Tema VII

Tema VII: Optimización de Algoritmos.

Concepto. Objetivos. Factibilidad. Reglas de Jackson. Formas de optimización. Por afinación. Por algoritmos. Recursos. Tablas. Parámetros. Matemáticos. Modelos de clasificación. Distintos
Métodos. Eficiencia de algoritmos.

Prof. Oscar Adolfo Vallejos - 2011

9.1 Concepto

Optimizar un algoritmo implica construirlo lo mas correctamente posible, con estilo, transparente, sin errores y eficiente.

9.2 Objetivos

Los objetivos principales son que el algoritmo debe ser construido lo más **pequeño** posible o que sea lo más **rápido** en su ejecución, o de ser viable, ambos a la vez.

- Lo más **pequeño** posible: significa que tenga la menor cantidad de instrucciones posibles

- Lo más **rápido** posible: significa economizar el tiempo de ejecución del algoritmo en máquina.

9.3 Factibilidad

La optimización puede provocar escribirlo incorrectamente, puede resultar más difícil de entender y más costoso su mantenimiento, como así también más propenso a incorporar errores.

Todo esto cuesta mucho dinero por lo que debe justificarse económicamente la tarea de optimizar antes de emprenderla, de allí deriva la primer regla de *Jackson.*

En caso de justificarse económicamente, la actitud debe ser construir inicialmente un algoritmo claro y transparente, para luego optimizarlo, esto da lugar a la segunda regla de *Jackson*.

9.4 Reglas de Jackson

Regla N° 1: ***"No lo haga"***

Si no se puede justificar económicamente.

Regla N° 2: ***"No lo haga todavía"***

Si está justificada comenzar con un diseño no óptimo, esto en pro de la claridad y simplicidad, y a posteriori optimizarlo.

9.5 Formas de Optimizacion

Por afinación

Esto implica no modificar la estructura del algoritmo sino utilizar factores de bloque, segmentación de programas, asignación de memorias intermedias, etc.

Por algoritmos

La optimización por algoritmos se realiza a través de recursos como ser:

- ***Estructuras de datos*** (arreglos, pilas, colas, árboles, etc).
- ***Tablas***
- ***Matemáticos*** (por ejemplo para determinar pares e impares se utilizan los recursos: parte entera, resto, potencia de menos uno (-1), etc).
- ***Parámetros*** (por ejemplo para determinar la longitud de los arreglos, tope de una pila, frente y final de una cola, tasa de interés, etc.).

9.5 Formas de Optimizacion

La mayoría de los autores (*Jackson, Boria, Rice y Rice*) eligen los modelos de clasificación para ilustrar la optimización por algoritmos.

Para ilustrar el objetivo **más pequeño** se emplea el caso de *determinación de tipos de triángulos*.

Para ilustrar el objetivo **más rápido** se aprovechará el método de clasificación de intercambio directo comunmente llamado "burbuja" (desarrollado en el tema de clasificacion).

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

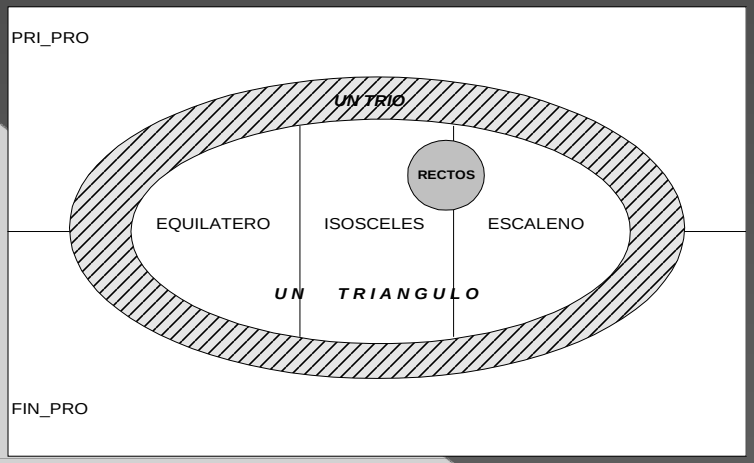
Dado un conjunto de tríos de valores A, B, C (mayores que cero) determinar, de entre los que forman triángulo, los distintos tipos (escaleno, isósceles, equilátero) y también cual de ellos es recto. Informar de acuerdo a la figura que contiene el modelo de la salida.

LADO 1	LADO 2	LADO 3	TIPO	RECTO
--	--	--	Escaleno	SI
--	--	--	No triángulo	
--	--	--	Equilátero	NO
--	--	--	Escaleno	SI

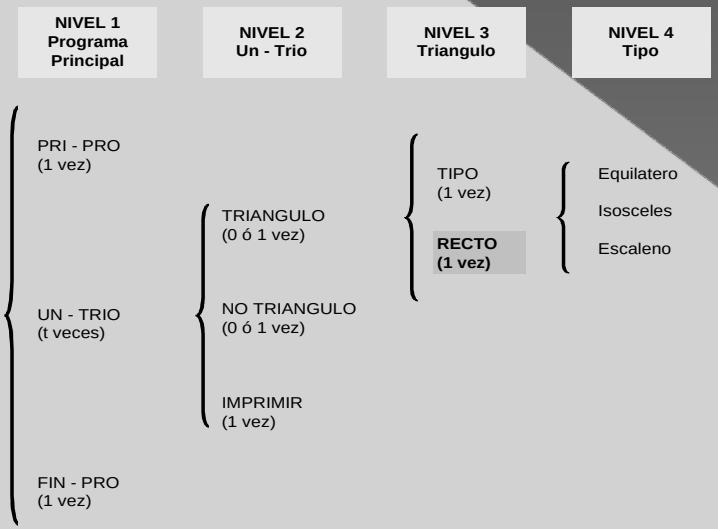
9.5 Formas de Optimizacion

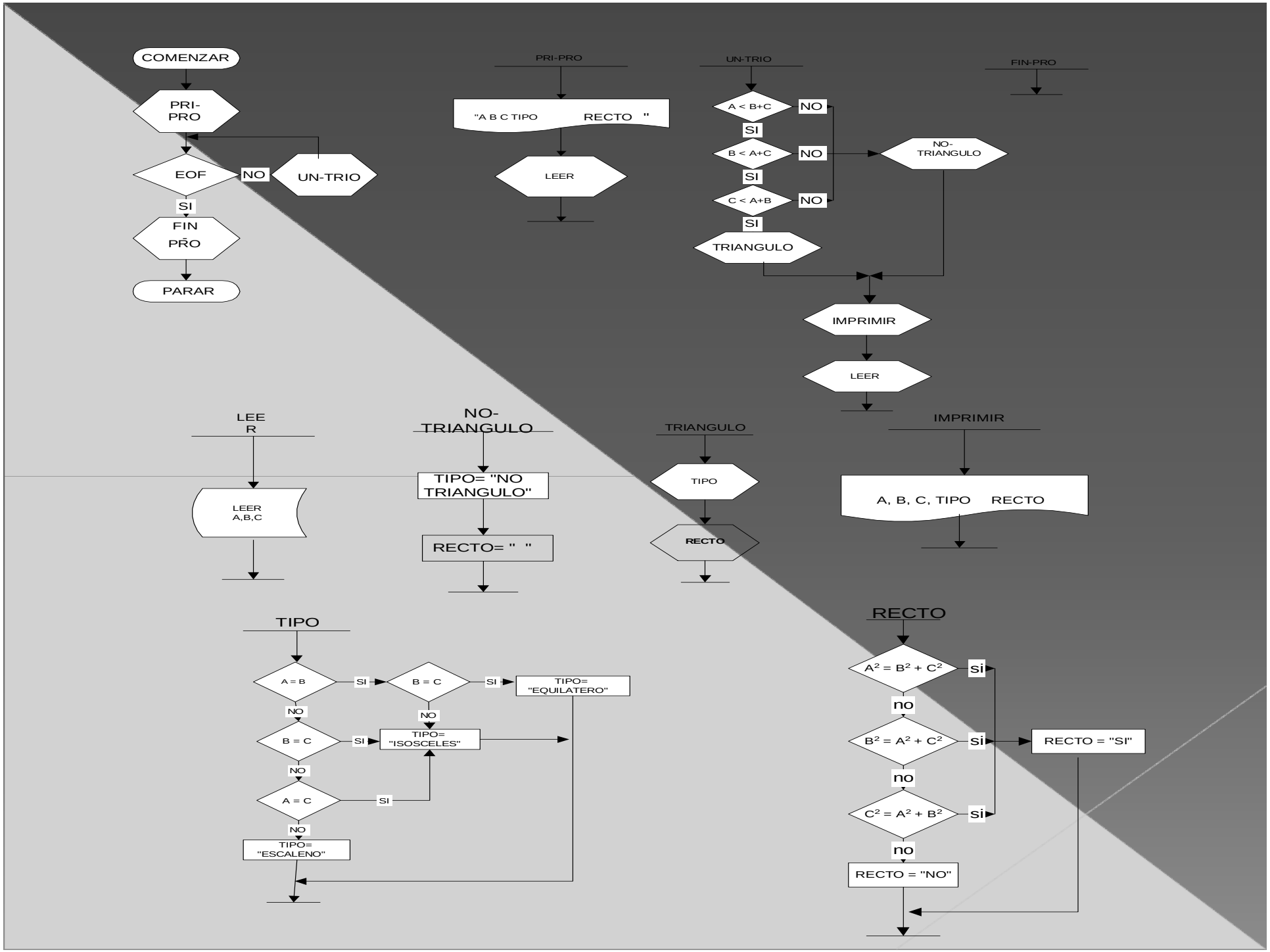
Optimización haciendo el algoritmo más *pequeño*

