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

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

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.