Desarrollo detallado de una solución (unidad 3)

Aclaración: Este texto se comparte en calidad de borrador, todavía puede estar incomplto o contener algunos errroes.

Es importante destacar que en programación (y muchísimos otros casos en materias de ingeniería) **es mucho más fácil entender cómo funciona una solución que se nos presenta, que poder llegar a ella nosotros mismos desde cero**.

En este documento se pretende hacer hincapié en el *razonamiento* o el *proceso mental* que nos permite, partiendo de la hoja en blanco, arribar a una solución correcta. Es decir, explicar en detalle *la construcción paso a paso del algoritmo*, y no centrarse solo en su *funcionamiento final* una vez terminado.

Para ello, vamos a tomar como referencia un ejercicio de complejidad alta de la unidad 3:

> Se quiere entregar una beca a los dos mejores egresados de una carrera. Desarrolle un algoritmo que permita ingresar el nombre completo, el promedio logrado en su carrera, y la cantidad de aplazos en su historia académica de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. El programa debe finalmente informar los nombres de los dos postulantes con mejor promedio. En caso de empate entre dos postulantes con igual promedio, se debe seleccionar aquel con menor cantidad de aplazos.

La estrategia para presentar la solución será comenzar resolviendo versiones muy simplificadas del problema (ignorando o modificando partes del enunciado), y sobre esas soluciones ir reinsertando de a una las complicaciones u omisiones hasta llegar nuevamente al problema original.

Sin embargo, **la mejor forma de aprovechar este material, es intentar primero resolver el problema uno mismo, antes de pasar a leer las explicaciones. El alumno puede intentar la versión completa. Si no logra resolverlo, introducir una simplificación. Si sigue sin poder resolverlo completamente, introducir otra. Y así sucesivamente**. En algunas versiones hay ayudas. Intente primero sin leer las ayudas, y si no sale reintente considerando las ayudas antes de ”bajar” a la siguiente versión. Se parte de un problema bastante difícil, pero se presentan suficientes simplificaciones como para que cualquier estudiante que haya asistido o leído la teoría de esta unidad llegue a un punto en que pueda resolverlo.

Cuando el alumno logre resolver una versión del problema (no antes!) se recomienda que lea la explicación que se ofrece en la segunda parte de este documento, para comparar con su propia solución y asegurarse de haber observado y entendido todo lo necesario antes de volver a la versión anterior (más compleja).

A continuación se listan las *versiones* del problema en orden de complejidad decreciente (desde el original, hacia el más simple, el número indica la complejidad). El alumno debe intentar esta lista en ese orden (si no puede resolver el 6, intentar antes hacer el 5; si no puede con ese, intentar antes el 4, etc).

Versiones del problema

6. Problema completo

> Se quiere entregar una beca a los dos mejores egresados de una carrera. Desarrolle un algoritmo que permita ingresar el nombre completo, el promedio logrado en su carrera, y la cantidad de aplazos en su historia académica de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. El programa debe finalmente informar los nombres de los

> dos postulantes con mejor promedio. En caso de empate entre dos postulantes con igual promedio, se debe seleccionar aquel con menor cantidad de aplazos.

Ayudas:

- **Si resolvió antes la versión 5**: analice la forma del algoritmo (en particular, las estructuras de control utilizadas y el por qué de las mismas), hasta entender qué parte del código implementa el criterio con el que se determina si un postulante ”es mejor” que otro; ya que ese criterio es lo único que cambia.

5. Sin criterio de desempate

> Se quiere entregar una beca a los dos mejores egresados de una carrera. Desarrolle un algoritmo que permita ingresar el nombre completo y el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. El programa debe finalmente informar los nombres de los dos postulantes con mejor promedio.

Ayudas

- Va a necesitar dos variables auxiliares para guardar los dos mejores promedios. La solución es mucho más simple cuando el mejor promedio va siempre en una de las variables, y el segundo en otra (es decir, que las dos no sean los dos mejores en cualquier orden, sino que siempre sepa cuál es el primero y cuál el segundo).
- Al probar/analizar el algoritmo, considere los siguientes casos con 3 postulantes: cuando los promedios sean {90, 70, 80}, y cuando sean { 80, 70, 90 }

4. Solo un becado

> Se quiere entregar una beca al mejor egresado de una carrera. Desarrolle un algoritmo que permita ingresar el nombre completo y el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. El programa debe finalmente informar el nombre del postulante con mejor promedio.

Ayuda:

- **Si resolvió antes la versión 3**: Identifique en qué *momento* (en el código, en qué lugar) su algoritmo determina que un postulante tienen el mejor promedio, ya que *ese* es el lugar/momento de guardar sus datos para informar luego.

3. Sin nombres

> Se quiere encontrar el mejor promedio de una carrera. Desarrolle un algoritmo que permita ingresar el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario ingrese 0 en lugar de un promedio. El programa debe finalmente informar el mayor promedio.