A)



B)

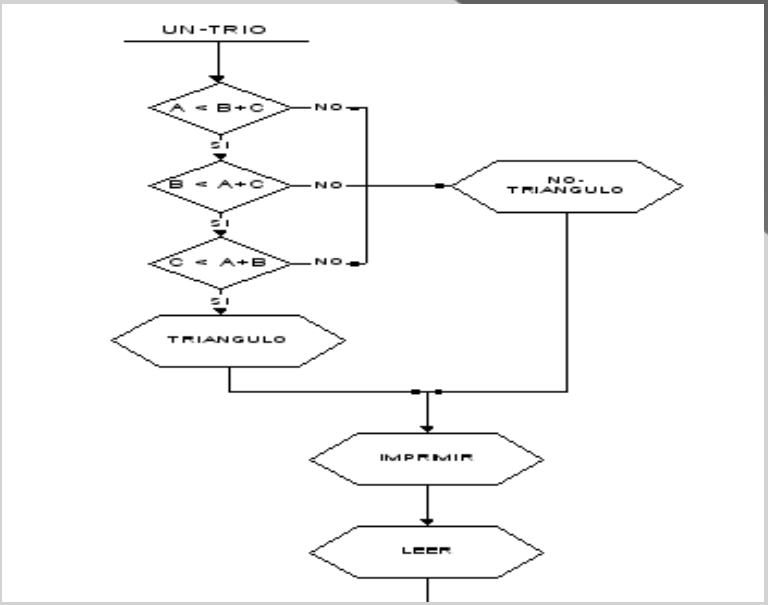



COMENZAR
PRI-PRO
EOF
NO
UN-TRIO
SI
FIN PRO
PARAR
PRI-PRO
"A B C TIPO RECTO "
LEER
UN-TRIO
A < B+C
NO
SI
B < A+C
NO
SI
C < A+B
NO
SI
TRIANGULO
NO-TRIANGULO
FIN-PRO
IMPRIMIR
LEER
LEE R
LEER A,B,C
NO-TRIANGULO
TIPO= "NO TRIANGULO"
RECTO= " "
TRIANGULO
TIPO
RECTO
IMPRIMIR
A, B, C, TIPO RECTO
TIPO
A = B
SI
B = C
SI
TIPO= "EQUILATERO"
NO
B = C
SI
TIPO= "ISOSCELES"
NO
A = C
SI
NO
TIPO= "ESCALENO"
RECTO
A² = B² + C²
Si
no
B² = A² + C²
Si
RECTO = "SI"
no
C² = A² + B²
Si
no
RECTO = "NO"

Optimización haciendo el algoritmo más *pequeño*

Seguidamente se procede a optimizar el algoritmo desarrollado en el ejemplo anterior para tratar de hacerlo más pequeño (menor cantidad de instrucciones), teniendo en cuenta lo siguiente:

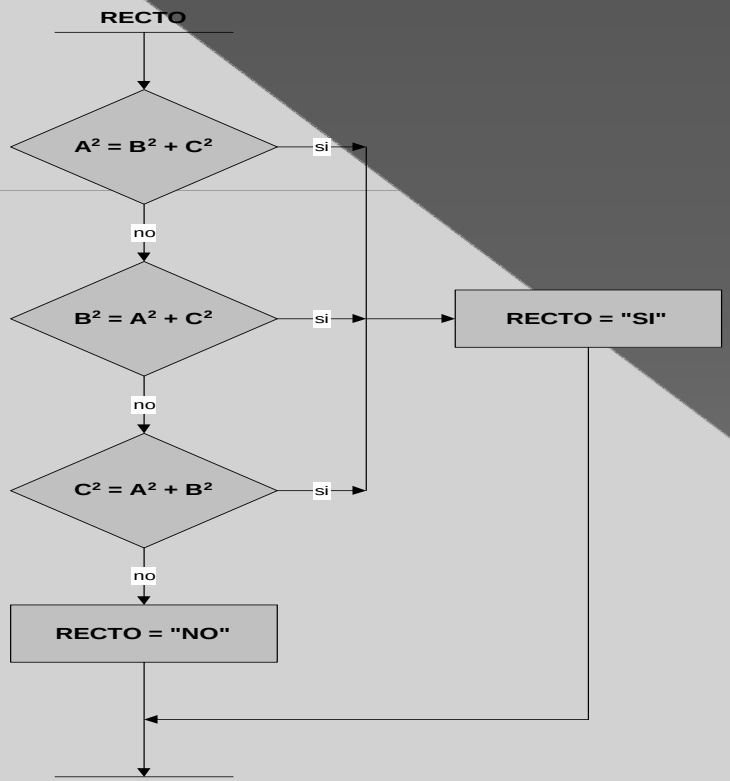
1) En la rutina **un-trío** se observa que la condición matemática de triángulo (un lado sea menor que la suma de los otros dos) se transforma en suficiente si el lado que se compara es el mayor de todos.



9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

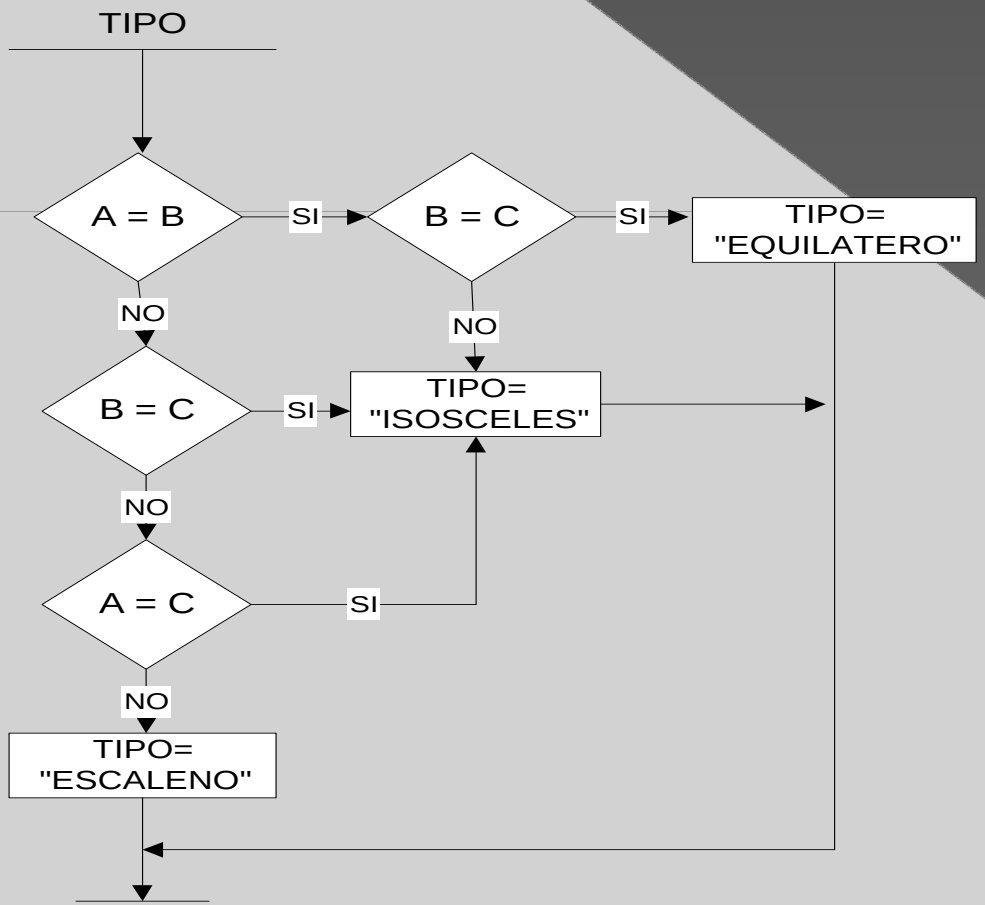
2) En la rutina de **recto** también la condición necesaria basada en *Pitágoras* ($A^2 = B^2 + C^2$) se transforma en suficiente si se sabe con anticipación cual es la hipotenusa, o sea el lado mayor.



9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

3) En la rutina **tipo** se advierte que deben compararse todos contra todos los lados para determinar el tipo de triángulo. Probablemente el tener juntos los posibles lados iguales, implique una simplificación del algoritmo.



