Tema VIII: Eficiencia y Optimización

Análisis de algoritmos: concepto de eficiencia. Principio de invarianza. Tipo de Análisis. Caso mejor, peor o promedio. Cálculo de la Eficiencia. Principio de Optimalidad. Formas de Optimización. Técnicas de aplicación de estrategias algorítmicas seguras. Otras estrategias algorítmicas. Herramientas de optimización.

8.1 Análisis de algoritmo

En la actualidad a causa de la revolución tecnológica y la digitalización, cada vez son más las capacidades y habilidades que demandan las empresas.
Entre ellas, podemos encontrar “habilidades de poder” como la eficiencia, cuya importancia va en aumento.
No obstante, la eficiencia es un concepto que puede generar confusión con la eficacia. Y es que, si bien podemos decir que una persona eficiente también es eficaz, no podemos afirmar que una persona eficaz sea también eficiente.
Dado que entender en qué consisten estas habilidades resulta fundamental para poder adquirirlas, llevarlas a la práctica en entornos laborales y poder demostrar su aplicación en el día a día, a continuación, te explicamos sus significados, sus características y la importancia de estas competencias a la hora de acceder y prosperar en el mercado laboral.

Para introducir el tema de la diferencia entre ambos términos se presenta la siguiente tabla (Tab. 8.01) donde se plantean las dos siguientes preguntas:

a) Como podemos hacer mejor lo que hacemos, y
b) Que es lo que deberíamos estar haciendo.

EFICIENCIA	EFICACIA
Enfasis en los medios	Enfasis en los resultados
Hacer las cosas correctamente	Hacer las cosas correctas
Resolver problemas	Lograr objetivos
Ahorrar gastos	Aumentar creación de valores
Cumplir tareas y obligaciones	Obtener resultados
Capacitar a los subordinados	Proporcionar eficacia a subordinados
Asistir al templo (Iglesia)	Practicar los valores religiosos
Enfoque reactivo	Enfoque proactivo
Del pasado al presente	Del futuro al presente
Pregunta principal que se hace en cada una de ellas	
¿Cómo podemos hacer mejor lo que hacemos?	¿Qué es lo que deberíamos estar haciendo?

Tab. 8.01 Relación Eficiencia y Eficacia

¿Qué es exactamente la eficacia? ¿Y la eficiencia?

La eficacia (según el Diccionario de la lengua española - DLE) se define como “la capacidad de lograr el efecto que se desea o se espera”. Es decir, es un concepto que está relacionado con el resultado que se obtiene de un proceso.
En este sentido, si hacemos una aproximación de su significado en entornos profesionales, podríamos calificar como eficaz a aquella persona que es capaz de alcanzar unas metas u objetivos (autoimpuestos o marcados por la empresa), en un tiempo determinado.
Sin embargo, es importante conocer las diferencias entre eficaz y eficiente para no confundir ambos términos, ya que no son lo mismo.
La eficiencia (DLE) es tanto “la capacidad de disponer de alguien o de algo para conseguir un efecto determinado” como “la capacidad de lograr los resultados deseados con el mínimo posible de recursos”. Es decir, se refiere a los medios que se disponen para desarrollar un proceso.
Por tanto, eficaz es “aquella persona que sirve para lo que se espera de ella”, mientras que la eficiente es “una persona competente, la que rinde en su actividad”.
Por tanto, la eficacia es la capacidad de conseguir lo que se propone en el tiempo indicado, pero la eficiencia es lograr el objetivo con menos recursos, lo que implica que el gasto temporal es el mismo, pero se reducen los costes de otros recursos.

Ej.: *En un proceso, dos programadores tienen que desarrollar un código para la solución de un problema en un tiempo determinado, y ambos lo consiguen. Sin embargo, uno de ellos ha utilizado estructuras de datos más adecuadas y estrategias algorítmicas que además de reducir la cantidad de instrucciones, mejoran sustancialmente el tiempo de ejecución, es decir. las herramientas necesarias y las ha gestionado de manera adecuada, por lo que ha reducido su consumo en el proceso. En cambio, el otro programador ha mantenido la estructura de los datos básicos y no ha mejorado las estrategias algorítmicas.*

A partir de este ejemplo, podemos establecer que ambos son eficaces porque han terminado la tarea a tiempo, pero, mientras que el primer programador ha logrado un código optimo y eficiente, el segundo presento una solución que funciona, pero no mejorar el uso de recursos.

En la figura de abajo (Fig. 8.01) se refleja la relación entre eficiencia y eficaz en el contexto de la utilización de los recursos.

Fig. 8.01 Relación Eficaz – Eficiente

Entonces …

- 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.
- Y a la forma de administración de todos los recursos disponibles en el sistema, de los cuales el tiempo de procesamiento es uno de ellos.

8.2 Origines del análisis de eficiencia de algoritmos

Antes de incursionar sobre los orígenes del análisis de eficiencia de los algoritmos es necesario reconocer el crecimiento de la complejidad de un algoritmo, para ello es se podría partir de esta antigua leyenda sobre el creador del juego del ajedrez.

La leyenda … El Sissa ben Dahir invento el juego de ajedrez para el rey Shirham de la India. El rey ofreció a Sissa la posibilidad de elegir su recompensa. Entonces Sissa le dijo al rey que podía recompensarlo con “trigo” de una de las siguientes alternativas:

a) Una cantidad equivalente a la cosecha de trigo dos años
b) Una cantidad de trigo que se calcularía de la siguiente forma:
    a. un grano de trigo en la primera casilla de un tablero de ajedrez,
    b. más dos granos de trigo en la segunda casilla,
    c. mas cuatro granos de trigo la tercera casilla,
    d. y así sucesivamente, duplicando el numero de granos de cada casilla, hasta llegar a la última casilla (64).

El rey pensó que la primera opción era muy cara y accedió a la segunda opción, eran simplemente granos de trigo por cada casilla.

En la figura (Fig. 8.02) siguiente se refleja la equivocación del rey por no analizar adecuadamente el crecimiento (grado de complejidad) que tendría el proceso de recompensa.

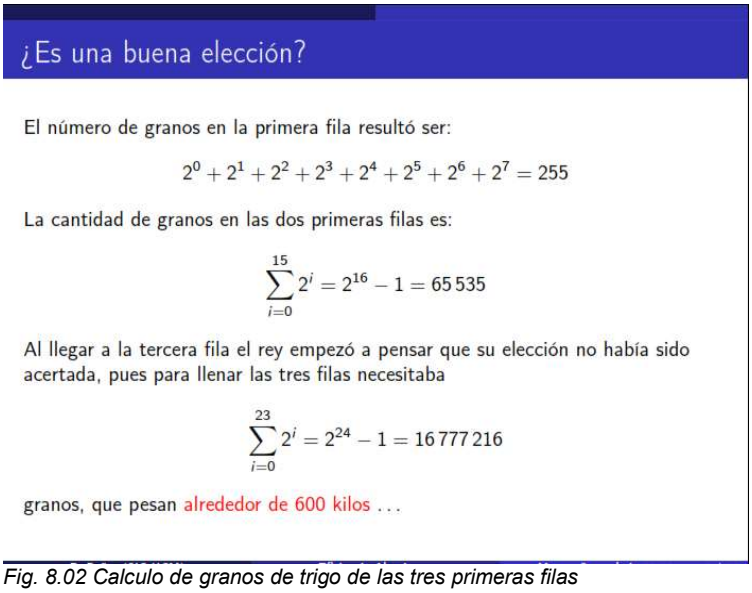


Fig. 8.02 Calculo de granos de trigo de las tres primeras filas

En efecto para rellenar las 64 casillas hace falta:

$\sum_{i=0}^{63} 2^i = 2^{64} - 1 = 18446744073709551615 = = 1{,}84 * 10^{19}$

Cantidad equivalente a miles cosechas mundiales.

La función $2^n - 1$ (exponencial) representa el número de granos adeudados en función del numero de *"n"* casillas a rellenar. Toma valores desmesurados, aunque el numero de casillas sea pequeño.
El coste en tiempo de algunos logarítmicos expresados en función del tamaño de los datos de entrada es también exponencial. Por ello es importante estudiar el coste de los algoritmos y ser capaces de comparar los costes de algoritmos que resuelven el mismo problema.

La técnica que se utilizaba en los primeros años de programación para comparar la eficiencia de distintos algoritmos, consistía en ejecutarlos para datos diferentes y medir el tiempo consumido.
Dado que los ordenadores y los lenguajes eran dispares, y que el tiempo de ejecución depende no solo del tamaño sino también del contenido de los datos, resultaba muy difícil compara tales resultados.
El primer estudio sobre eficiencia de los algoritmos fue el realizado por Daniel Goldenberg (1952). Realizo un análisis matemático del numero de comparaciones necesarias, en el mejor y en el peor caso, de cinco algoritmos distinto de ordenación.

Tiempo después (1956) en sus tesis doctoral Howard B. Demuth (Universidad de Stanford) estableció las bases de lo que hoy llamamos *análisis de la complejidad de los programas*.

Principios esenciales del análisis de complejidad

Lo primero que se decide es si se desea un estudio del caso mejor, del caso peor o del caso promedio del algoritmo. El análisis mas frecuente y el mas sencillo es el del caso peor.

Después se decide cual es el parámetro "n" que va a medir el tamaño del problema. Para cada problema habrá que determinar que significa dicha "n".
Finalmente, se estudia la función de dependencia *f(n)* entre el tamaño *"n"* del problema y del numero de pasos elementales del algoritmo.
El objetivo es clasificar dicha función en una determinada clase de complejidad. Para ello, se ignorar las constantes multiplicativas y aditivas y se desprecian los sumandos de menor orden.
La consecuencia es que funciones tales como $f(n) = 10n^2 + 32$ y $g(n) = 1/2n^2 - 123n - 15$ se clasifiquen en la misma clase que $h(n) = n^2$. Dado que n^2 es la función mas sencilla de dicha clase, la propia clase se denota $O(n^2)$, expresión que ha de entenderse como el conjunto de funciones del orden de n^2. La figura (Fig. 8.03) siguiente de Medidas Asintóticas se resume lo especificado anteriormente.