Ayuda:

- Recuerde que el algoritmo solo *verá* una nota por vez; en cada iteración se tiene una sola nota, ya que cada vez que se lee una nueva se pierde la anterior [1]. Póngase usted en lugar del intérprete, lea una lista larga de

[1] recuerde que se planteó como un ejercicio de la unidad 3, no es necesario utilizar arreglos.

nros mirando solo uno por vez, y analice cómo hace para saber el mayor al final, aunque no recuerde la lista completa.
- **Si resolvión antes la versión 2**: Notar que el ”procesamiento” de cada nota irá en lugar del mensaje ”Ok”.

2. Sin resultados

> Desarrolle un algoritmo que permita ingresar el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario ingrese 0 en lugar de un promedio. El programa debe mostrar el mensaje ”Ok” luego de leer cada promedio (pero no cuando se ingrese 0).

Ayudas:

- Notar que si el usuario quiere cargar, por ej, 10 promedios, en realidad ingresará 11 datos (10 promedios y un 0).
- Debe evitar mostrar el mensaje ”Ok” cuando se ingrese un 0. Para ello debe asegurarse de que siempre exista una condición que verifique si el promedio es o no 0 entre la lectura del mismo y la escritura de ”Ok”.

1. Cantidad fija

> Desarrolle un algoritmo que permita ingresar primero la cantidad de postulantes, y luego el promedio logrado en su carrera de cada uno de ellos. El programa debe mostrar los mensajes ”Bienvenido!” al principio, y ”Adios!” al final.

Este ya es un caso demasiado básico. **Si no resuelve rápidamente esta carga de datos, debería primero volver a la teoría básica** de la unidad 3 (o tal vez la 2?), antes de intentar continuar con la práctica.

Resumiendo, el alumno debe...

Intentar resolver una versión del problema (recorriendo la lista en orden) y:

- Si no lo logra, pasar a la siguiente versión, que será un poco más simple. Si no lo logra, seguir avanzando, hastas llegar a una versión que pueda resolver sin más ayudas.
- Cuando logre resolver una vesión, leer la explicación de la solución (de esa misma versión) para reforzar la lógica y la teoría subyacente; y luego volver a intentar la versión anterior, hasta resolver la primera y más compleja.

En cada caso, recuerde comenzar primero por el **análisis** del enunciado; asegurarse de comprender claramente cuáles son los datos que se tienen, y cuáles los resultados esperados.

Soluciones

En la segunda parte del documento, se explica cómo arribar a la solución de cada versión en orden inversio al de la lista; es decir, comenzando desde la versión más simple, y resolviendo luego las más complejas. En cada caso, agregando algo a la solución anterior.

Como ya se dijo anteriormente, **se recomienda que el alumno lea cada una de estas explicaciones, solo después de que haya logrado resolver esa versión del problema**.

1. Cantidad fija

> Desarrolle un algoritmo que permita **ingresar primero la cantidad de postulantes, y luego el promedio logrado en su carrera de cada uno de ellos**. El programa debe mostrar los mensajes ”Bienvenido!” al principio, y ”Gracias, vuelvan pronto!” al final.

Una vez identificados los datos y resultados, en cualquier versión del problema, lo primero en el algoritmo seguramente será la carga de esos datos. Esta versión súper-simplificada del problema incluye solo eso, solo la carga de datos. Y la carga misma está simplificada: aquí podemos preguntar cuántos datos hay y plantear una lectura sabiendo a priori la cantidad de promedios a leer. Esta versión 1 es tan simple que casi que debería llevar más tiempo escribir la solución que pensarla [2]. Pero justamente por ser tan simple y habitual, es común que ”salga de memoria”, y aquí vamos a razonarla.

Los datos serán entonces primero una cantidad (la cantidad de postulantes), y luego una nota por cada postulante. Leer un solo dato (la cantidad, desde ahora `CantTotal`) es simple: `Leer CantTotal`. Leer luego muchos datos, es un poquito más complicado. Uno puede optar por hacer muchas lecturas escribiendo muchas veces `Leer` (si sabe cuántas), o escribiendo el `Leer` una sola vez, pero dentro de una estructura de control que se encargue de ejecutarlo muchas veces (una estructura *repetitiva/iterativa*). *Obviamente* iremos por la segunda opción.

El problema se puede resolver con cualquier de las estructuras iterativas vistas: `Mientras`, `Repetir` o `Para`. Tal vez la más directa en este caso sería `Para`, pero se ve recién en la unidad 4, así que tomemos para este ejemplo `Repetir`:

```
Escribir "Bienvenidos!"
Leer CantTotal
Repetir
    Leer UnPromedio
Hasta Que <alguna_condición>         //<< falta completar aquí
Escribir "Gracias, vuelvan pronto!"
```

Las acciones de escribir, y la lectura de la cantidad, son cosas que se deben hacer una sola vez (algunas al principio, otras al final, pero solo una vez cada una). Por esto es importante que estén fuera de la estructura repetitiva. La estructura repetitiva es solo para la lectura de los promedios.

Llamo a la variable de la lectura `UnPromedio` para enfatizar que en cada momento (iteración) tengo allí solo uno, por más que a lo largo del algoritmo vayan pasando todos: cada vez que lea uno nuevo, reemplaza al anterior (se pierde el anterior).