9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

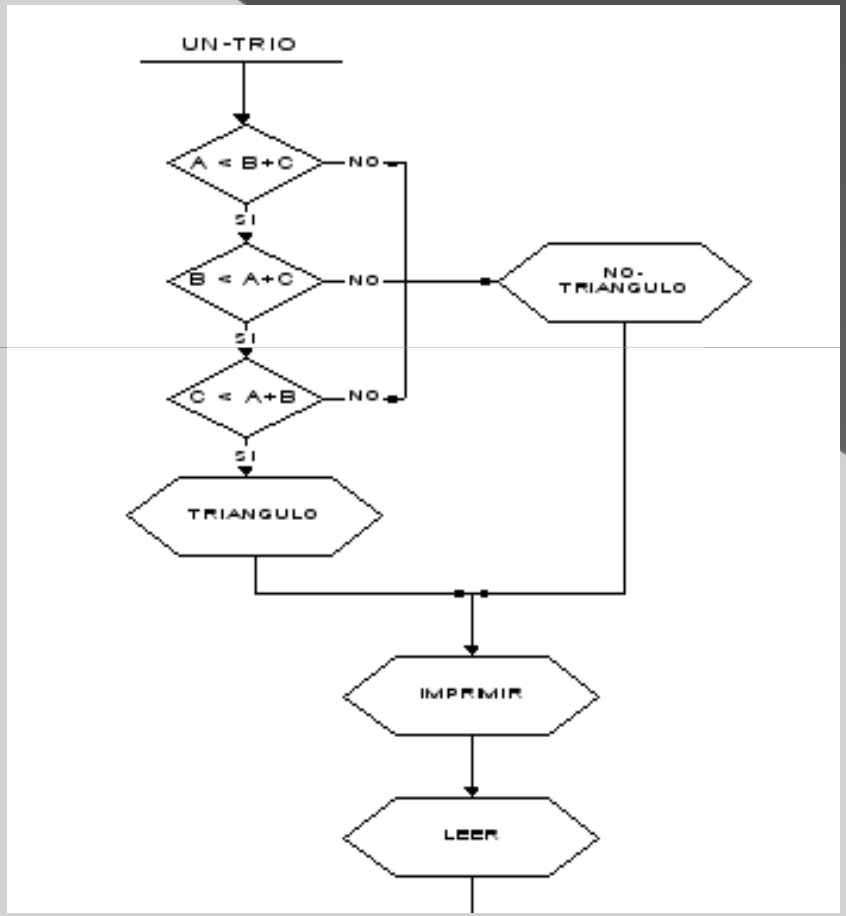
Transformar las condiciones necesarias con que fueron definidas inicialmente, en dos condiciones suficientes (puntos 1 y 2) y simplificar el análisis de **tipo** en el punto 3, se logra si los lados están clasificados en orden descendente.

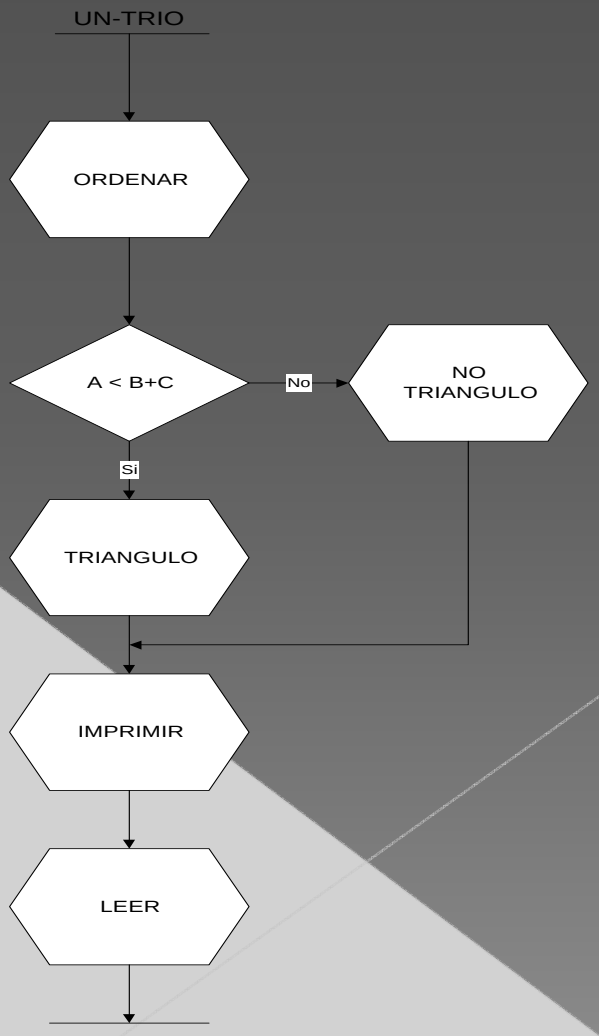
Aprovechando una rutina desarrollada que ordena en forma descendente un conjunto de números, se invoca a la misma al comienzo de la rutina **un_trío** y a partir de allí se optimiza el algoritmo con los lados ordenados, quedando como se indica a continuación las rutinas **un-trío , tipo y recto.**

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

Un_trío: Se suprimen las condiciones de B<A+C y C<A+B.

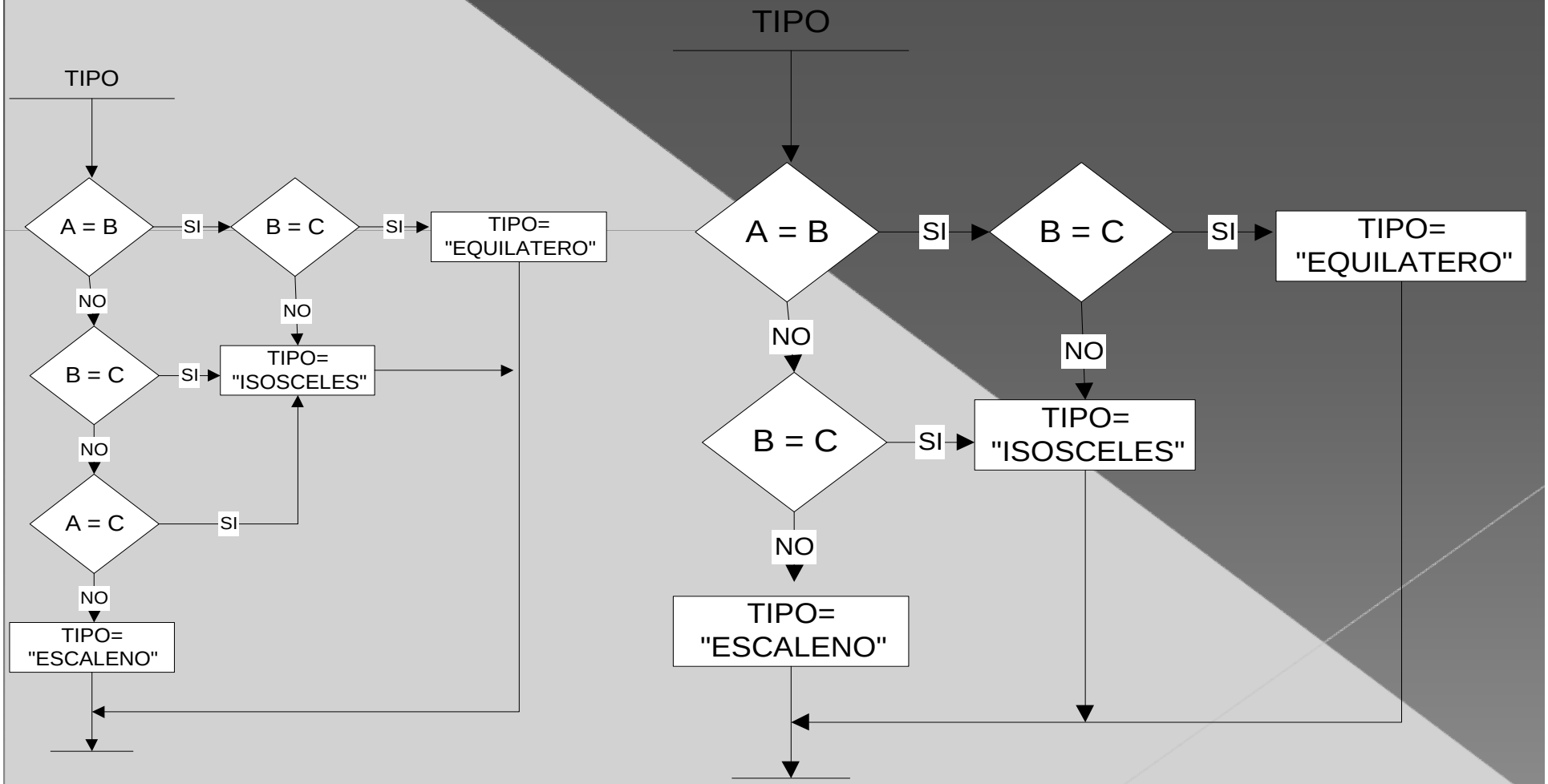




9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

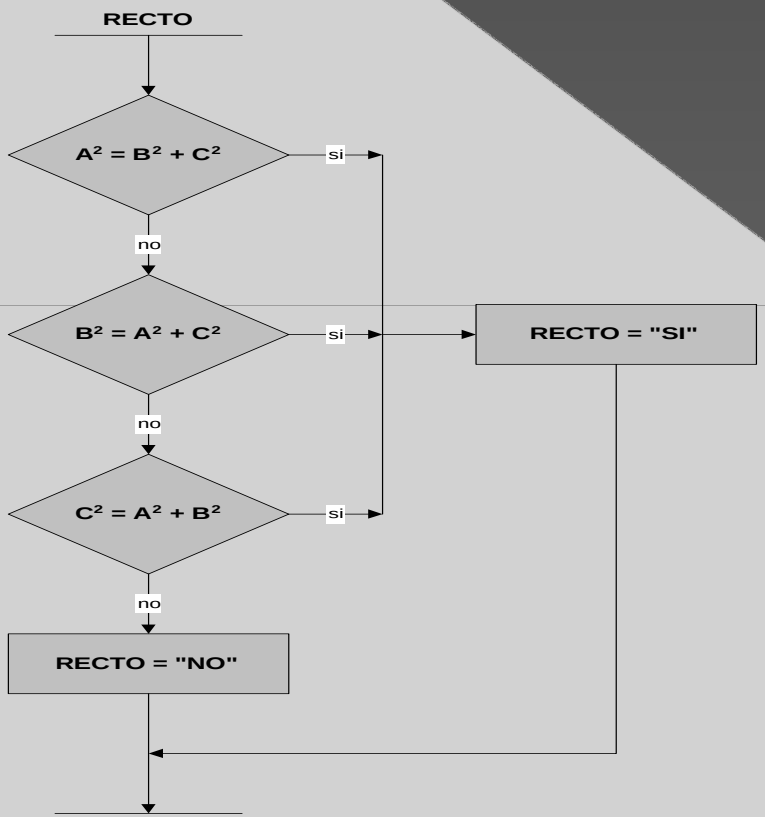
Tipo: En el caso de triángulos isósceles, al ordenar los lados, los que son iguales están contiguos, por lo que la condición A = C es innecesaria.