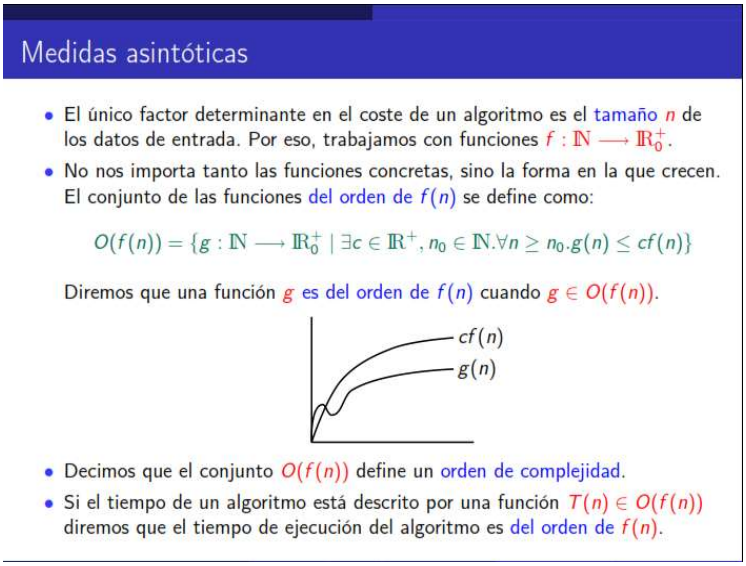


Fig. 8.03 Medidas asintóticas

Las dos figuras que se presentan refieren a la Jerarquía de órdenes de complejidad (Fig. 8.04) y Orden de complejidad en escala semilogarítmica (Fig. 8.05).

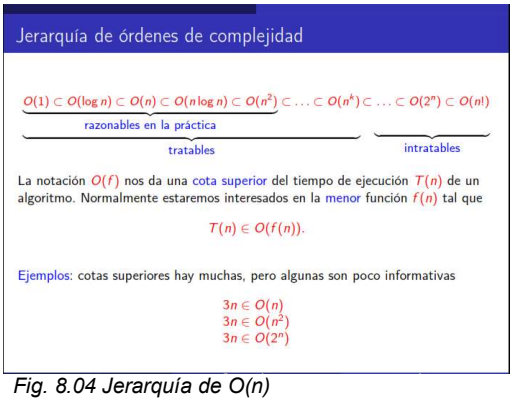


Fig. 8.04 Jerarquía de O(n)

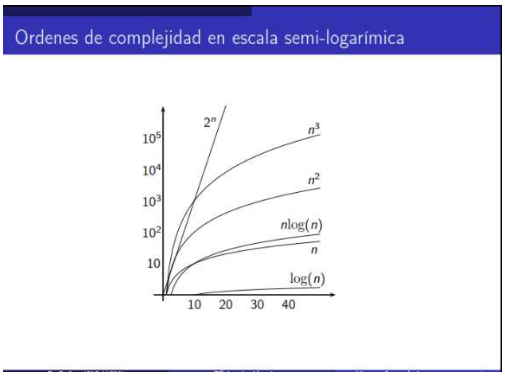


Fig. 8.05 O(n) en escala semi logarítmica

Es un error pensar que basta esperar algunos años para que algoritmos tan costosos se puedan ejecutar con un coste en tiempo razonable.
Para ello y sabiendo que el tiempo estimado transcurrido desde el Big Bang es de $1{,}4 * 10^8$ siglos, la siguiente tabla (Tab. 8.01) ilustra la importancia de que el coste del algoritmo sea pequeño.
Nomenclatura a considerar: ms = milisegundos, s = segundos, m = minutos, h = horas, d = días, a = años, sig = siglos.

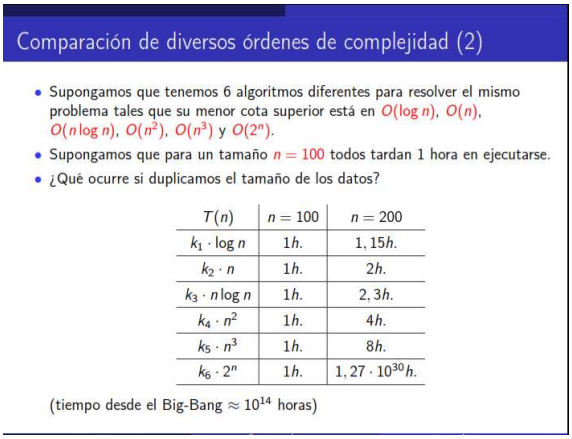
Comparación de diversos órdenes de complejidad

- Sabiendo que el tiempo transcurrido desde el Big-Bang es de $1,4 \times 10^8$ siglos, la siguiente tabla ilustra la importancia de que el coste del algoritmo sea pequeño (ms = milisegundos, s = segundos, m = minutos, h = horas, d = dias, a = años, sig = siglos).

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 *ms*	10 *ms*	10 *ms*	0,1 *s*	1 *s*	1,02 *s*
10^2	2 *ms*	0,1 *s*	0,2 *s*	10 *s*	16,67 *m*	$4{,}02 * 10^{20}$ *sig*
10^3	3 *ms*	1 *s*	3 *s*	16,67 *m*	11,57 *d*	$3{,}4 * 10^{291}$ *sig*
10^4	4 *ms*	10 *s*	40 *s*	1,16 *d*	31,71 *a*	$6{,}3 * 10^{3000}$ *sig*
10^5	5 *ms*	1,67 *m*	8,33 *m*	115,74 *d*	317,1 *sig*	$3{,}16 * 10^{30093}$ *sig*
10^6	6 *ms*	16,67 *m*	1,67 *h*	31,71 *a*	317 097,9 *sig*	$3{,}1 * 10^{301020}$ *sig*

- Es un error pensar que basta esperar algunos años para que algoritmos tan costosos se puedan ejecutar con un coste en tiempo razonable.

Las dos figuras siguientes comparan diversos ordenes de complejidad.
En el primer cuadro n = 100 y todos las funciones de ordenes de complejidad de seis algoritmos tardan 1 h en ejecutarse. Que ocurriría si se duplican los datos?.
En el segundo cuadro n = 100 y todos las funciones de ordenes de complejidad de seis algoritmos, en 1 h tratan los 100 elementos. Que ocurriría si se duplica la velocidad del ordenador, es decir, si duplicamos el tiempo disponible?.



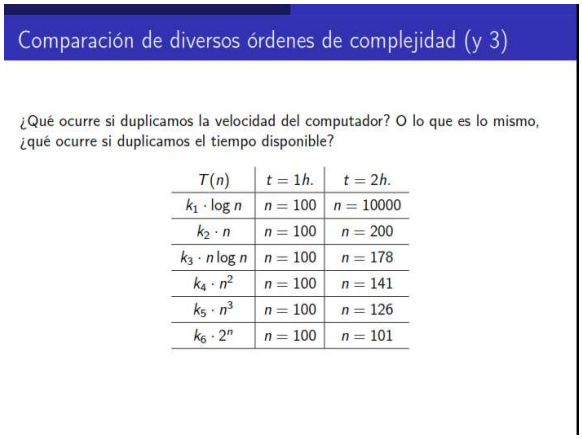
Comparación de diversos órdenes de complejidad (2)

- Supongamos que tenemos 6 algoritmos diferentes para resolver el mismo problema tales que su menor cota superior está en $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ y $O(2^n)$.
- Supongamos que para un tamaño $n = 100$ todos tardan 1 hora en ejecutarse.
- ¿Qué ocurre si duplicamos el tamaño de los datos?

$T(n)$	$n = 100$	$n = 200$
$k_1 \cdot \log n$	$1h.$	$1,15h.$
$k_2 \cdot n$	$1h.$	$2h.$
$k_3 \cdot n \log n$	$1h.$	$2,3h.$
$k_4 \cdot n^2$	$1h.$	$4h.$
$k_5 \cdot n^3$	$1h.$	$8h.$
$k_6 \cdot 2^n$	$1h.$	$1,27 \cdot 10^{30}h.$

(tiempo desde el Big-Bang $\approx 10^{14}$ horas)

Fig. 8.07 1er Cuadro comparación O(n)



Comparación de diversos órdenes de complejidad (y 3)

¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$T(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

Fig. 8.08 2do Cuadro compracion O(n)

8.03 Principio de invarianza

Dos implementaciones de un mismo algoritmo no diferirán mas que en una constante multiplicativa.

Si $t_1(n)$ y $t_2(n)$ son los tiempos de dos implementaciones de un mismo algoritmo,

se puede comprobar que: $\exists\, c, d \in R,\ t_1(n) \leq c\, t_2(n);\ t_2(n) \leq d\, t_1(n)$.

8. 04 Algoritmia

Para iniciar presentamos una definición de algoritmia:

Algoritmia o Algorítmica es el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes. (Brassard y Bratley – 1988; 1997).

Para comparar dos algoritmos que resuelven el mismo problema basta calcular sus respectivas clases de complejidad: si coinciden, diremos que los algoritmos son igualmente buenos; en caso contrario, será mejor el que pertenezca a la clase mas baja de la jerarquía.
La algoritmia se ocupa fundamentalmente de encontrar algoritmos polinomiales para los problemas computables. Dentro de los polinomiales, las clases más codiciadas son, por ese orden, O (1), O (log n), O(n) y O (n log n).
La primera no es posible para la gran mayoría de los problemas porque, dada su naturaleza (ej.: búsqueda, recorridos, etc.), el tiempo crece necesariamente con el numero *"n"* de objetos considerados.
La segunda incluye algoritmos excepcionalmente buenos (ej.: búsquedas en estructuras de datos) cuyo tiempo crece muy despacio.

Las otras dos se comportan como un usuario no informatico intuitivamente espera de los ordenadores: ej. Si el tamaño del fichero a comprimir se hace doble o triple, espera que el tiempo de comprensión se duplique o triplique.
En la practica, son muy infrecuente los algoritmos polimoniales con complejidad superior a O(n^5).

A continuacion se presenta la figura (Fig. 8.09) que refiere a los ordenes de complejidad y sus graficas correspondiente.