Hay que pensar entonces en una condición adecuada, que garantice que esa lectura se va a repetir exactamente la cantidad de veces que queremos, que para esta altura (cuando lleguemos al `Repetir`) ya se conoce y ese número está en la variable `CantTotal`. Entonces, para saber si hay que seguir leyendo o no en cada momento, hay que saber si la cantidad de lecturas hechas llega a esa cantidad total. Pero no tenemos todavía nada que indique la cantidad de lecturas hechas como para poder plantear esa condición. Por esto, y recién ahora, decidimos agregar un contador, que llamaremos `CantLecturas` [3].

```
Escribir "Bienvenidos!"
Leer CantTotal
```

[2] si no es el caso, tal vez deba volver a repasar desde la unidad 2

[3] Como esta versión es tan simple, y probablemente ya hicieron muchos ejercicios donde se lee sabiendo la cantidad, es muy común que el alumno, adelantándose al problema, haya comenzado la codificación inicializando el contador. Pero si estamos tratando de explicarlo desde cero, ese contador no tiene justificación evidente hasta no necesitar la condición del `Repetir`.

```
CantLecturas ← 0                        // inicializar el contador
Repetir
    Leer UnPromedio
    CantLecturas ← CantLecturas + 1     // incrementarlo con cada lectura
Hasta Que CantLecturas=CantTotal        // condición completa
Escribir "Gracias, vuelvan pronto!"
```

Para implementar correctamente el contador es importante, primero saber qué representa. Dado que representa la cantidad de lecturas de promedios *ya hechas*, debe comenzar en 0 (y no en 1, ya que cuando el programa comienza, no hay ningún promedio cargado todavía); y la condición del repetir dirá que iteremos hasta que la cantidad de promedios leídos llegue a la cantidad total (hasta que ambas cantidades sean *iguales*). Obviamente la inicialización va fuera del `Repetir` (una sola vez, y antes de necesitar el valor de la variable para evaluar la condición), y no hay que olvidar el incremento dentro (siempre junto a la lectura de `UnPromedio`, para asegurarnos de que realmente cuente esas y solo esas lecturas).

Alternativamente se podría plantear lo mismo con un `Mientras` y el resto del algoritmo, en este caso, no cambiaría:

```
Escribir "Bienvenidos!"
Leer CantTotal
CantLecturas ← 0
Mientras CantLecturas<CantTotal Hacer    // condición "opuesta" a la del Repetir
    Leer UnPromedio
    CantLecturas ← CantLecturas + 1
FinMientras
Escribir "Gracias, vuelvan pronto!"
```

2. Sin resultados

> Desarrolle un algoritmo que permita ingresar el promedio logrado en su carrera de cada uno de los postulantes. **Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario ingrese 0 en lugar de un promedio**. El programa debe mostrar el mensaje ”Ok” luego de leer cada promedio (pero no cuando se ingrese 0).

En esta versión, lo importante a notar es que ahora se desconoce la cantidad de promedios que hay que leer. Entonces la condición del `Repetir` (o del `Mientras`) ya no dependerá de la cantidad de lecturas realizadas. En lugar de eso, el programa debe detectar un valor especial que cuando se ingrese le indique que debe finalizar la lectura.

Por ej, si los promedios a cargar son 70, 80 y 90; el usuario ingresará 70, 80, 90 y 0. El cuarto valor (0) no es un promedio, sino la indicación de que finaliza la carga[4].

Podríamos entonces *repetir* la lectura *hasta que llegue ese 0*, y esta oración es casi pseudocódigo:

```
Repetir
    Leer UnPromedio
    Escribir "Ok"
Hasta que UnPromedio=0
```

[4] Ese valor especial tiene que ser tal que no se confunda con un promedio real (supongamos que los promedios van de 1 a 100), pero del mismo tipo, ya que se cargará mediante la misma acción de lectura y variable.

Pero si agregamos el mensaje ”Ok” (que según el enunciado ahora debe aparecer luego de cada promedio válido) luego de la lectura, lo estaremos mostrando aún cuando se ingrese el 0. Para evitar esto, haremos que el ”Ok” no salga siempre *incondicionalmente*, sino sólo cuando el promedio sea válido. Para esto (decir que a veces sí y a veces no), tendremos que poner ese `Escribir` dentro de un condicional `Si...Entonces`:

```
Repetir
    Leer UnPromedio
    Si UnPromedio!=0 Entonces     // Evitar el "Ok" cuando ingresen 0
        Escribir "Ok"
    FinSi
Repetir Hasta Que UnPromedio=0
```

Es importante que inmediatamente después de leer `UnPromedio` (y antes de mostrar el ”Ok”) esté la pregunta que verifica si en verdad es un promedio, o es el valor especial que indica la salida. Por eso en este caso debemos repetirla: el `Si...Entonces` tiene casi la misma condición que el `Repetir`, pero es necesario porque antes de llegar a la condición del `Repetir` tenemos decidir si mostrar el ”Ok”.