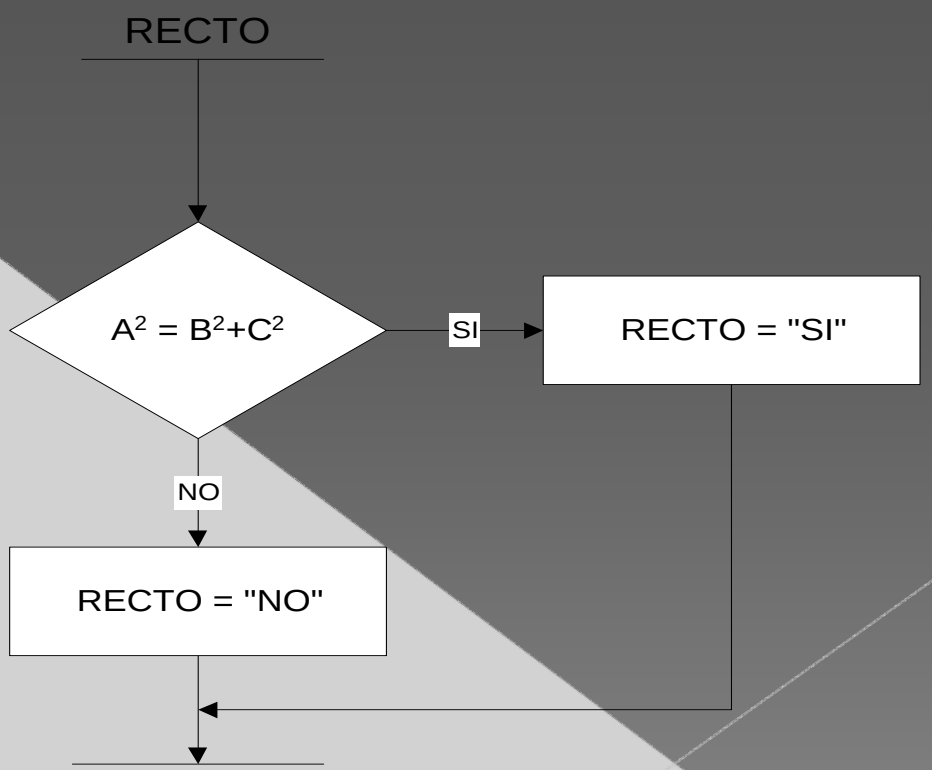


9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más *pequeño*

Recto: Se suprimen las condiciones $B^2 = A^2 + C^2$ y $C^2 = A^2 + B^2$





9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más **rápido**.

Aprovechando la utilización de la rutina de ordenación de los lados del ejemplo anterior (ORDENAR), a los efectos de introducir otros recursos de optimización que no hagan a la reducción del programa - lo más **pequeño** posible - , sino a la reducción de los tiempos de ejecución - lo más **rápido** posible - .

A continuación se desarrolla uno de los métodos de clasificación más conocido: *intercambio directo o burbuja*.

Recordemos que este algoritmo se basa en recorrer la lista (arreglo), de derecha a izquierda o de izquierda a derecha, examinando pares sucesivos de elementos adyacentes, permutándose los pares desordenados. Así el desarrollo del algoritmo tal cual se mostró en el tema correspondiente.

