^n + c_{m_1+\ldots+m_{j-1}+2} n r_j^n + \ldots + c_{m_1+\ldots+m_j} n^{m_j-1} r_j^n \end{array}$$

   como $m_1 + \ldots + m_j = k + d + 1$, tenemos $k + d + 1$ incógnitas: $c_1, \ldots, c_{k+d+1}$,
5. a partir de las k condiciones iniciales $t_{n_0}, \ldots, t_{n_0+k-1}$ (n_0 es usualmente 0 ó 1), obtener usando la ecuación característica, los valores de $t_{n_0+k}, \ldots, t_{n_0+k+d}$,
6. con los $k+d+1$ valores $t_{n_0}, \ldots, t_{n_0+k+d}$ plantear un sistema de $k+d+1$ ecuaciones con $k + d + 1$ incógnitas:

$$\begin{array}{rcl} t(n_0) & = & t_{n_0} \\ t(n_0+1) & = & t_{n_0+1} \\ \vdots & \vdots & \vdots \\ t(n_0+k+d) & = & t_{n_0+k+d} \end{array}$$

7. obtener de este sistema los valores de $c_1, \ldots, c_{k+d+1}$,
8. escribir la **solución final** de la forma $t_n = t'(n)$, donde $t'(n)$ se obtiene a partir de $t(n)$ reemplazando c_i y r_i por sus valores y simplificando la expresión final.
9. **corroborar** que efectivamente $t(n_0 + k + d + 1) = t_{n_0+k+d+1}$, donde $t_{n_0+k+d+1}$ puede obtenerse utilizando la ecuación característica.

Ejemplo: calcular explícitamente t_n donde $t_n = \begin{cases} 0 & n = 0 \\ 2t_{n-1} + n & \end{cases}$ usando la técnica presentada para las recurrencias no homogéneas.

1. ecuación característica $t_n - 2t_{n-1} = n$, $k = 1$, $b = 1$, $p(n) = n$, $d = 1$.
2. polinomio característico asociado $(x-2)(x-1)^2$.
3. raíces $r_1 = 2$ de multiplicidad $m_1 = 1$ y $r_2 = 1$ de multiplicidad $m_2 = 2$.
4. forma general $t(n) = c_1 2^n + c_2 1^n + c_3 n 1^n$, simplificando $t(n) = c_1 2^n + c_2 + c_3 n$.
5. condiciones iniciales $t_0 = 0$. También podemos obtener usando la recurrencia $t_1 = 2t_0 + 1 = 1$ y $t_2 = 2t_1 + 2 = 4$.
6. sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ 2c_1 + c_2 + c_3 & = & 1 & (t(1) = t_1) \\ 4c_1 + c_2 + 2c_3 & = & 4 & (t(2) = t_2) \end{array}$$

7. despejando, $c_1 = 2$, $c_2 = -2$ y $c_3 = -1$.
8. solución final $t_n = 2 * 2^n - 2 - n$, simplificando $t_n = 2^{n+1} - n - 2$.
9. efectivamente, con la solución final $t_3 = 11$ coincidiendo con el resultado obtenido para calcular t_3 usando la recurrencia original.