Veamos otra forma de resolver lo mismo, que evita repetir la pregunta, ahora usando un `Mientras`. Podemos tratar de hacer que la condición del `Mientras` sea el único lugar donde verificamos si el promedio es válido (no `0`). Entonces, si entramos al `Mientras` seguro tendremos un promedio válido, y podremos mostrar el ”Ok” incondicionalmente. Para ello debemos notar que hay dos formas de llegar a la condición del `Mientras`: la primera vez llegaremos desde la instrucción previa al `Mientras`; las siguientes veces desde la última instrucción contenida dentro del `Mientras` (esto puede ser mucho más evidente mirando el diagrama de flujo). Entonces, en ambos lugares hay que hacer la lectura de `UnPromedio`, para que siempre inmediatamente después de la lectura se evalúe la condición:

```
Leer UnPromedio                // lectura para el 1er dato
Mientras UnPromedio!=0 Hacer
    Escribir "Ok"
    Leer UnPromedio            // lectura del 2do en adelante, incluyendo al 0 final
FinMientras
```

En la versión del `Repetir` debimos escribir 2 veces casi la misma pregunta; aquí dos veces la misma lectura; en principio ninguna es mejor ni peor que la otra[^estructurada].

3. Sin nombres

> Se quiere encontrar el mejor promedio de una carrera. Desarrolle un algoritmo que permita ingresar el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario ingrese 0 en lugar de un promedio. **El programa debe finalmente informar el mayor promedio**.

Una vez resuelta la lógica de la carga de datos, agregamos un resultado que calcular: el mejor promedio. Aquí hay que notar, como decía la ayuda, que cada vez que leemos un promedio nuevo, perdemos el anterior; nunca tenemos todos los promedios. Entonces *¿cómo puedo asegurarme de que el décimo promedio es el mejor, si no puedo recordar los 9 anteriores?* La respuesta más simple de lo que parece, y surge de la otra ayuda, de ponerse en el lugar del intérprete: haga el ejercicio de leer la siguiente lista mirando solo una vez cada número y tratar de recordar el mayor:

```
10, 13, 27, 15, 43, 12, 54, 39, 41, 47, 21, 52, 15, 35, 40, 53, 20
```

Seguramente con una sola lectura pudo reconocer el mayor valor sin necesidad de memorizar toda la lista. Si analiza cómo lo hizo en su cabeza, concluirá que no hace falta recordar todos números. Volviendo a la pregunta *¿Cómo puedo asegurarme de que el décimo promedio es el mejor, si no puedo recordar los nuevo anteriores?* Pues no hace falta recordar los nueve anteriores, sino solo el mayor de ellos. Aquí por ej, el mayor de los nueve primeros es 52. Si el décimo número (41) no es mejor que el mayor de los nueve anteriores (54), entonces tampoco será necesario recordar ese décimo. Pero si el décimo le ganase al mejor de los nueve anteriores (y por endse a todos ellos), entonces ese décimo promedio sería el único valor importante a recordar a partir de ahora (podríamos *olvidar* al 54).

Intentemos escribir esto en pseudocódigo. Necesitaremos una variable más para ”recordar” el mejor de los anteriores, que llamaremos `PromedioMejor`. Partiendo de la lectura de la versión 2, reemplazamos el ”Ok” por este análisis:

```
Leer UnPromedio
Mientras UnPromedio!=0 Hacer
    Si UnPromedio>PromedioMejor Entonces  // ¿"este" promedio es el mejor (hasta ahora)?
        PromedioMejor ← UnPromedio        // si es así, lo guardamos
    FinSi                                 // si no, nada
    Leer UnPromedio
FinMientras
Escribir PromedioMejor                    // mostrar el mejor de todos
```

El bucle es correcto, y de esta forma intentamos tener en cada momento en `PromedioMejor` el mejor promedio ”hasta ahora” (de los ingresados previamente). Cuando ya se hayan ingresado todos los datos (luego de finalizado el `Mientras`), `PromedioMejor` será el mejor de todos y podremos informarlo.

Pero queda un problema por detectar y resolver. La primera vez que lleguemos al `Si...Entonces` no se podrá evaluar la condición porque la variable `PromedioMejor` todavía no existirá (no se asigna ni lee antes del `Si...Entonces`). Para solucionar esto podemos pensarlo como una caso especial para el `Si...Entonces` (lo cual nos hará complicar la condición o anidar otro condicional), o mejor aún, inventar un valor inicial para `PromedioMejor` que no ”moleste”. La segunda alternativa resulta más simple porque no hay que modificar el bucle, solo inicializar esa variable antes del mismo.

La siguiente pregunta lógica es ¿con qué valor inicial? Dado que el único objetivo de ese valor es evitar el error en la primera iteración, debe ser un valor que no altere el resultado. Si la inicializamos en `500`, la condición `UnPromedio>PromedioMejor` nunca será verdadera, el valor nunca cambiará, y nuestro algoritmo mostrará erróneamente a `500` como si fuera el mejor promedio. Es importante que sea un valor que haga que la condición `UnPromedio>PromedioMejor` resulte verdadera. Para ello, si estamos buscando un mayor, el valor inicial debe ser muy ”pequeño”. Como estamos hablando de números positivos, algo como -1 o 0 es suficiente[5]. En nuestro caso queda:

```
PromedioMejor ← 0                         // valor inicial pequeño
Leer UnPromedio
Mientras UnPromedio!=0 Hacer
    Si UnPromedio>PromedioMejor Entonces
        PromedioMejor ← UnPromedio
    FinSi
```

[5] Si los datos incluyeran negativos, habría que usar algo todavía más negativo (-999999?).

```
    Leer UnPromedio
FinMientras
Escribir PromedioMejor
```

4. Solo un becado

> Se quiere entregar una beca al mejor egresado de una carrera. Desarrolle un algoritmo que permita **ingresar el nombre completo y el promedio logrado** en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. **El programa debe finalmente informar el nombre** del postulante con mejor promedio.