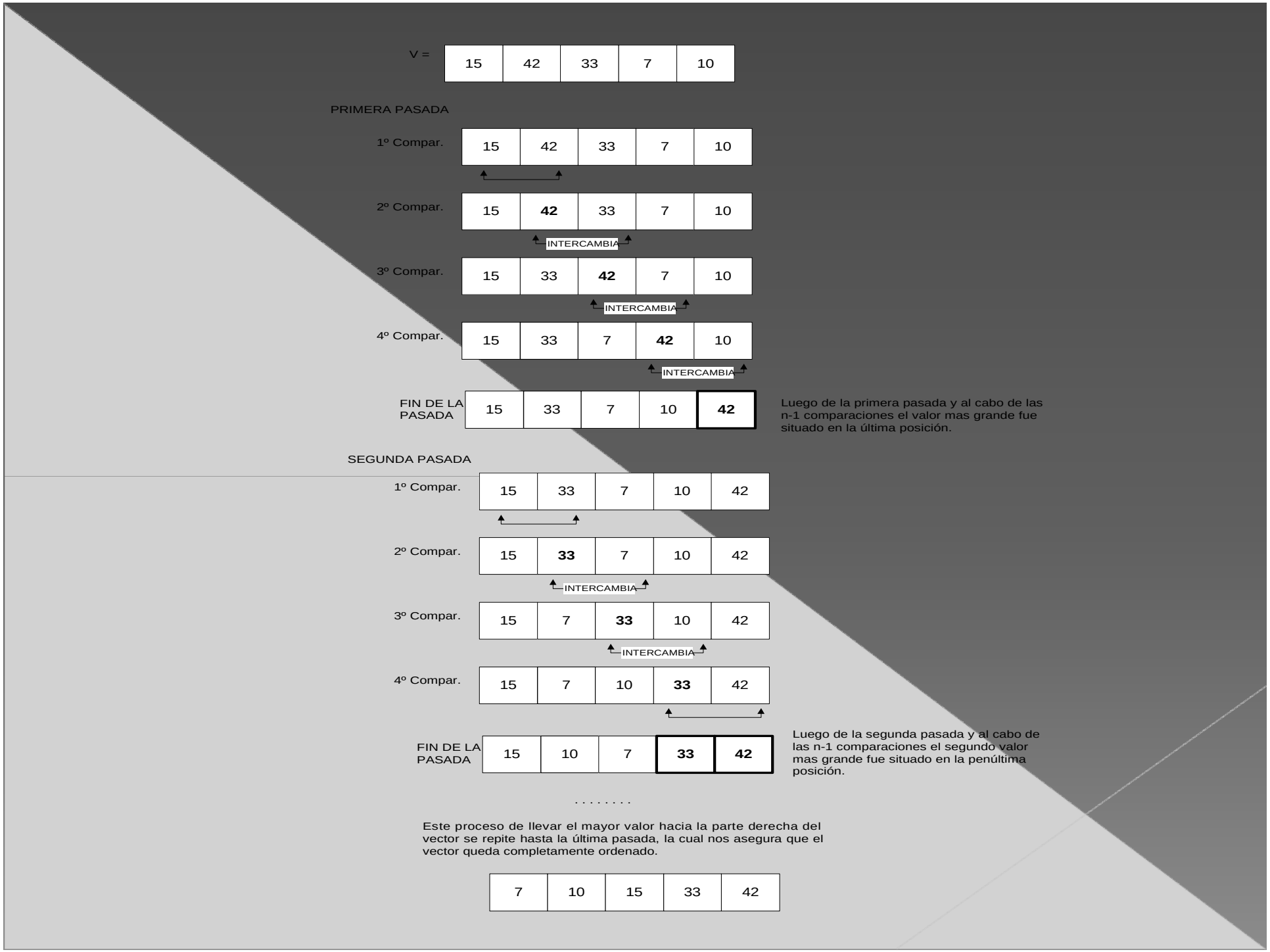
9.5 Formas de Optimizacion

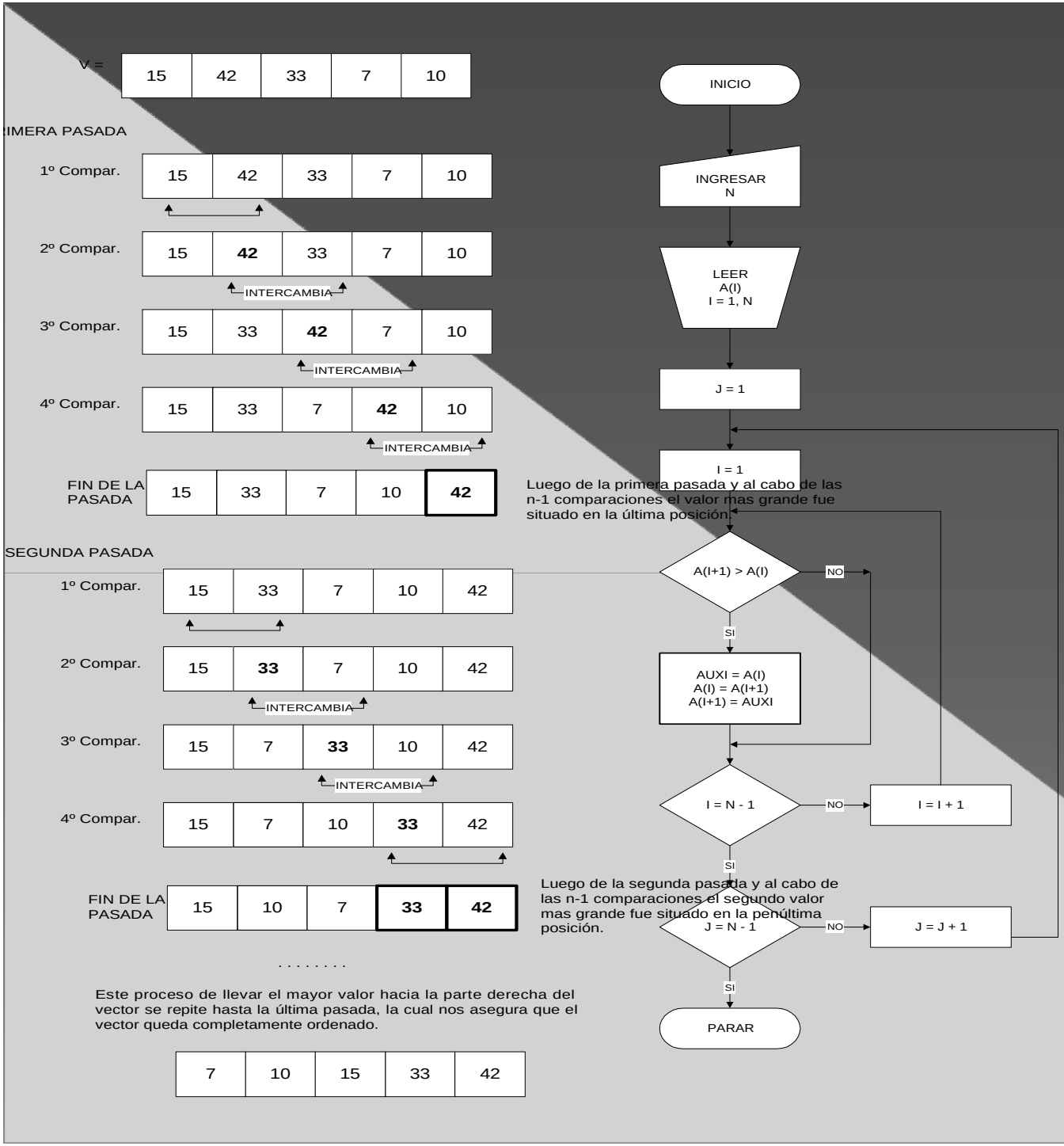
Optimización haciendo el algoritmo más rápido.

Aprovechando la utilización de la rutina de ordenación de los lados del ejemplo anterior (ORDENAR), a los efectos de introducir otros recursos de optimización que no hagan a la reducción del programa - lo más **pequeño** posible - , sino a la reducción de los tiempos de ejecución - lo más **rápido** posible - .

A continuación se desarrolla uno de los métodos de clasificación más conocido: *intercambio directo o burbuja*.

Recordemos que este algoritmo se basa en recorrer la lista (arreglo), de derecha a izquierda o de izquierda a derecha, examinando pares sucesivos de elementos adyacentes, permutándose los pares desordenados. Así el desarrollo del algoritmo tal cual se mostró en el tema correspondiente.

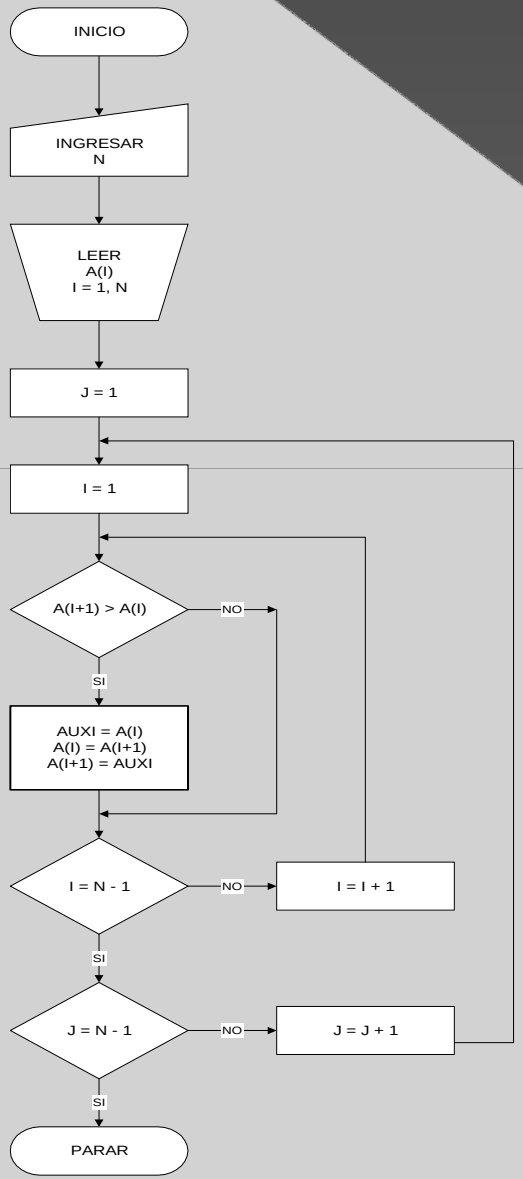

V = 15 42 33 7 10
PRIMERA PASADA
1º Compar. 15 42 33 7 10
2º Compar. 15 42 33 7 10
INTERCAMBIA
3º Compar. 15 33 42 7 10
INTERCAMBIA
4º Compar. 15 33 7 42 10
INTERCAMBIA
FIN DE LA PASADA 15 33 7 10 42
Luego de la primera pasada y al cabo de las n-1 comparaciones el valor mas grande fue situado en la última posición.
SEGUNDA PASADA
1º Compar. 15 33 7 10 42
2º Compar. 15 33 7 10 42
INTERCAMBIA
3º Compar. 15 7 33 10 42
INTERCAMBIA
4º Compar. 15 7 10 33 42
FIN DE LA PASADA 15 10 7 33 42
Luego de la segunda pasada y al cabo de las n-1 comparaciones el segundo valor mas grande fue situado en la penúltima posición.
. . . . . .
Este proceso de llevar el mayor valor hacia la parte derecha del vector se repite hasta la última pasada, la cual nos asegura que el vector queda completamente ordenado.
7 10 15 33 42

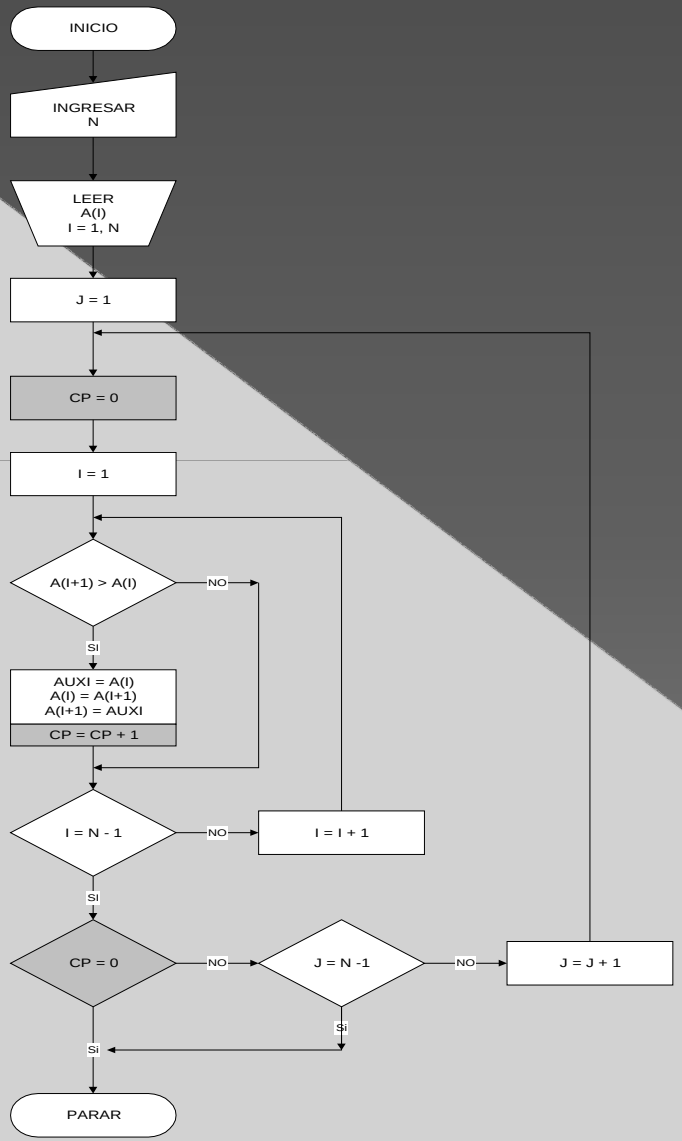


Al analizar el algoritmo se observa que se realiza en cada pasada *n-1 comparaciones* para analizar los *n-1* elementos adyacentes, pero además se realizan *n-1 pasadas* garantizando la ordenación total de la lista, realizando en total *(n-1) * (n-1) comparaciones* resultando el tiempo de ejecución del algoritmo función de n^2.

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.