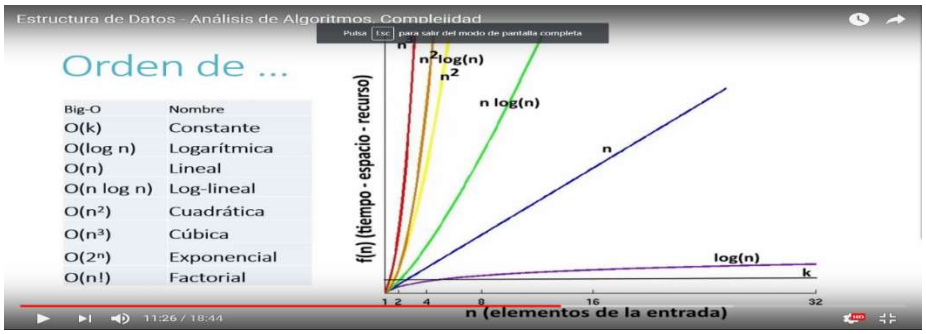


Fig. 8.09 Ordenes de Complejidad y graficas correspondientes

Crecimiento exponencial o factorial

Imaginemos un micropocesador que puede realizar 1000 instrucciones por nanosegundo, lo que implica que en 1 segundo ejecute 10^{12} instrucciones.

Para comprender acabadamente el crecimiento exponencial presentamos la siguiente tabla (Taba 8.02) donde se presenta dos funciones de complejidad. Una del O(2^n) y la otra de O(n!) y ambas tratan la ejecución de n elementos que toma los valores de 50, 60 y 100.

N	O(2^n)	O(n!)	ejecucion
50	1,13E+15	3,04E+64	O(2^n) => 19 minutos
60	1,15E+18	8,32E+81	O(2^n) => 19.215 minutos ~ 13 dias
100	1,27E30	9,33E+157	O(2^n) => 40.196.936.841 años

Tabla 8.02 Crecimiento exponencial

8.05 Eficiencia y Exactitud

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características:

- Que sea fácil de entender, codificar y depurar
- Que consiga la mayor eficiencia a los recursos de la computadora

Entonces la “Eficiencia” de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo mas corto posible y/o utilizando la cantidad mas pequeña posible de recursos físicos, y que sea

compatible con su exactitud y corrección y que sea desarrollado bajo normas de calidad. Es decir, la eficiencia como factor “espacio – tiempo” debe estar estrechamente relacionada con la buena calidad, el funcionamiento optimo y la facilidad de mantenimiento del programa.

Para lo cual consideramos

- 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.).
Concluimos que 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 (u operaciones) de valores y las asignaciones.
Según el ordenador que se utilice se tendrá en cuenta el tiempo de cada clase de operaciones.

A continuación desarrolamos algunos ejemplos para conocer y comprender de los que estamos hablando.

Primer Ejemplo: Se quiere saber si Un elemento cualquiera esta o no en un arreglo. Los elementos en el arreglo no estan ordenados, lo que implica realizar una busqueda secuencial.

```
bool buscar (int datos []]), int n , int buscado) {
        for (int i = 0; i < n; i++) {
            if (datos[i] == buscado) {
                return true;
                                    }
                                }
      return false;
    }
```

En el peor de los casos, en el presente código, la sentencia “if” se ejecutara n veces

Segundo Ejemplo: Se desea desarrollar un código para determinar si cada elemento del arreglo es único.

```
bool unico (int datos []]), int n )
                    {
        for (int i = 0; i < n; i++)
                                {
          for (int j = i + 1; j < n ; j++ )
              {
                  if (datos[i] == datos[j])
                  {
            return false;
           }
             }
      }
     return true;
  }
```

En cada iteración del ciclo externo (indice “i”), el ciclo interno (indice “J”) se ejecuta una cantidad de veces.
La primera vez el ciclo interno se ejecuta n-1 veces, la segunda vez se ejecuta n-2, y así sucesivamente, hasta 1 vez.
En el peor de los casos la sentencia “if” se va a ejecutar: n-1 + n-2 + n-3 + 3 + 2 + 1 veces.
Utilizando la formula de Gauss para n-1 números, se ejecutara **[n(n-1)] / 2** veces

Tercer Ejemplo: Tener en cuenta el incremento o paso del ciclo (for).

```
for (int i = 1; i < n; i = i * 2)
  {
          // hacemos algo .. cualquier cosa
  }
```

El indice “i” va a tomar los valores 1, 2, 4, 8, 16, … 2K.
Esto va a ocurrir hasta que 2k sea igual o mayor que “n”, es decir, el ciclo se ejecuta 2k-1 <= n
Aplicando matemática …

$$\log (2k-1) \leq \log (n)$$
$$(k-1) \log (2) \leq \log (n)$$
$$K - 1 \leq \log(n) / \log (2)$$

Es decir, lo que tengamos que hacer dentro del FOR se va a ejecutar :

K <= log 2 (n) + 1 veces

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

Método 1	**Método 2**

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

Método 3	**Método 4**

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

En el metodo 1 se realizan dos comparaciones y, al menos, una asignación. Si las dos comparaciones son verdaderas, se realizan tres asignaciones en lugar de una.

En el método 2 se utilizan también dos comparaciones y una sola asignación.
Tengamos presente que el trabajo de un algoritmo se mide por la cantidad de operaciones que realiza y no por la longitud del código.

En el método 3 se 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.
En el método 4 se requiere una sola asignación. Pero puede llegar a hacer tres comparaciones.

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.
Se puede ver que las comparaciones se pueden convertir en 400, 600 o 1200 (incremento potencial = importante en el tiempo de ejecución).
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.

Concluimos que el método 1 aparece como el más fácil de implementar y generalizar.
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.
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 (1, 2 y 3) presentan 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.

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).

Quinto Ejemplo: No repetir cálculos innecesarios.

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)
```

Ambos código producen el mismo efecto. La primera forma de escribir es mas optima debido a que la doble multiplicación la realiza una sola ves y la guarda en una variable. Luego utiliza esta variable para calcular el valor de la variable “Y”.
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.

Algo similar sucede con la escritura dl siguiente código:

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

Que puede escribirse de la siguiente manera:

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

La maera correcta y mas optima es la primera, ya que deje por fuera del FOR el doble producto.

8. 06 Eficiencia de bucles

En general el formato de eficiencia se puede expresar mediante una funcion:

F(n) = eficiencia

Es decir la eficiencia del algoritmo se examina como una funcion del numero de elementos a ser procesado.

8.05.01 Bucles lineales

En los bucles lineales se repiten las sentencias del cuerpo del bucle un numero determinado de veces m, que determina la eficiencia del mismo. Normalmente en los algoritmos los bucles son el termino dominante en cuanto a la eficiencia del mismo.

```
I := 1
mientras  (I <= n)
        codigo de la aplicación
I := I + 1
Fin – mientras
```

El numero de iteraciones es directamente proporcional al factor del bucle, n. Como la eficiencia es directamente proporcional al numero de iteraciones la funcion que expresa la eficiencia es:

F(n) = n

Analicemos cuantas veces se repite el cuerpo del bucle en el siguiente codigo?

```
I := 1
mientras  (I <= n)
```

```
        codigo de la aplicación
        I := I + 2
fin – mientras
```

Como dijimos y se observa que es directamente proporcinal al numero de n y eneste caso particular el incremento del subíndice se realiza de a dos (2), el cuerpo del mientras se realizara **F(n) = n / 2.**

Bucle multiplicar	Bucle de dividir
I := 1 Mientras (I < 1000) codigo de la aplicación I := I * 2 Fin-mientras	I := 1000 Mientras (I >= 1) codigo de la aplicación I := I / 2 Fin-mientras

Podriamos preguntarnos cual es el numero de iteración en cada una de las funciones?. Y aun mas, cual seria el grado de complejidad del algoritmo?.
En la tabla siguiente (Tabla 8.03) se obseva que el crecimiento en ambas funciones es el mismo. Minetras en una se incrementa de a dos (2) en la otra función se decrementa en mitades.

Bucle de multiplicar		Bucle de dividir	
Iteracion	Valor de I	Iteracion	Valor de I
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
salida	1024	Salida	0

Tabla 8.03. Comparacion de iteraciones de dos bucles

En ambos casos se ejecutan 10 iteraciones. La razon es que en cada iteracion el valor de I se dobla en el bucle de multiplicar y se divide a la mitad en el bucle de

dividir. Por consiguiente, el numero de iteraciones es una funcion del multipicador o divisor, en este caso: 2.

- Bucle de multiplicador 2 ** iteraciones < 1000
- Bucle de division 1000 / 2 ** iteraciones >= 1

Generalizando: **F(n) = [log2 n]**

8.05.02 Bucles anidados

Basicamente el total de iteraciones resulta de:

Iteraciones = iteraciones del bucle externo * bucle interno.
La tabla (tabla 8.04) siguiente refiere al tipo de complejidad en los diferentes casos de bucles anidados.

Formulas de eficiencia en bucles anidados	
LINEAL LOGARTIMICA	F(n) = [n * log2 n]
DEPENDIENTE CUADRATICA	F(n) = n *((n + 1) / 2)
CUADRATICA	F(n) = n ** 2

Tabla 8.04. Funciones de eficiencia en bucles anidados

8.07 Tiempo de Ejecucion

Para determinar el tiempode ejecución podemos deducir dos formas.

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.

Entonces definimos al tiempo de ejecución

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

Ej. Sean T1(n) = O(2*n); y T2(n) = O(4*n*n), se observa que para n > 5, resulta que T1 es mas eficiente que T2.

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.

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).

Costo encubierto de la sentencia FOR

Analizamos el código de la presente tabla.

```
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, que resulta de ➔ (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).

En detalle ….

Linea 1	…….	1
Linea 2	…….	2n + 2
Linea 3	…….	4n
Linea 4	…….	1
Ttotal		6n + 4

8.08 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 codigo 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.

Regla del calculo de eficiencia

1.- Sentencias Simples

Se considera que cualquier sentencia simple (lectura, escritura, asignación, etc.) va a consumir un tiempo constante O(1). Salvo que contenga un llamado a una función.

2.- Bloques de sentencias

Timepo de ejecución = Suma de los tiempos de ejecución de cada una de las sentencias del bloque.
Orden de eficiencia = Maximo de los ordenes de eficiencia de cada una de las sentencias del bloque.

3.- Sentencias condicionales

El tiempo de ejecución de una sentencia condicional es el máximo del tiempo del bloque *if* y del tiempo del bloque *else.*
Si el bloque *if* es O(f(n)) y el bloque *else* es O(g(n)), la sentencia condicional será **O(max{f(n), g(n)})**

4.- Bucles

Tiempo de ejecución = suma de los tiempos invertidos en cada iteración.
En el tiempo de cada iteración se ha de incluir el tiempo de ejecución del cuerpo del bucle y también el asociado a la evaluación de la condición (y, en su caso, la actualización del contador).