En esta versión se agregan los nombres de los postulantes a los datos (cada nombre antecede como entrada a cada promedio); y con estos datos el programa debe mostrar ahora el nombre (ya no el promedio) del mejor postulante.

Si mantemos esa idea de que en cada iteración tenemos los datos de solo un postulante, y que hay que procesar esos datos inmediatamente después de leerlos (porque en la próxima iteración se van a perder, serán reemplazados por los de otro postulante), las modificaciones son un poco más obvias.

Primero, donde se leía solo el promedio, ahora se deben leer nombre y promedio; y el valor especial de salida no será un 0 en el promedio, sino un nombre en blanco[6]:

```
...
Leer UnNombre, UnPromedio                  // leer nombre y promedio
Mientras UnNombre!="" Hacer
    ...ver si es el mayor...
    Leer UnNombre, UnPromedio              // leer nombre y promedio
FinMientras
...
```

Esto está bastante bien, pero se puede hacer una pequeña mejora. Ya que la condición de salida depende solo del nombre, ¿para qué pedir una nota cuando se deje el nombre en blanco? Actualmente el usuario debe dejar el nombre en blanco y además poner cualquier nota para poder salir. Evitemos leer la nota si el nombre está en blanco. Para ello separamos las lecturas y ponemos la de la nota solo luego de verificar la condición de validez del nombre (la condición del `Mientras`):

```
...
Leer UnNombre                              // leer solo nombre
Mientras UnNombre!="" Hacer
    Leer UnPromedio                        // leer promedio para el nombre anterior
    ...ver si es el mayor...
    Leer UnNombre                          // leer solo nombre
FinMientras
...
```

Ahora que tenemos los datos, hay que modificar cómo informamos el resultado. Vamos a querer escribir el nombre asociado al mejor promedio, así que inventemos una variable para guardarlo, por ej `NombreMejor`. Esto es lo que mostrará el último `Escribir`, pero ¿dónde y cómo se le asigna un valor?

[6] El usuario presionará *enter* sin ingresar nada. En el código, este valor será la cadena vacía “”“.

La clave es entender en qué momento/lugar nuestro algoritmo detecta y guarda al mejor promedio. Actualmente lo dectecta con el `Si....Entonces`, cuando da verdadero, y allí dentro es donde guarda el mejor promedio. Si queremos guardar algo más de ese mismo alumno además de su promedio, allí (dentro de la rama por `Verdadero` del `Si...Entonces`) es el lugar para hacerlo:

```
PromedioMejor ← 0
Leer UnNombre
Mientras UnNombre!="" Hacer
    Leer UnPromedio
    Si UnPromedio>PromedioMejor Entonces
        PromedioMejor ← UnPromedio
        NombreMejor ← UnNombre            // guardar el nombre además del promedio
    FinSi
    Leer UnNombre
FinMientras
Escribir NombreMejor                      // mostrar el nombre en lugar del promedio
```

Notar que no hace falta inicializar previamente a la variable `NombreMejor` como lo hicimos con `PromedioMejor`. En aquel caso era necesario porque se usaba en la condición del `Si...Entonces`, antes de la asignación. Pero `NombreMejor` no participa en la condición, sino que será asignada directamente, así que no necesita un valor previo.

Por otro lado, también se debe notar que por más que ya no se muestre el mejor promedio, se debe guardar igual, porque sigue siendo necesario para la condición del `Si...Entonces`, eso no ha cambiado.

5. Sin criterio de desempate

> Se quiere entregar una beca a los dos mejores egresados de una carrera. Desarrolle un algoritmo que permita ingresar el nombre completo, y el promedio logrado en su carrera de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. **El programa debe finalmente informar los nombres de los dos postulantes con mejor promedio.**

Esta penúltima versión difiere de la anterior en que pide informar los dos mejores alumnos, y no solamente el mejor de todos. Repasamos cómo encontramos el mejor de todos en la versión anterior:

- Una variable auxiliar (`PromedioMejor`) debía guardar en todo momento el mejor promedio entre los datos leidos hasta ese momento; para que cuando llegue el siguiente dato podamos comparar solo con ese auxiliar y no necesitemos toda la lista previa completa.
- Antes del bucle, inicializamos la variable con un valor muy bajo (`0`) para que el primer promedio ya ”le gane” y tome su lugar. Luego del bucle, el valor que quede en esa variable será el mayor de todos.
- Al momento de guardar el mejor promedio (dentro del `Si...Entonces`), guardamos además el nombre asociado, porque es lo que vamos a querer informar al final.