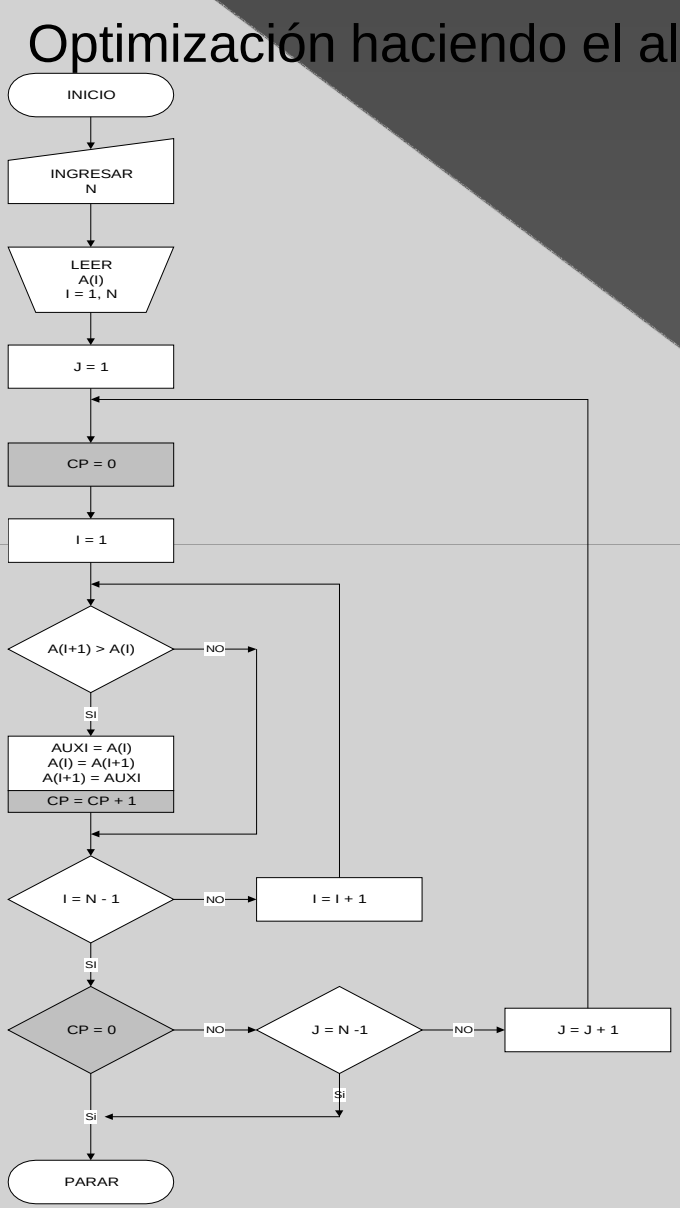
1˚ Optimización

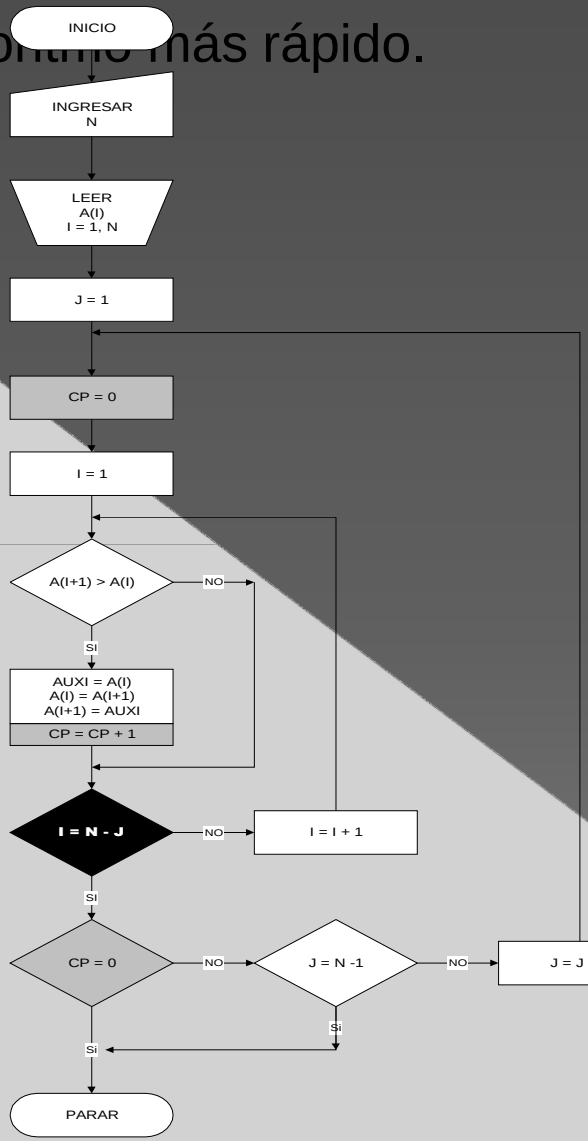
Como la lista pudo haber sido ordenada en alguna pasada intermedia, una primera optimización del método consiste en parar cuando se detecte una pasada nula (cuando no existe permutación), agregando al algoritmo un control de permutaciones (CP) quedando como se muestra en la figura

De esta manera cuando no existan permutaciones, o sea CP = 0, se detendrá el algoritmo realizando un número menor de comparaciones al calculado anteriormente

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.





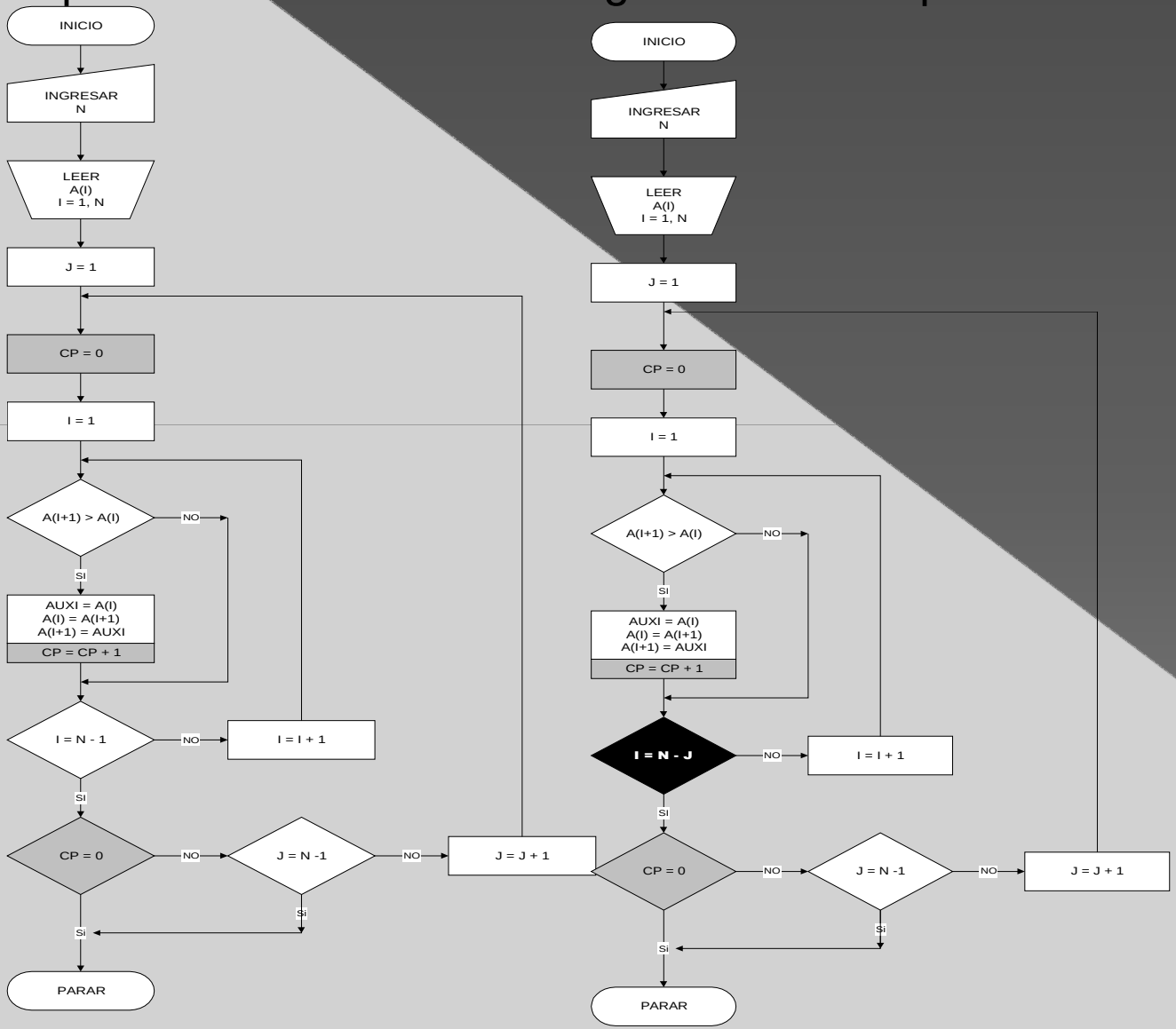
2° Optimización

Como en la primer pasada el algoritmo lleva el elemento más chico o más grande al final de la lista, en la siguiente pasada lleva el segundo más chico o más grande a la penúltima posición y a continuación hace una comparación ociosa, por lo que en esta segunda optimización lo que se pretende es que en las pasadas siguientes se reduzca en uno la cantidad de comparaciones, y así sucesivamente.

o se logra cambiando en el algoritmo el control de salida de la iteración más interna **N - 1 por N – J** quedando como se muestra en la figura

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.