Si todas las iteraciones son iguales, el tiempo total de ejecución del bucle será el producto del numero de iteraciones por el tiempo empleado en cada iteración.

5.- Llamadas de funciones

Si una determinada función P tiene una eficiencia de O(f(n)), cualquier llamada a P es de orden O(f(n)).
Las asignaciones con diversas llamadas a función deben sumar las cotas de tiempo de ejecución de cada llamada.
La misma consideración es aplicable a las condiciones de bucles y sentencias condicionales.

6.- Funciones recursivas

Las funciones de tiempo asociadas a funciones recursivas son también recursivas.

Ej.: $T(n) = T(n - 1) + f(n)$

Para analizar el tiempo de ejecución de un algoritmo recursivo, le asociamos una función de eficiencia desconocida T(n), y la estimamos a partir de T(k) para distintos valores de k (menores que n).

Ejemplo

```
int fact(int n)
{
  if (n <= 1)
     return 1;
  else
     return (n * fact(n - 1));
 }
```

Los bloques `if` y `else` son operaciones elementales, por lo que su tiempo de ejecución es O(1).

$T(n) = 1 + T(n-1)$

$= 1 + (1 + T(n-2)) = 2 + T(n-2)$

$= 2 + (1 + T(n-3)) = 3 + T(n-3)$

...

$= i + T(n-i)$

...

$= (n-1) + T(n-(n-1)) = (n-1) + 1$

Por tanto, T(n) es O(n)

La implementación recursiva del cálculo del factorial es de orden lineal.

Ejemplo

```
int E(int n)
{
   if (n == 1)
      return 0;
   else
      return E(n/2) + 1;
}
```

$$T(n) = \begin{cases} 1, & n = 1 \\ 1 + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

T(n) es O ($\log_2 n$)

$T(n) = T(n/2) + 1$

$= (T(n/4) + 1) + 1 = T(n/4) + 2$

$= (T(n/8) + 1) + 2 = T(n/8) + 3$

...

$= T(n/2^i) + i$

...

$= T(n/2^{\log(n)}) + \log_2(n)$

$= T(1) + \log_2(n)$

Por tanto, T(n) es O(log n)

8.09 Principio de Optimalidad

Optimización hace referencia a la acción y efecto de optimizar. En términos generales, se refiere a la capacidad de hacer o resolver alguna cosa de la manera más eficiente posible y, en el mejor de los casos, utilizando la menor cantidad de recursos.
La optimización de algoritmos hace referencia al análisis cuidadoso de su desempeño para analizar las fallas y concebir mejoras antes de llevarlos al computador. Lo usual es que un primer algoritmo que se nos ocurra puede mejorarse muchísimo.
Optimizar un algoritmo implica construirlo lo mas correctamente posible, con estilo, transparente, sin errores y eficiente.

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.

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*.

Regla 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.

8.10 Formas de Optimización

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.).

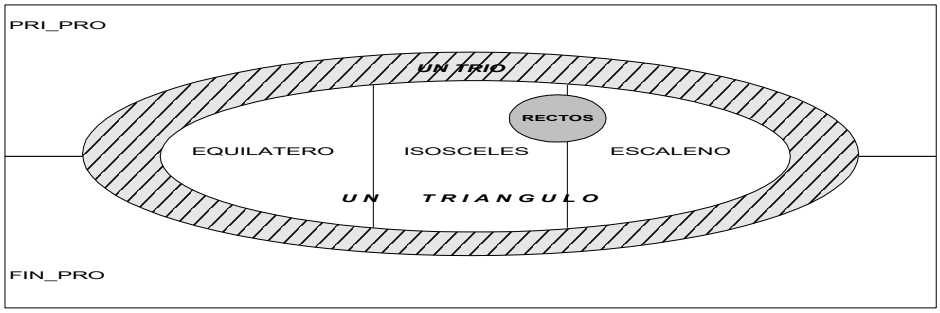
La mayoría de los autores (*Jackson, Boria, Rice y Rice*) eligen los modelos de clasificación para ilustrar la optimización por algoritmos.
Para ilustrar el objetivo **más pequeño** se emplea el caso de *determinación de tipos de triángulos*.
Para ilustrar el objetivo **más rápido** se aprovechará el método de clasificación de intercambio directo comunmente llamado "burbuja" (desarrollado en el tema de clasificacion).

8.10.1 Optimización haciendo el algoritmo más *pequeño*

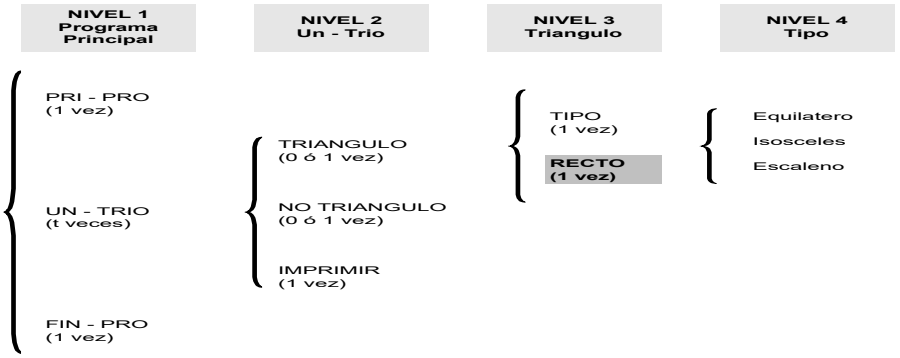
Dado un conjunto de tríos de valores A, B, C (mayores que cero) determinar, de entre los que forman triángulo, los distintos tipos (escaleno, isósceles, equilátero) y también cual de ellos es recto. Informar de acuerdo a la figura que contiene el modelo de la salida.

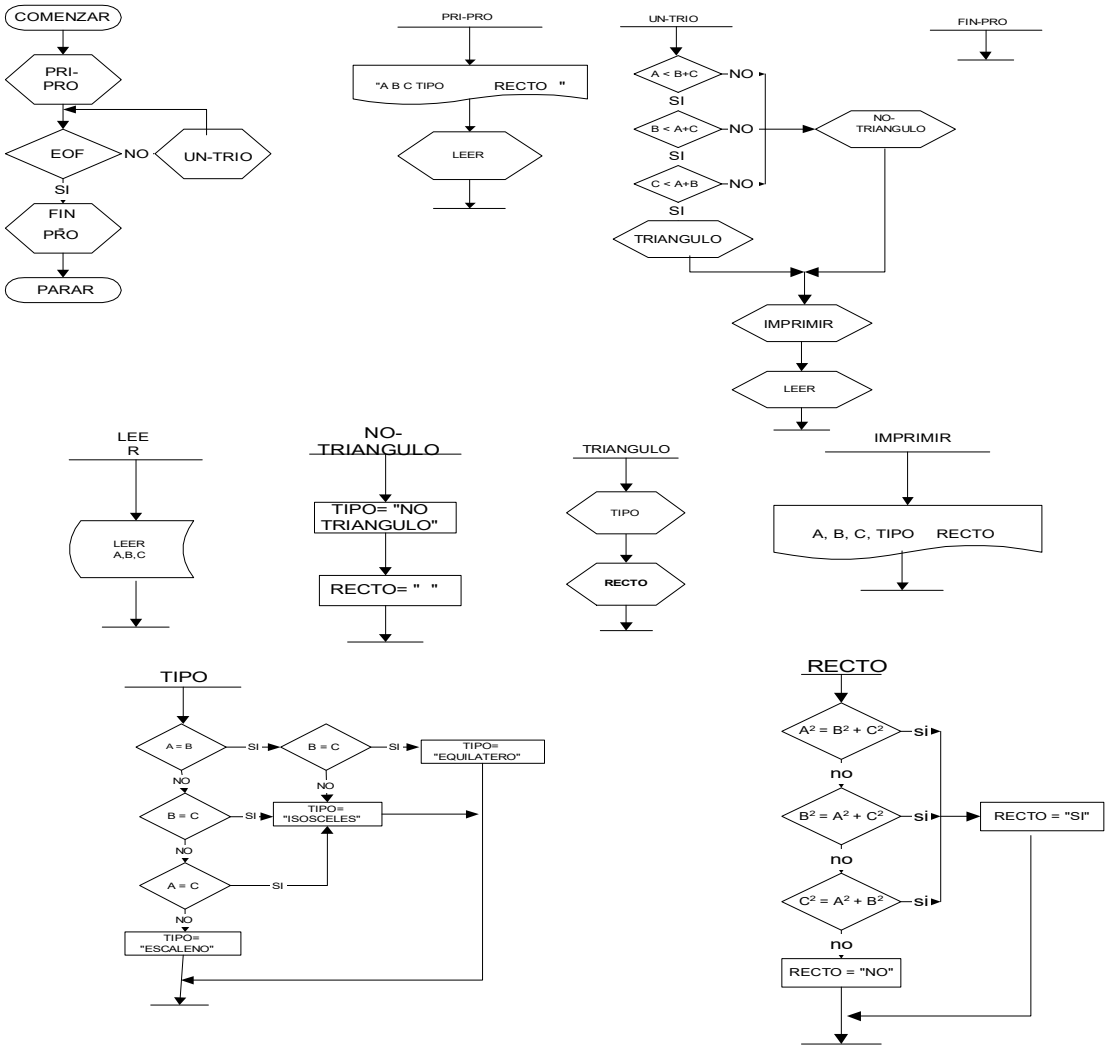
LADO 1	LADO 2	LADO 3	TIPO	RECTO
--	--	--	Escaleno	SI
--	--	--	No triángulo	
--	--	--	Equilátero	NO
--	--	--	Escaleno	SI

A)


PRI_PRO
UN TRIO
RECTOS
EQUILATERO
ISOSCELES
ESCALENO
UN TRIANGULO
FIN_PRO

B)