Para encontrar los dos mayores en lugar de solo el mayor, la parte que cambia esencialmente es el `Si...Entonces` que hay dentro del bucle.

Cuando se busca un solo mayor, cada promedio que se lee en el bucle tiene dos posibles destinos: o es el mayor de todos (hasta ahora) y hay que guardarlo; o no lo es y entonces no importa (no se guarda ni se hace nada más con ese dato). Un `Si...Entonces` con la rama del `Falso` vacía (en el código, sin el `SiNo`) representa perfectamente esta situación. Pero ahora cada valor que se lea tendrá tres posibles destinos: ser el mejor de todos, el segundo mejor, o que sea menor a ambos y no importe.

Como un `Si...Entonces` solo permite elegir entre dos posibles caminos[7], para generar 3 agregaremos un segundo `Si...Entonces` *anidado*[8].

```
...
Mientras ... Hacer
    Leer ...
    Si <va primero?> Entonces
        ...                         // es el mejor
    Sino
        Si <va segundo?> Entonces
            ...                     // es el 2do mejor
        SiNo
            ...                     // 3ro, 4to, 5to, etc
        FinSi
    FinSi
    ...
FinMientras
...
```

Como ahora nos interesa guardar dos promedios, tendremos que agregar un segundo auxiliar (dos en realidad, promedio y nombre). Aquí es importante pensar que entre los dos auxiliares que guardan los dos mejores promedios hay un orden: saber que uno siempre guarda al primero, y otro siempre al segundo (y no que entre ambos tienen los dos, pero cualquiera en cualquiera). Establecer este orden simplifica lo que ocurre en los `Si...Entonces` que analizan cada promedio. Llamemos a este segundo auxiliar para el promedio `PromedioSegundo`, y de la mano del mismo vendrá `NombreSegundo` para guardar también el nombre asociado a ese promedio.

Entonces cuando leemos un promedio, podríamos tener que guardarlo en `PromedioMejor` si es el mayor de todos, en `PromedioSegundo` si es el segundo mayor, o no guardarlo en otro caso.

- Si planteamos que la rama verdadera del `Si...Entonces` más externo corresponda al caso en que va primero, la condición para este podría ser simplemente `UnPromedio>PromedioMejor`. Es decir, tal como estaba, sin considerar explícitamente `PromedioSegundo`, porque ya establecimos que `PromedioSegundo` será menor que `PromedioMejor`, por lo cual, si un valor es mayor a `PromedioMejor`, también será mayor a `PromedioSegundo`.
- Si esa primera pregunta dio `Falso`, entonces ya hemos descartado la posibilidad de que vaya primero, así que solo queda definir entre dos opciones: va segundo, o no importa. Por eso, en la rama del falso del primer `Si...Entonces` agregamos un segundo `Si...Entonces` para definir entre estas dos alternativas con la condición `UnPromedio>PromedioSegundo` (aquí ya no participa `PromedioMejor` por lo que acabamos de decir: al estar en el `Falso`, ya lo hemos descartado).

Con esto, normalmente llegamos a:

```
PromedioMejor ← 0
PromedioSegundo ← 0
Leer UnNombre
Mientras UnNombre!="" Hacer
    Leer UnPromedio
```

[7] porque la condición, al ser una expresión de tipo lógico, solo puede dar dos posibles resultados: `Verdadero` o `Falso`.

[8] Cuidado! si no fuera anidado, por la forma del algoritmo, parecerían independientes y tendríamos 4 combinaciones de caminos, y no 3 como queremos (para analizar los "caminos" puede ser mejor ver el diagrama de flujo).

```
    Si UnPromedio>PromedioMejor Entonces        // es el mayor?
        PromedioMejor ← UnPromedio
        NombreMejor ← UnNombre
    Sino                                        // no es el mayor
        Si UnPromedio>PromedioSegundo Entonces  // es el segundo?
            PromedioSegundo ← UnPromedio
            NombreSegundo ← UnNombre
        FinSi                                   // 3ro, 4to, etc no interesa
    FinSi
    Leer UnNombre
FinMientras
Escribir NombreMejor
Escribir NombreSegundo
```

Este algoritmo ya está muy cerca de la solución final, pero todavía hay un problema.

Pensemos en un caso donde hay 3 alumnos con promedio 80, 70 y 90. Si hacemos un seguimiento, veremos que al leer el 90 irá primero (será mayor al 0 con que inicializamos `PromedioMejor` y reemplazará su valor). Al leer el 70 irá segundo (no será mayor al 80 que hay en `PromedioMejor` pero sí que el 0 con que inicializamos `PromedioSegundo`). Y finalmente al leer el 90 irá primero (será mayor al 80 con que había en `PromedioMejor` , por lo que lo reemplazará). Entonces como resultados finales tendremos 90 y 70. ¿Dónde está el 80?