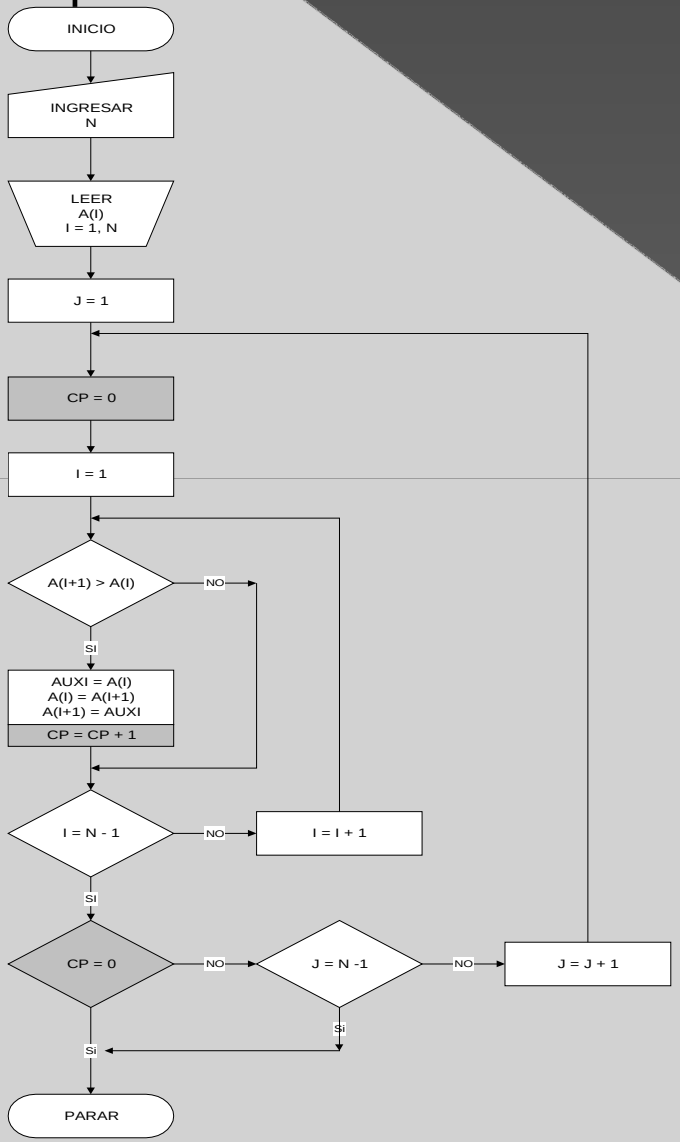
2° Optimización

En este caso el algoritmo, si no tuviera el control de permutaciones, haría en la primer pasada n-1 comparaciones, en la segunda n-2, y así sucesivamente hasta que en la n-1 pasada haría 1 sola comparación. Por lo tanto la suma de todos los términos de esta progresión aritmética sería:

$$C = ((n-1) + 1) * (n-1) / 2 = (n^2 - n) / 2$$

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.



3° Optimización

Buscar la forma de detener la ejecución del algoritmo cuando la lista ya está ordenada, en lugar de esperar hasta que se haga una pasada nula innecesaria, como ocurre en el caso de la 1° Optimización.

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.

4° Optimización

Dependiendo del grado de desorden de la lista es útil ejecutar pasadas alternativas de izquierda a derecha y de derecha a izquierda. Esta optimización se conoce como el *método de la sacudida*.

Por ejemplo si tuviéramos la siguiente lista: 1, 5, 4, 3, 2

Si hiciéramos pasadas de izquierda a derecha la lista se ordenaría en una sola pasada, en cambio si hiciéramos pasadas de derecha a izquierda serían necesarias cuatro pasadas.

Sin embargo si tuviéramos la siguiente lista: 4, 3, 2, 1, 5

Si se realizan pasadas de izquierda a derecha la lista se ordenaría en cuatro pasadas, en cambio si se realizan pasadas de derecha a izquierda sería necesaria una sola pasada.

9.5 Formas de Optimizacion

Optimización haciendo el algoritmo más rápido.

5° Optimización

Clasificación binaria (es utilizada por los programas productos comerciales: *SORT*).

Los métodos binarios tienen por principio común subdividir la lista a ordenar en dos o más sublistas, las que una vez ordenadas utilizando algoritmos simples de clasificación, intercalan las mismas (*MERGE*).

La razón es que es más rápido ordenar dos listas de (n/2) elementos e intercalarlas, que ordenar una lista de n elementos.

Ejemplo elemental: Ordenar una lista de diez(10) elementos requiere (n-1)*(n-1) comparaciones igual a 9*9 = 81 con el método de burbuja simple versus dos listas de cinco(5) elementos cada una (4*4)*2 = 32 comparaciones más las comparaciones que insume el proceso de intercalación (aproximadamente n=10 comparaciones) lo que nos daría un total de 42 comparaciones, sustancialmente menor que las 81 hechas con el método burbuja.

Conclusión:

Hemos visto los dos principios básicos de optimización:

- El primero lo más **pequeño** posible.
- El segundo lo más **rápido** posible.

En forma pura cada una de ellas.

En los problemas reales a veces se exige no hacerlo, o no hacerlo todavía. En caso de hacerlo puede presentarse prioritariamente uno de ellos (en función de las restricciones de tamaño o tiempo), o ambos a la vez , para lo cual habría que balancear las posibilidades de cada una.

9.6 Eficiencia de algoritmos

Definición de Eficiencia

Uso racional de los recursos con que se cuenta para alcanzar un objetivo predeterminado. A mayor eficiencia menor la cantidad de recursos que se emplearán, logrando mejor optimización y rendimiento.

El análisis de eficiencia de los algoritmos o métodos aplicados en los algoritmos es una tarea generalmente compleja y afecta directamente al tiempo de ejecución de los mismos.

El concepto de eficiencia

- Pensar en la optimización de un algoritmo requiere analizar previamente su eficiencia.
- La utilización que se hace desde el algoritmo de los recursos del sistema físico donde se ejecuta.
- Eficiencia se refiere al a forma de administración de todos los recursos disponibles en el sistema, de los cuales el tiempo de procesamiento es uno de ellos.

Definición

Un algoritmo es eficiente si realiza una administración correcta de los recursos del sistema en el cual se ejecuta

Análisis de la eficiencia de un algoritmo

- Tiempo de ejecución

Desde este punto se consideran mas eficientes aquellos algoritmos que cumplan con la especificación del problema en el menor tiempo posible. En este caso el recurso a optimizar es el tiempo de procesamiento. (Ej.: reserva de pasaje; monitoreo de señales en tiempo real, el control de alarmas, etc.).

- Uso de la memoria

Serán eficientes aquellos algoritmos que utilicen estructuras de datos adecuadas de manera de minimizar la memoria ocupada. Se pone énfasis en el volumen de la información a manejar en la memoria (Ej.: manejo de base de datos, procesamiento de imágenes en memoria, reconocimiento de patrones, etc.).

Análisis de algoritmos según tiempo de ejecución

- Para poder medir la eficiencia de un algoritmo, desde el punto de vista de su tiempo de ejecución, es fundamental contar con una medida del trabajo que realiza. Esta medida permitirá comparar los algoritmos y seleccionar, de todas las posibles, la mejor implementación.

- Básicamente en los algoritmos hay dos operaciones elementales: las comparaciones de valores y las asignaciones.

- Según el ordenador que se utilice se tendrá en cuenta el tiempo de cada clase de operaciones.

Análisis de algoritmos

Se considera la tarea de calcular el mínimo de tres números a, b y c y se analizan cuatros formas de resolverlo:

Método 1

```
m := a;
If b < m then m:= b;
If  c < m then m:= c;
```

Método 2

```
If a <= b
then If a <= c then m:= a
                else    m:= c
else If  b <= c then m:= b
                else    m:= c
```

Método 3

```
If (a<=b) and (a<=c) then m:= a;
If (b<=a) and (b<=c) then m:= b;
If (c<=a) and (c<=b) then m:= c;
```

Método 4

```
If (a<=b) and (a <=c)
              then m:=a
   else If  b <= c m then m:= b
         else m := c;
```

Análisis de Algoritmos

Método 1

```
m := a;
If b < m then m:= b;
If  c < m then m:= c;
```

Realiza dos comparaciones y, al menos, una asignación. Si las dos comparaciones son verdaderas, se realizan tres asignaciones en lugar de una

Análisis de Algoritmos

Método 2

```
If a <= b
then If a <= c then m:= a
                    else    m:= c
else If  b <= c then m:= b
                    else    m:= c
```

- Utiliza también dos comparaciones y una sola asignación.
- El trabajo de un algoritmo se mide por la cantidad de operaciones que realiza y no por la longitud del código.

Análisis de Algoritmos

Método 3

```
If (a<=b) and (a<=c) then m:= a;
If (b<=a) and (b<=c) then m:= b;
If (c<=a) and (c<=b) then m:= c;
```

Realiza seis comparaciones y una asignación, ya que aunque las primeras dos comparaciones den como resultado que a es el mínimo, las otras cuatro también se realizan.

Análisis de Algoritmos

Método 4

```
If (a<=b) and (a <=c)  then
    m:=a
    else If  b <= c m then
                m:= b
                else  m := c;
```

Requiere una sola asignación. Pero puede llegar a hacer tres omparaciones.

Análisis de Algoritmos

La diferencia entre 2, 3 o 6 comparaciones puede parecer poco importante.

Hagamos una modificación al problema:

Hallar el menor (alfabéticamente) de tres strings o cadenas en lugar del mínimo de tres números.

Donde cada uno de los sgtrings contenga 200 caracteres.