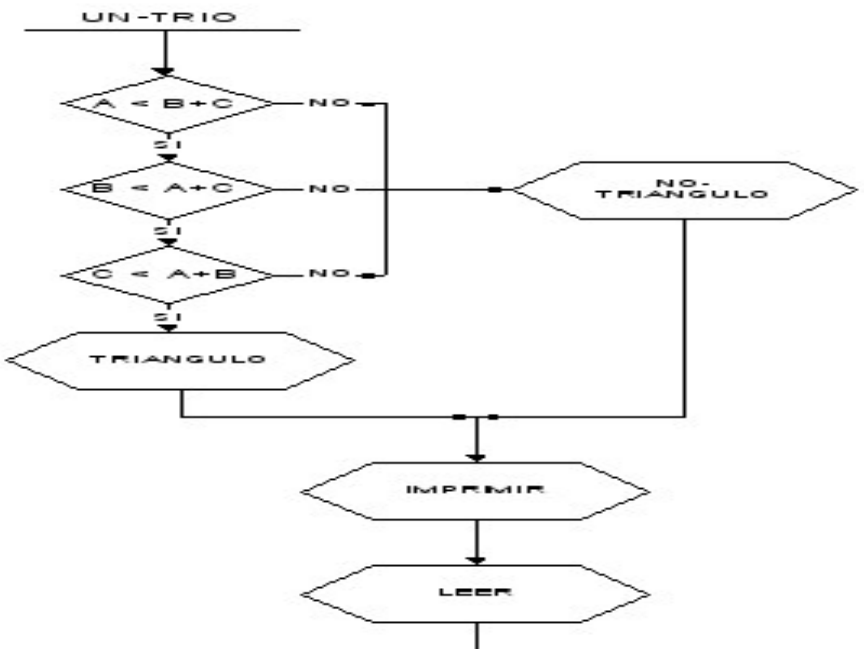

NIVEL 1
Programa Principal
NIVEL 2
Un - Trio
NIVEL 3
Triangulo
NIVEL 4
Tipo
PRI - PRO
(1 vez)
UN - TRIO
(t veces)
FIN - PRO
(1 vez)
TRIANGULO
(0 ó 1 vez)
NO TRIANGULO
(0 ó 1 vez)
IMPRIMIR
(1 vez)
TIPO
(1 vez)
RECTO
(1 vez)
Equilatero
Isosceles
Escaleno



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:

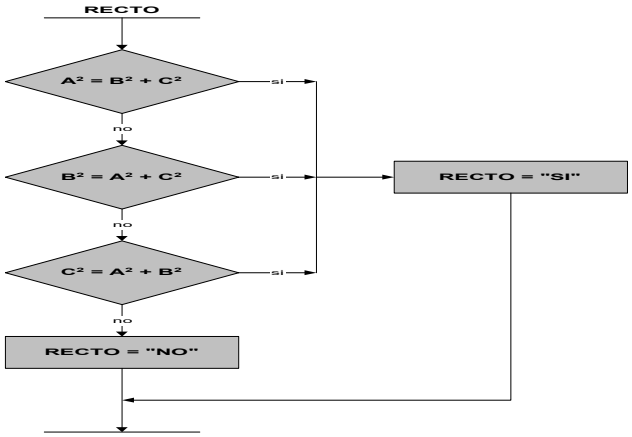
Primera optimizacion

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.



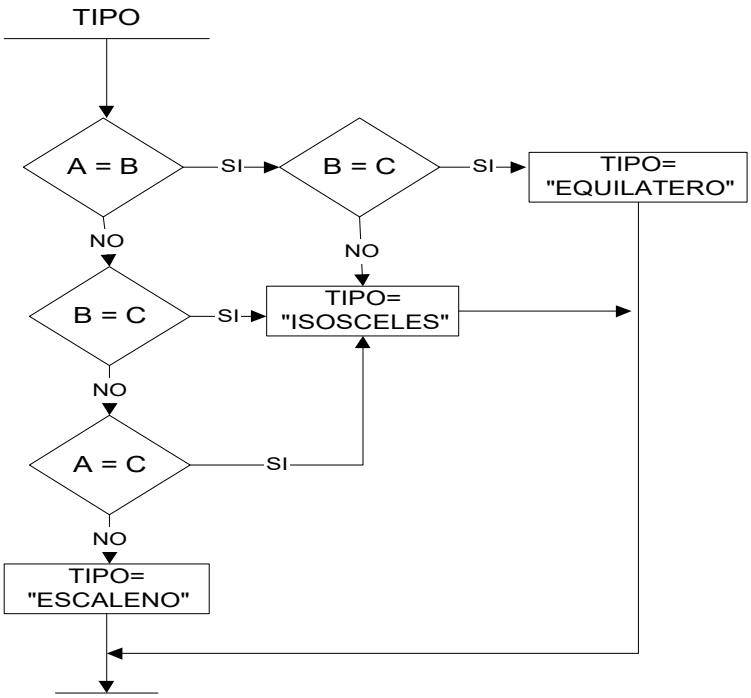
Segunda optimizacion

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.



Tercera optimizacion

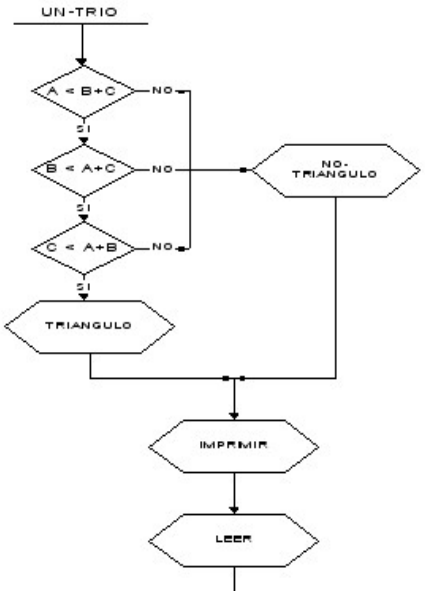
En la rutina **tipo** se advierte que deben compararse todos contra todos los lados para determinar el tipo de triángulo. Probablemente el tener juntos los posibles lados iguales, implique una simplificación del algoritmo.

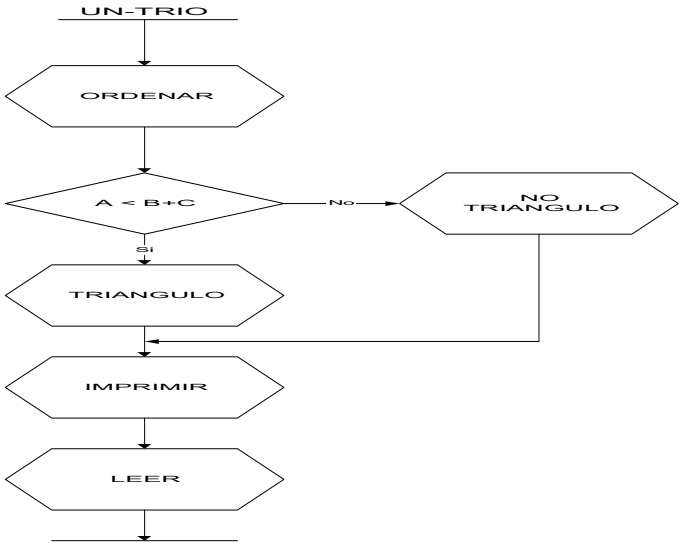


Transformar las condiciones necesarias con que fueron definidas inicialmente, en dos condiciones suficientes (puntos 1 y 2) y simplificar el análisis de **tipo** en el punto 3, se logra si los lados están clasificados en orden descendente.

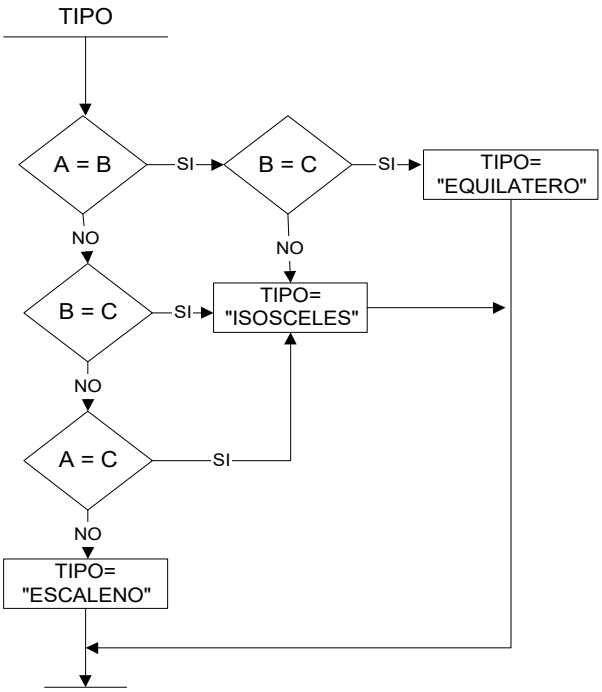
Aprovechando una rutina desarrollada que ordena en forma descendente un conjunto de números, se invoca a la misma al comienzo de la rutina **un_trío** y a partir de allí se optimiza el algoritmo con los lados ordenados, quedando como se indica a continuación las rutinas **un-trío , tipo y recto.**

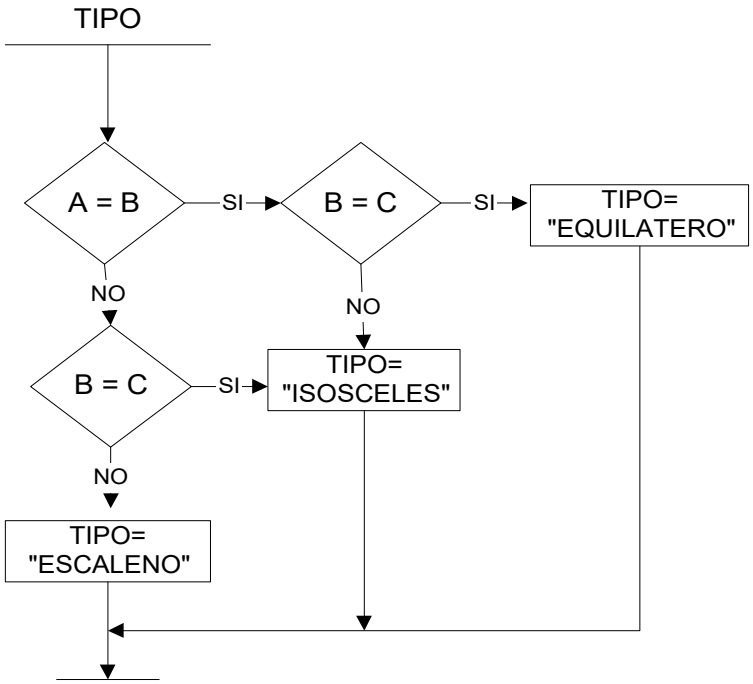
Primera optimizacion: Un_trío: Se suprimen las condiciones de B<A+C y C<A+B.