Si analizamos la última iteración, donde los dos mayores eran 80 y 70, y el dato nuevo a comparar era 90, reemplazar simplemente 80 por 90 genera el error. No se debe descartar el mayor anterior (80), sino que debe pasar a ocupar el segundo lugar. Y esto hay que hacerlo antes de perder ese valor:

```
PromedioMejor ← 0
PromedioSegundo ← 0
NombreMejor ← ""                            // hay que inicializar solo el primer nombre
Leer UnNombre
Mientras UnNombre!="" Hacer
    Leer UnPromedio
    Si UnPromedio>PromedioMejor Entonces    // es el mayor?
        PromedioSegundo ← PromedioMejor     // el anterior 1ro, ahora será el 2do
        NombreSegundo ← NombreMejor
        PromedioMejor ← UnPromedio          // el nuevo dato toma el 1er lugar
        NombreMejor ← UnNombre
    Sino
        Si UnPromedio>PromedioSegundo Entonces
            PromedioSegundo ← UnPromedio
            NombreSegundo ← UnNombre
        FinSi
    FinSi
    Leer UnNombre
FinMientras
Escribir NombreMejor
Escribir NombreSegundo
```

El detalle final es la inicialización de `NombreMejor`. En la versión 5 dijimos que no hacía falta inicializarlo porque en ningún momento preguntábamos por su valor antes de asignarlo. Pero esa situación cambió, ya que el código que

agregamos recién hará que en la primera iteración, antes de asignar los datos del primer postulante a `PromedioMejor` y `NombreMejor`, se copien los datos del mayor ”anterior” (no leímos nada antes, así que estos datos serán los que hayamos inicializado antes del `Mientras`) al segundo lugar. Ahí es donde necesitamos que `NombreMejor` esté inicializado. Y debemos inicializarlo con una cadena (cualquiera, no importa, será reemplazada) y no por un número, porque es una variable de texto (para guardar un nombre).

6. Problema completo

> Se quiere entregar una beca a los dos mejores egresados de una carrera. Desarrolle un algoritmo que permita **ingresar el nombre completo, el promedio logrado en su carrera, y la cantidad de aplazos** en su historia académica de cada uno de los postulantes. Se desconoce la cantidad de postulantes, la carga de datos debe finalizar cuando el usuario deje el nombre en blanco. El programa debe finalmente informar los nombres de los dos postulantes con mejor promedio. **En caso de empate entre dos postulante con igual promedio, se debe seleccionar aquel con menor cantidad de aplazos.**

Finalmente llegamos a la versión completa, donde lo único que falta respecto a la versión 5 es agregar el criterio de desempate: si dos alumnos tienen el mismo promedio, hay que mirar la cantidad de aplazos.

Lo importante aquí es notar que no cambia la ”forma” del algoritmo: hay una estructura repetitiva para leer y procesar varios alumnos, y por cada alumno hay 3 posibles acciones (guardarlo como mejor, como segundo o ignorarlo) representadas en los 3 caminos que generan los dos `Si...Entonces` anidados. Todo esto era cierto en la versión 5, y sigue siendo cierto en la versión completa. ¿Qué cambia entonces?

Lo que cambia es el *criterio* con que se determina que un postulante es mejor que otro. Ese criterio antes era muy simple, pero ahora incluye una regla para desempatar. Deberíamos modificar el algoritmo intentando cambiar solo ese criterio y no toda la forma del mismo. Es decir, cambiando lo menor posible para no complicarnos por demás.

Y ¿dónde está ese ”criterio” simple representado en el código de la versión anterior? En las condiciones de los `Si...Entonces`. El primero compara si el postulante que se acaba de ingresar ”es mejor” que el que estaba guardado en el primer lugar. Si da falso, el segundo hace lo mismo con el segundo lugar. Esas dos condiciones son las dos implementaciones de ese criterio; esas dos condiciones son lo único que necesitamos modificar[9].

Tomemos por ejemplo la primera: `UnPromedio>PromedioMejor`. Estamos diciendo que la única forma de que un postulante tome el primer lugar, es que tenga mejor promedio. Pero el nuevo enunciado dice que hay dos posibilidades, también puede ser que tenga el mismo promedio pero menor cantidad de aplazos. La forma de agregar una segunda condición que permita entrar a la rama verdadera el `Si...Entonces` es concatenarla con el operador `O`. Cuando unimos dos condiciones con `O`, el resultado es `Verdadero` cuando cualquiera de las dos es `Verdadero`. Entonces, la parte de esa decisión queda:

```
...
    Si UnPromedio>PromedioMejor O <alternativa_empate> Entonces
        PromedioSegundo ← PromedioMejor
        NombreSegundo ← NombreMejor
        PromedioMejor ← UnPromedio
        NombreMejor ← UnNombre
    Sino
```

[9] Se podría haber implementado el criterio alternativo con más `Si...Entonces` anidados que traten el caso de empate por separado, pero el algoritmo (que podría dar el resultado correcto igualme) habría quedado más largo y complejo, y con bastante código repetido. Por eso preferimos esta versión, donde la mayor parte de la lógica es exactamente igual a la versión 5, y solo cambian las dos condiciones donde se comparan alumnos, nada más.

```
        Si UnPromedio>PromedioSegundo O <alternativa_empate> Entonces
            PromedioSegundo ← UnPromedio
            NombreSegundo ← UnNombre
        FinSi
    FinSi
...
```

Pensemos entonces como plantear la condición alternativa. Esa condición se da cuando empatan en promedio pero el nuevo postulante tiene menos aplazos. Ambas cosas (igual promedio y menos aplazos) deben ser ciertas; por esto usamos el operador `Y` para combinarlas. `Y` da `Verdadero` cuando a ambos lados del `Y` tengo expresiones verdaderas. Entonces, el segundo criterio sería `UnPromedio=PromedioMejor y CantAplazos<AplazosMejor` (suponiendo que agregamos las variables `CantAplazos` y `AplazosMejor` donde corresponda).