Suponemos que el lenguaje no provee un mecanismo para compararlos directamente, sino que debe hacerse letra por letra.

Análisis de Algoritmos

Se puede ver:

Las comparaciones se pueden convertir en 400, 600 o 1200 (incremento potencial = importante en el tiempo de ejecución).

Análisis de Algoritmos

Si en lugar de hallar el menor de 3 strings, se requiere hallar el mínimo de n strings, con n grande.

- El método 3 implica comparar cada numero con los restantes, lo cual lleva a realizar n(n-1) comparaciones, mientras que el método 1 solo compara el número buscado una vez con cada uno de los n elementos y haciendo de esta forma (n-1) comparaciones.
- El método 3 requiere de mucho mas tiempo que el método 1 ya que realiza n comparaciones por cada elemento.
- El método 1 aparece como el más fácil de implementar y generalizar.

Método 3

```
If (a<=b) and (a<=c) then m:= a;
If (b<=a) and (b<=c) then m:= b;
If (c<=a) and (c<=b) then m:= c;
```

Método 1

```
m := a;
If b < m then m:= b;
If  c < m then m:= c;
```

Análisis de Algoritmos

Si se identifica la unidad intrínseca de trabajo realizada por cada uno de los cuatro métodos, es posible determinar cuales son los métodos que realizan trabajo superfluo y cuales los que realizan el trabajo mínimo necesario para llevar a cabo la tarea.

Análisis de Algoritmos

El método 4 no es un buen método en particular, pero sirve para ilustrar un aspecto importante relacionado con el análisis de algoritmos.

Cada uno de los otros métodos presenta un número igual de comparaciones independientemente de los valores de a, b y c.

En el método 4, el número de comparaciones puede variar, dependiendo de los datos.

Esta diferencia, se describe diciendo que el método 4 puede realizar dos comparaciones en el mejor caso, y tres comparaciones en el peor caso.

Método 4

```
If (a<=b) and (a <=c) then
    m:=a
    else If  b <= c m then
                m:= b
                else m := c;
```

Análisis de Algoritmos

En general se tiene interés en el comportamiento del algoritmo en el peor caso o en el caso promedio. En la mayoría de las aplicaciones no es importante cuan rápido puede resolver el problema frente a los datos organizados favorablemente, sino ocurre frente a datos adversos (caso peor) o en el caso estadístico (caso promedio).

(Ej.: si se desea encontrar el mínimo de tres números distintos, hay seis casos diferentes correspondientes a los seis ordenes relativos en que pueden aparecer los tres números, en los dos casos donde "a" es el mínimo, el método 4 realiza dos comparaciones y en los otros cuatro casos, realiza tres comparaciones. De esto se deduce que si todos los casos son igualmente probables, el método 4 realiza 8/3 comparaciones en promedio).

Programas Eficientes

No repetir cálculos innecesarios

Ej.: puede escribirse:

```
a:= 2 * x * t;
y := 1/(a-1) + 1/(a-2) + 1/(a-3) + 1/(a-4)
```

en lugar de:

```
y:= 1/(2*x*t-1) + 1/(2*x*t-2) + 1/(2*x*t-3) + 1/(2*x*t-4)
```

Si esta expresión que se recalcula está dentro de un lazo repetitivo de 1000 veces, la ejecución reiterada de 2 * x * t, significa 4000 operaciones redundantes.

Programas Eficientes

Debe escribirse

```
t:=x*x*x;
y:= 0;
for n:= 1 to 2000 do
      y := y + 1/(t – n);
```

en lugar de:

```
for n:= 1 to 2000 do
      y := y + 1/(x*x*x – n);
```

Se evita calcular x*x*x dos mil veces.

Tiempo de ejecución

Análisis teórico: se busca obtener una medida del trabajo realizado por el algoritmo a fin de obtener una estimación teórica de su tiempo de ejecución. Básicamente se calcula el número de comparaciones y de asignaciones que requiere el algoritmo. Los análisis similares realizados sobre diferentes soluciones de un mismo problema permiten estimar cual es la solución más eficiente.

Análisis empírico: se basa en la aplicación de juegos de datos diferentes a una implementación del algoritmo, de manera de medir sus tiempos de respuestas. La aplicación de los mismos datos a distintas soluciones del mismo problema presupone la obtención de una herramienta de comparación entre ellos. El análisis empírico tiene la ventaja de ser muy fácil de implementar, pero no tiene en cuenta algunos factores como:

- La velocidad de la maquina, esto es, la ejecución del mismo algoritmo en computadores diferentes produce distintos resultados. (no es posible tener una medida de referencia).

- Los datos con los que se ejecuta el algoritmo, ya que los datos empleados pueden ser favorables a uno de los algoritmos, pero pueden no representar el caso general, con lo cual las conclusiones de la evaluación pueden ser erróneas.

Conclusión: es valiosos realizar un análisis teórico que permita estimar el orden de tiempo de respuesta, que sirva como comparación relativa entre las diferentes soluciones algorítmicas, sin depender de los datos de experimentación.

Tiempo de ejecución

Definición

El tiempo de ejecución T(n) de un algoritmo se dice de orden f(n) cuando existe una función matemática f(n) que acota a T(n).

T(n) = O(f(n)) si existen constantes “c” y “n0” tales que T(n) >= c f(n) cuando n>= n0.

Tiempo de ejecución

La definición anterior establece un orden relativo entre las funciones del tiempo de ejecución de los algoritmos T1(n) y T2(n).

Ej.

$T1(n) = O(2*n)$ $T2(n) = O(4*n*n)$

para n > 5, resulta que T1 es mas eficiente que T2

Tiempo de ejecución

En cuanto a la velocidad de crecimiento:

Ej.: T(n) = 1000 n con f(n) = n**2.

Para valores pequeños de n, 1000 n es mayor que n**2.

Pero n**2 crece mas rápido, con lo cual n**2 podría ser eventualmente la función mas grande. Esto ocurre p/n>= 1000

Tiempo de ejecución

Resumiendo

Al decir que T(n) = O(f(n)), se esta garantizando que la función T(n) no crece más rápido que f(n), es decir que f(n) es un limite superior (cota) para T(n).

Ej.: n**3 crece mas rápido que n**2, por lo tanto se puede afirmar que n**2 = O(n**3).

```
Function sum(n: integer):integer; var j, SumaParcial integer;
Begin
{1}       SumaParcial := 0;
{2}       for j := 1 to n do
{3}                 SumaParcial := SumaParcial + j*j*j;
{4}                 sum := SumaParcial;
End;
```

Las declaraciones no consumen tiempo.

Las líneas {1} y {4} implican una unidad de tiempo c/u.

La línea {3} consume 4 unidades de tiempo y se ejecuta n veces. Da un total de 4n unidades.

La línea {2} tiene un costo encubierto: inicializar (1); testear si j <=n (n+1) e incrementar j (n). Lo que nos da: 2n+2 (1+ n + 1 + n).

Si se ignora el costo de la invocación de la función y el retorno, el total es de: 6n + 4, es una función de O(n).

Reglas generales

Regla 1: Para lazos incondicionales.

El tiempo de ejecución de un lazo incondicional es, a lo sumo, el tiempo de ejecución de las sentencias que están dentro del lazo, incluyendo testeos, multipicada por cantidad de iteraciones que se realizan.

Regla 2: Para lazos incondicionales anidados.

Se debe realizar el análisis desde adentro hacia afuera. El tiempo total de un bloque dentro de lazos anidados es el tiempo de ejecución del bloque multiplicado por el producto de los tamaños de todos los lazos incondicionales.

Regla 3: If then else

Dado un fragmento de código de la forma:

```
If condición
      then S1
      else S2
```

El tiempo de ejecución no puede ser superior al tiempo del testeo mas el max (t1, t2) donde t1 es el tiempo de ejecución de S1 y t2 es el tiempo de ejecución de S2.

Regla 4: Para sentencias consecutivas.

Si un fragmento de código esta formado por dos bloques de uno con tiempo t1(n) y otro t2(n), el tiempo total es el máximo de los dos anteriores.

BIBLIOGRAFIA. General

oW. I. Salmon. Introducción a la computación con Turbo Pascal. Addison-Wesley Iberoamericana, 1993.
oJ. Castro, F. Cucker, F. Messeguer, A. Rubio, Ll. Solano, y B. Valles. Curso de programación. McGraw-Hill, 1993.

oSe ofrecen buenos enfoques de la programación con subprogramas. El primero de ellos introduce los subprogramas antes incluso que las instrucciones estructuradas. El segundo ofrece una concreción de los conceptos de programación modular explicados en los lenguajes C y Modula-2.

oS. Alagíc y M.A. Arbib. The design of well-structured and correct programs. Springer Verlag, 1978.
oEs una referencia obligada entre los libros orientados hacia la verificación con un enfoque formal.

oR. S. Pressman. Ingeniería del Software. Un enfoque práctico. McGraw-Hill, 2005.
oAlgunos de los conceptos contenidos en el tema provienen de la ingeniería del software.

oFundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642.

oALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2

oFUNDAMENTOS DE PROGRAMACIÓN. Libro de Problemas en Pascal y Turbo Pascal; Luis Joyanes Aguilar Luis Rodríguez Baena y Matilde Fernandez Azuela; 1999; Editorial: MCGRAW-HILL. ISBN: 844110900.

oPROGRAMACIÓN; Castor F. Herrmann,María E. Valesani.; 2001; Editorial: MOGLIA S.R.L..ISBN: 9874338326.