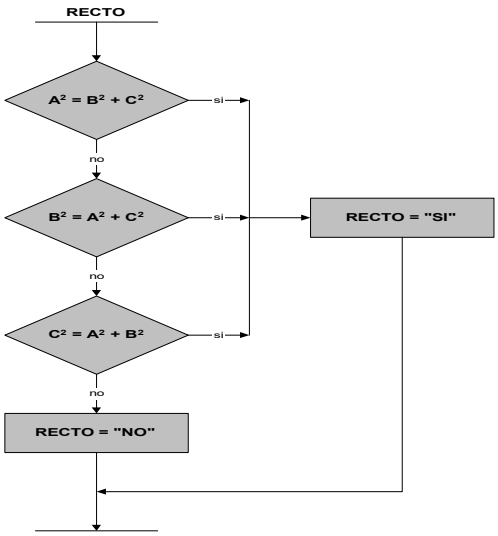
Segunda optimizacion **Tipo**: En el caso de triángulos isósceles, al ordenar los lados, los que son iguales están contiguos, por lo que la condición A = C es innecesaria.

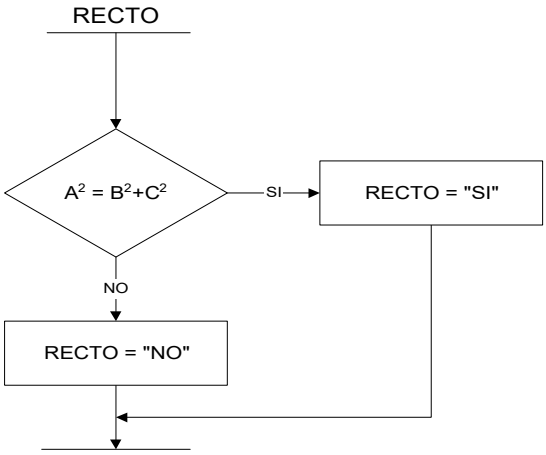




Tercera optimizacion Recto: Se suprimen las condiciones

$$B^2 = A^2 + C^2 \text{ y } C^2 = A^2 + B^2$$



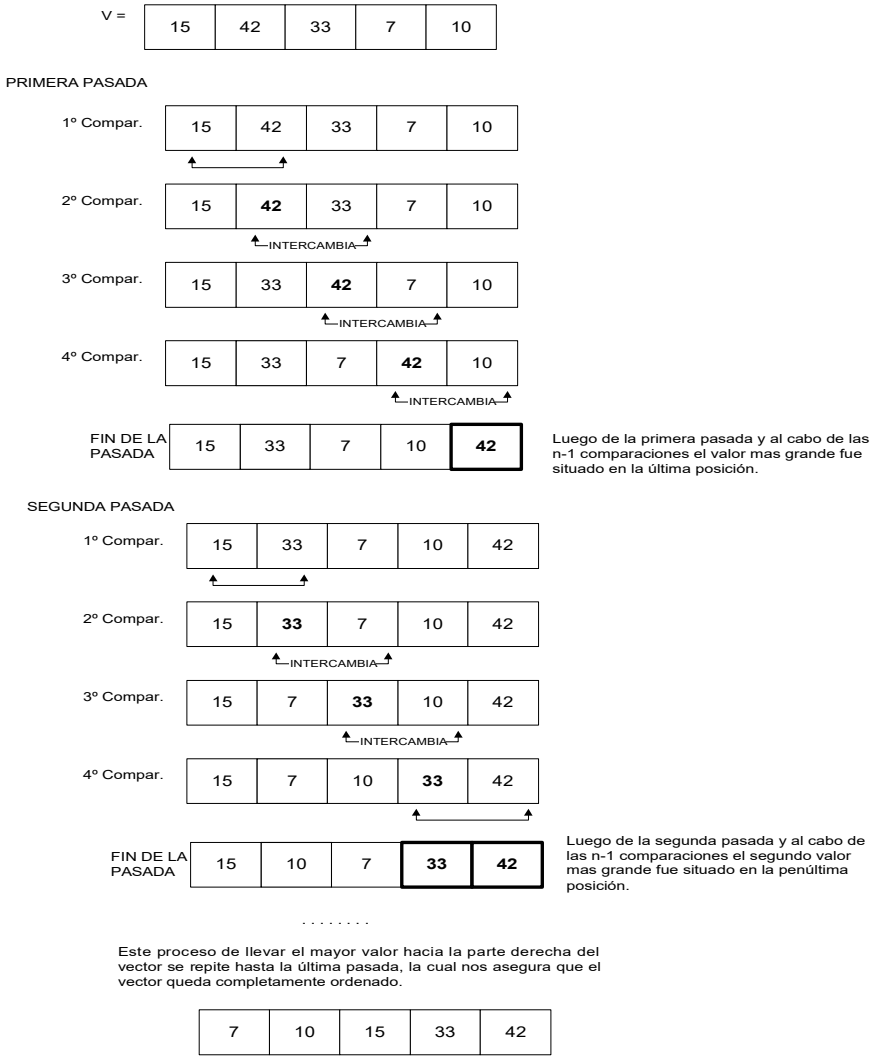


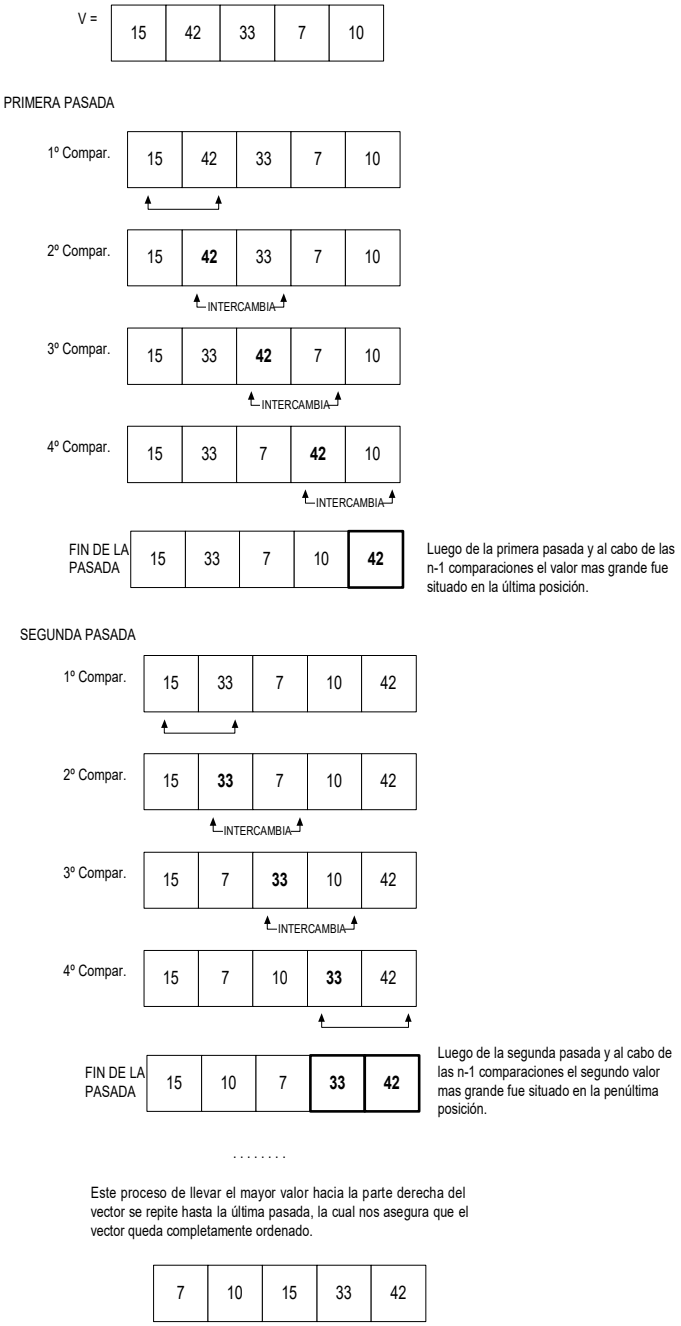
8.10.2 Optimización haciendo el algoritmo más rápido.

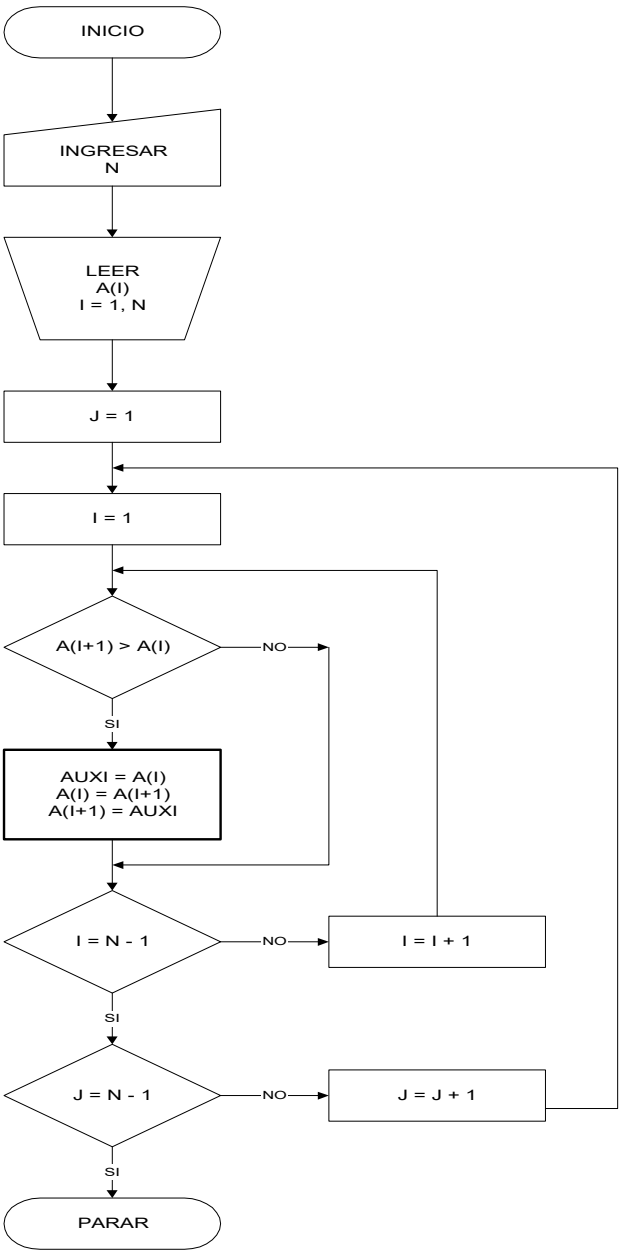
Aprovechando la utilización de la rutina de ordenación de los lados del ejemplo anterior (ORDENAR), a los efectos de introducir otros recursos de optimización que no hagan a la reducción del programa - lo más **pequeño** posible - , sino a la reducción de los tiempos de ejecución - lo más **rápido**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