Al combinar esta segunda condición con la original, mediante el operador `O` como dijimos previamente, puede surgir alguna duda sobre la precedencia de los operadores `O` e `Y`[10]. Lo mejor es poner paréntesis para que quede claro[11].

```
PromedioMejor ← 0
PromedioSegundo ← 0
NombreMejor ← ""
AplazosMejor ← 0
Leer UnNombre
Mientras UnNombre!="" Hacer
    Leer UnPromedio, CantAplazos
    Si UnPromedio>PromedioMejor o (UnPromedio=PromedioMejor y CantAplazos<AplazosMejor) Entonces
        PromedioSegundo ← PromedioMejor
        NombreSegundo ← NombreMejor
        AplazosSegundo ← AplazosMejor
        PromedioMejor ← UnPromedio
        NombreMejor ← UnNombre
        AplazosMejor ← CantAplazos
    Sino
        Si UnPromedio>PromedioSegundo o (UnPromedio<PromedioSegundo
                                         y CantAplazos<AplazosSegundo) Entonces
            PromedioSegundo ← UnPromedio
            NombreSegundo ← UnNombre
            AplazosSegundo ← CantAplazos
        FinSi
    FinSi
    Leer UnNombre
FinMientras
Escribir NombreMejor
Escribir NombreSegundo
```

[10] Como cuando en matemáticas tenemos `A+B/C` y podemos preguntarnos si primero se hace `A+B` o primero `B/C`. Como estamos habituados a las expresiones matemáticas tenemos claro que primero se hace `B/C` (+ y - separan términos), a menos que se altere con paréntesis (`(A+B)/C`). Pero en cuanto a las expresiones lógicas, no es tan obvio si se resuelve primero el `Y` o el `O`.

[11] Notar que si se leen las condiciones, coinciden casi textualemente con lo que uno diría coloquialmente: "un postulante es mejor que otro si tiene mejor promedio o (en caso de empate, tiene menos aplazos)" se traduce en "`UnPromedio>PromedioMejor o (UnPromedio=PromedioMejor y CantAplazos<AplazosMejor)`".

Autoevaluación

De ninguna forma un solo ejercicio sirve de muestra para concluir cuán bien o mal está asimilando los contenidos de la materia y el resultado esperado en su parcial (más aún considerando que un ejercicio del parcial se parece más a un ejercicio de la unidad 4, no 3). Pero, si de todas formas intentamos extrapolar, podríamos decir que:

- Un alumno que promocione, debería resolver al menos la versión 5.
- Un alumno que regularice, las versiones 4 o 3, sin más ayudas.
- Si no ha podido resolver la versión 3 sin bajar hasta las simplificaciones 1 o 2, seguramente no habría aprobado el parcial, si hubiese sido hoy. Pero la buena noticia es que si el parcial no es en 3 días, está a tiempo de extraer de este ejercicio las pistas que le permitan encontrar dónde debe mejorar su método de estudio (vea las conclusiones).

Algunas conclusiones

El objetivo de este documento fue guiar al alumno en una forma de estudiar que evite problemas habituales. Hay alumnos que desaprueban porque no estudian, pero también hay alumnos que desaprueban porque estudian *mal* (y no *poco*). Para no caer en este segundo grupo es que se hace hincapié en la forma de procesar los ejemplos y ejercicios: importa y es mucho más difícil el proceso de desarrollo, antes que el resultado final.

En este mismo sentido, es importante destacar la necesidad de *razonar* ese desarrollo, y no de *memorizar* la solución. No estamos estudiando un *catálogo* de problemas tipo, sino adquiriendo conocimientos y habilidades que nos permitan enfrentarnos luego a problemas nuevos.

Paralelamente, también se mostró una estrategia para encarar problemas complejos: partir de entender bien versiones parecidas pero más simples, que nos permitan tener una buena base sobre la cual luego razonar cómo añadir la complejidad faltante. Y esto, la habilidad para entender y simplificar o descomponer un problema, es una habilidad muy importante de adquirir, y usualmente no somos consientes de cuándo la estamos practicando.

Casi cualquier problema, si se modifica o simplifica *adecuadamente* se reduce a un caso típico y básico de la práctica o de la teoría (o tal vez una combinación de dos o tres). Esto permite partir de un problema conocido (que ya resolvimos y analizamos más de una vez) para llegar de a poco y metódicamente al caso particular que plantee el enunciado. Encontrar la simplificación o descomposición adecuada no es fácil al principio, y la única forma de adquirir esta habilidad es practicarla. Cuantos más problemas resuelva el alumno, más fácil le será identificar la esencia y las particularidades de cada uno.