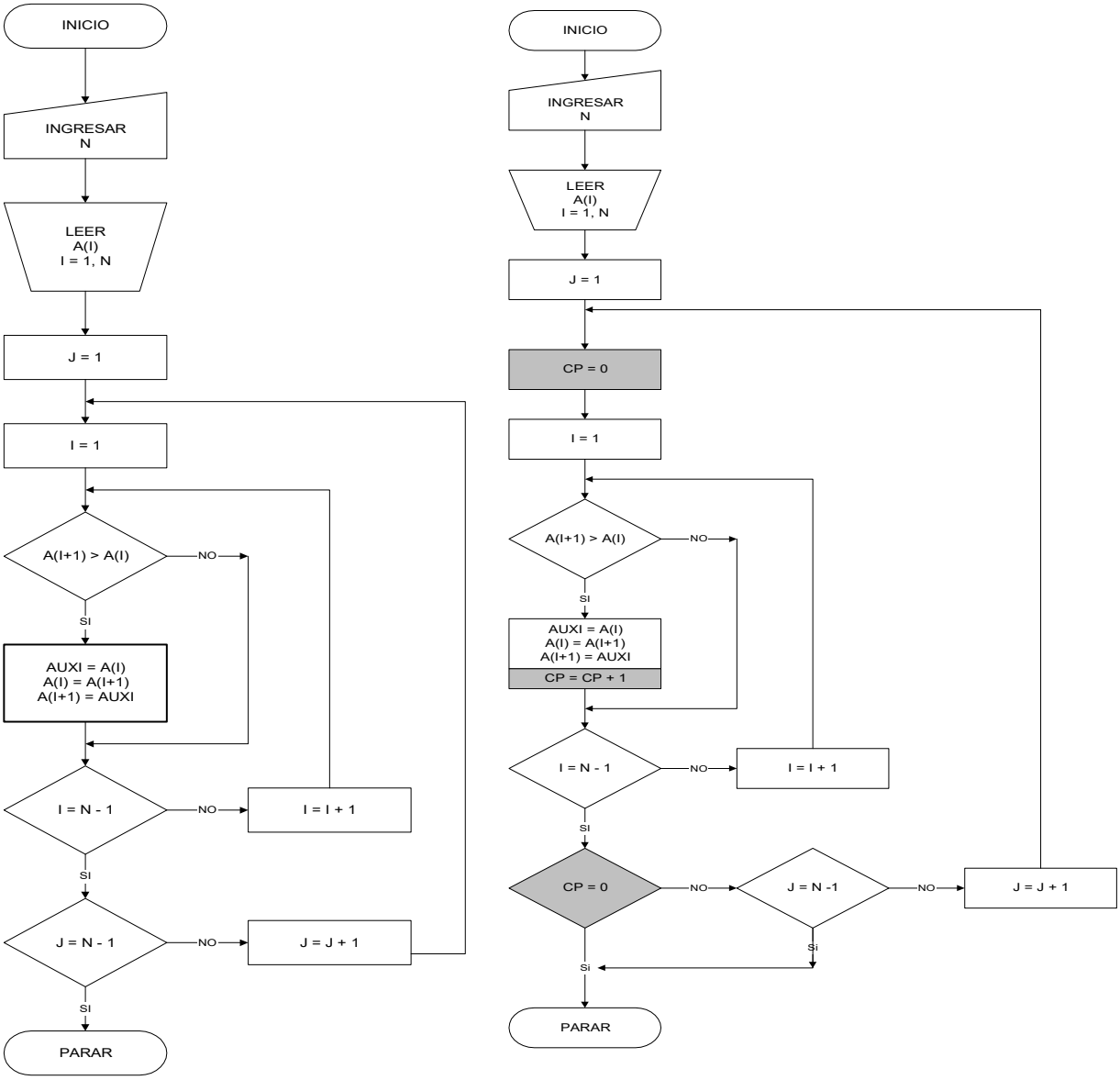


Al analizar el algoritmo se observa que se realiza en cada pasada *n-1 comparaciones* para analizar los *n-1* elementos adyacentes, pero además se realizan *n-1 pasadas* garantizando la ordenación total de la lista, realizando en total *(n-1) * (n-1) comparaciones* resultando el tiempo de ejecución del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

1° optimizacion

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.

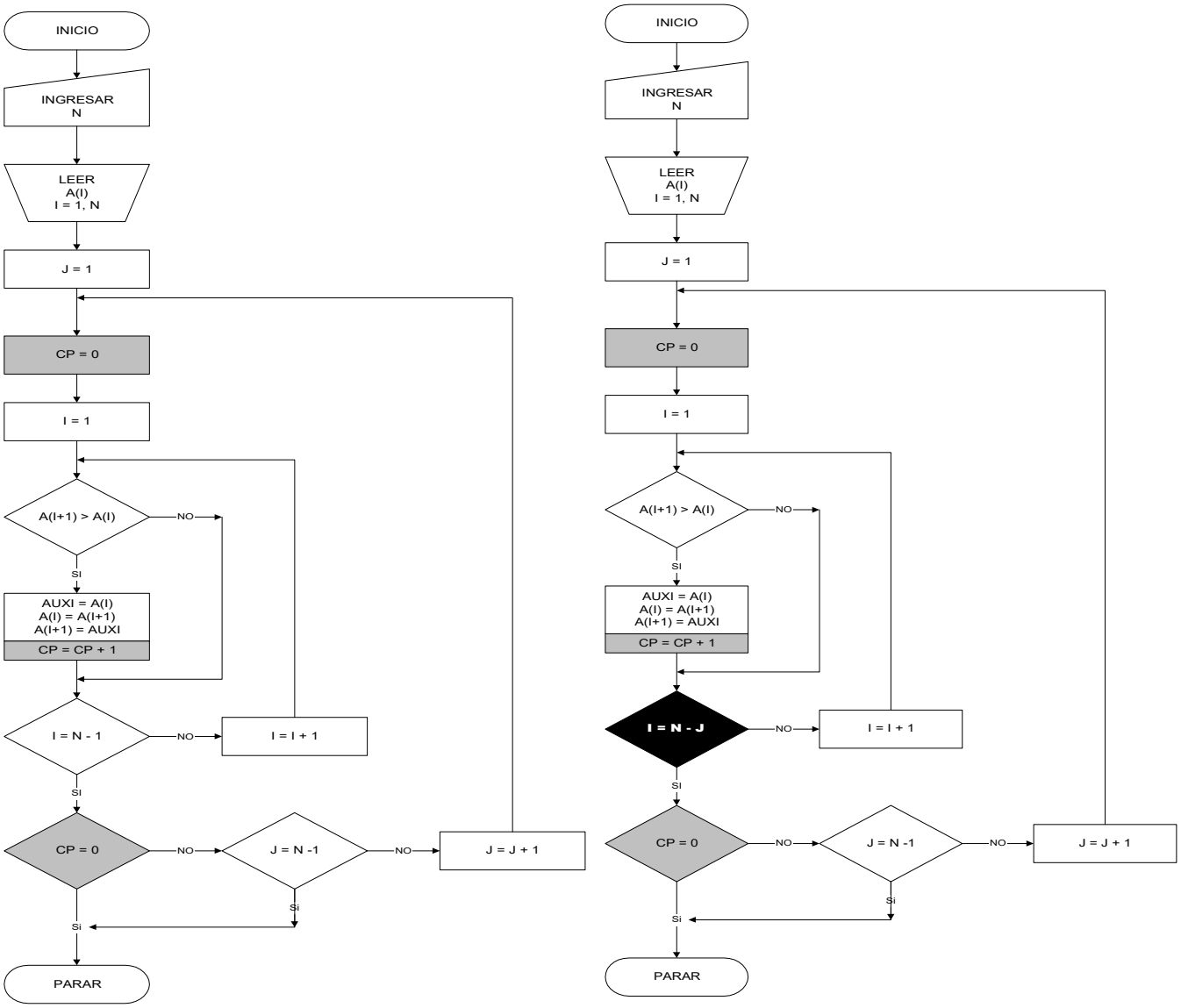
De esta manera cuando no existan permutaciones, o sea CP = 0, se detendrá el algoritmo realizando un número menor de comparaciones al calculado anteriormente



2° Optimización

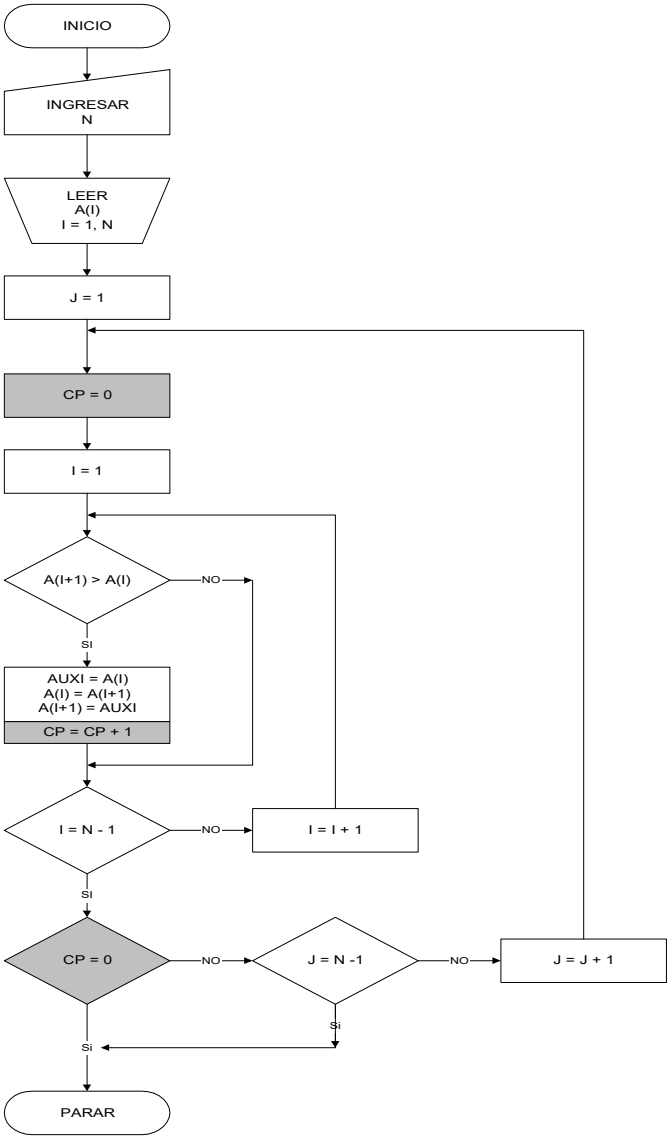
Como en la primer pasada el algoritmo lleva el elemento más chico o más grande al final de la lista, en la siguiente pasada lleva el segundo más chico o más grande a la penúltima posición y a continuación hace una comparación ociosa, por lo que en esta segunda optimización lo que se pretende es que en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

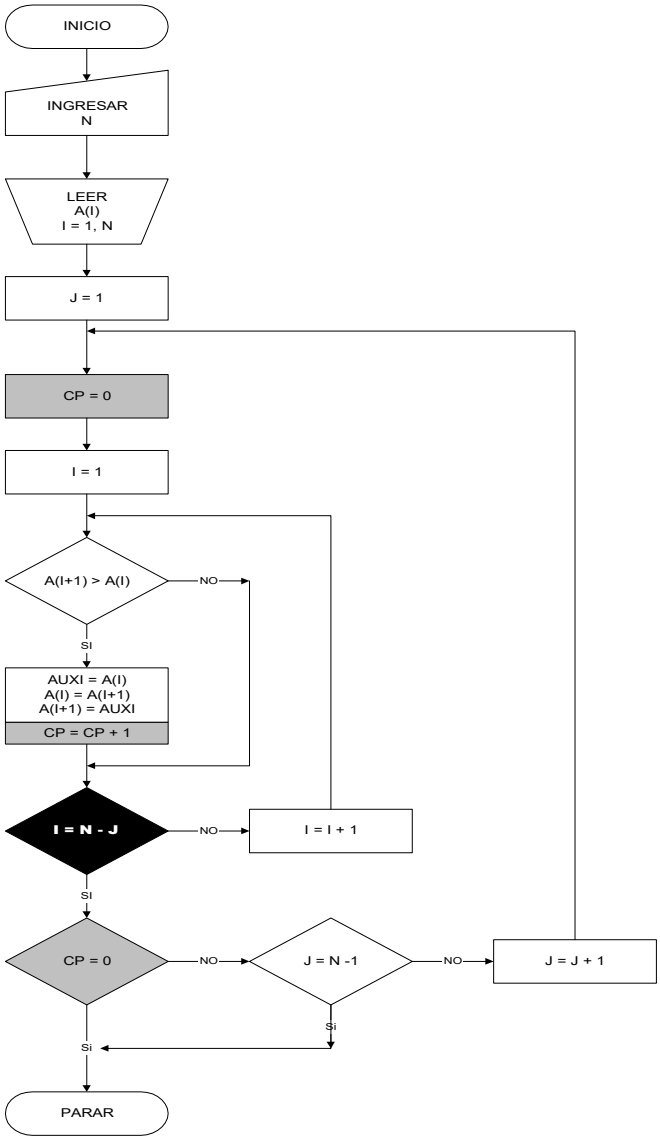
Esto se logra cambiando en el algoritmo el control de salida de la iteración más interna **N - 1 por N – J** quedando como se muestra en la figura.



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:

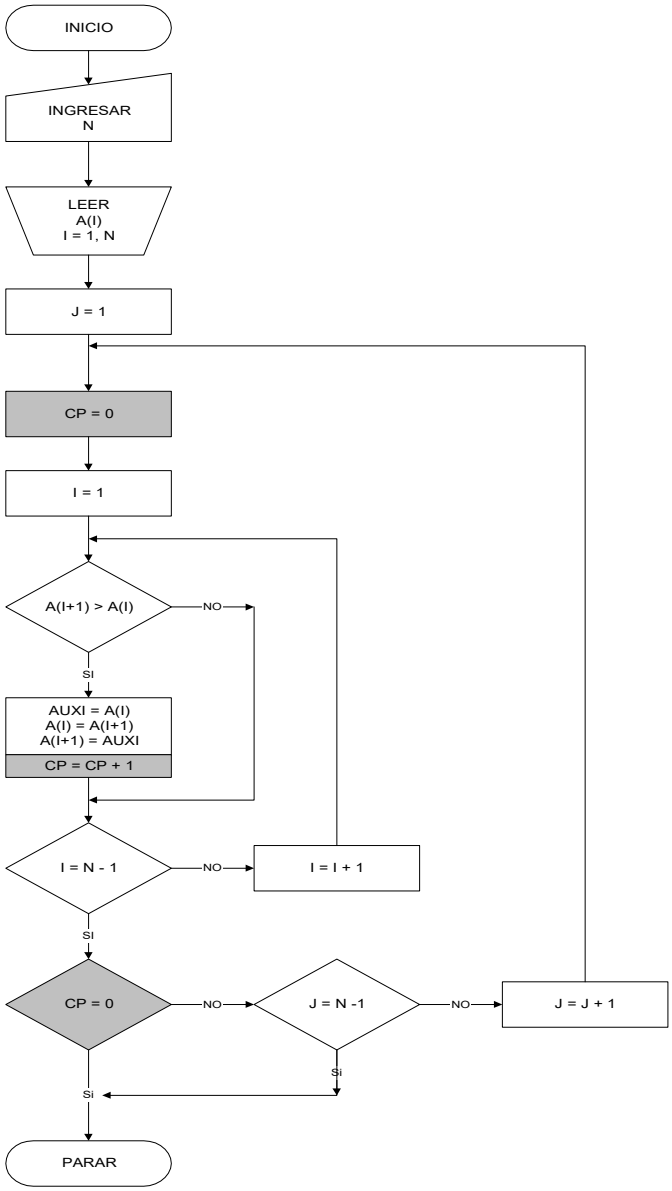
$C = ((n-1) + 1) * (n-1) / 2 = (n^2 - n) / 2$





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.



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.

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.

La excelencia no solo radica en el diseño del algoritmo en sí, sino también en su rendimiento.

La eficiencia de un algoritmo es un factor crítico que determina su impacto en la vida real. Muchos factores influyen en qué tan performante es un algoritmo, se destaca la importancia de uno de ellos: la cantidad de pasos.

Es cierto que existen muchos otros factores, como la velocidad del CPU, el tipo de compilador, la memoria disponible y la eficiencia del código generado, que también influyen en el rendimiento de un algoritmo. Sin embargo, centrémonos en la cantidad de pasos por un momento.

Cada paso adicional en un algoritmo es una oportunidad para ralentizar su ejecución. Cada instrucción innecesaria es un obstáculo en el camino hacia la eficiencia. Como desarrolladores, el objetivo es encontrar formas de reducir estos pasos innecesarios sin comprometer la lógica o la precisión de nuestro algoritmo.

La optimización de algoritmos es un proceso continuo y desafiante. Requiere un profundo conocimiento de la complejidad algorítmica, así como la capacidad de aplicar técnicas avanzadas de optimización. Pero el esfuerzo vale la pena. Al reducir la cantidad de pasos en los algoritmos, no solo mejoramos el rendimiento, sino que también contribuimos a una experiencia de usuario más rápida y eficiente.

Cada vez que logramos una mejora significativa en la eficiencia, no solo celebramos un logro técnico, sino que también generamos un impacto tangible en el mundo real.

Así que, siempre hay que tener en mente mantener siempre la optimización de algoritmos. Recordemos que cada paso cuenta y que, al reducirlos, estamos creando software más eficiente y poderoso.

8.11 Herramientas de Optimización

Software comercial:

- **Compuware DevPartner Studio**, suite de depuración y optimización para Microsoft Visual Studio 6.0 y .NET. Abarca todos los lenguajes de Visual Studio e incluye una herramienta para control y optimización de código fuente bajo Visual Basic .NET y Visual C#.

http://www.compuware.com/products/devpartner/studio.htm

- **Rational Purify**, producto de IBM muy similar al anterior. Tiene versiones para Windows(Visual Studio .NET), Linux y Unix. al contrario que DevPartner Studio, Purify no instrumenta el código fuente(sólo Windows y Unix), haciendo posible la depuración de programas sin código fuente disponible. No necesita una máquina tan potente como DevPartner studio.

http://www.rational.com/products/pqc/index.jsp

- **Intel VTune**, suite de optimización con versiones Windows(Visual Studio .NET) y Linux. Recoge multitud de datos acerca de la ejecución de nuestros programas, y permite usarlo tambien de forma remota.

http://www.intel.com/software/products/vtune/

Software libre:

- **Gcov, Gprof, Memprof**, herramientas para realizar tests de cobertura, optimización y detección de errores de memoria, respectivamente.
GNU Cov: http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html
GNU Prof: http://www.gnu.org/manual/gprof-2.9.1/gprof.html
Memprof: http://www.gnome.org/projects/memprof/

- **Electric Fence**, herramienta pasiva que detecta errores relacionados con la memoria, como acceso fuera de rango a arrays, etc. No requiere tocar el código de la aplicación, ya que se introduce en tiempo de enlazado. http://perens.com/FreeSoftware/

Bibliografia

- W. I. Salmon. Introducción a la computación con Turbo Pascal. Addison-Wesley Iberoamericana, 1993.
- J. Castro, F. Cucker, F. Messeguer, A. Rubio, Ll. Solano, y B. Valles. Curso de programación. McGraw-Hill, 1993.
- Se 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.
- S. Alagíc y M.A. Arbib. The design of well-structured and correct programs. Springer Verlag, 1978.
- Es una referencia obligada entre los libros orientados hacia la verificación con un enfoque formal.
- R. S. Pressman. Ingeniería del Software. Un enfoque práctico. McGraw-Hill, 2005.
- Algunos de los conceptos contenidos en el tema provienen de la ingeniería del software.
- Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642.
- ALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2
- FUNDAMENTOS 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.
- PROGRAMACIÓN; Castor F. Herrmann, María E. Valesani.; 2001; Editorial: MOGLIA S.R.L..ISBN: 9874338326.