Tema 2. Complejidad de algoritmos iterativos

Tema 2. Complejidad de algoritmos iterativos

1. Introducción

COMPLEJIDAD de un algoritmo: eficiencia: coste: rendimiento: medida de la cantidad de recursos necesarios para su ejecución.

Complejidad temporal: Eficiencia temporal: tiempo de ejecución de un algoritmo, en función del tamaño del problema.

Complejidad espacial: Eficiencia espacial: espacio de memoria utilizado para ejecutar el algoritmo, en función del tamaño del problema.

Para medir la complejidad se pueden emplear dos tipos de análisis:

- **A POSTERIORI: Pruebas: benchmarking**: pruebas con una muestra típica de los posibles datos de entrada del programa, midiendo los tiempos de ejecución para cada uno de los casos; después se realizan análisis estadísticos para inferir el rendimiento del programa. Dependiente de la máquina. Es un método relativamente fácil pero las conclusiones no siempre son fiables. *Medida real.*

- **A PRIORI**: **Análisis**: se usan procedimientos matemáticos para determinar el coste. Las conclusiones son muy fiables pero, a veces, su realización es difícil a efectos prácticos. Independiente de la máquina. *Medida teórica.*

Ejemplo: Algoritmo que suma todos los elementos de una matriz cuadrada de tamaño **n**.

```
suma = 0;                                   1 * Ta
i = 0;                                      1 * Ta
while (i < n) {                             (n+1) * Tc
        j = 0;                              n * Ta
        while (j < n) {                     n * (n + 1) * Tc
                suma = suma + Matriz[i][j]; n * n * (Ta + Ts)
                j ++;                       n * n * (Ta + Ts)
        }
        i ++;                               n * (Ta + Ts)
}
```

Tiempo total = $an^2 + bn + c$, donde **a**, **b** y **c** están en función del lenguaje, la máquina, el compilador, etc.

Usaremos una función **T(*n*)** para representar el número de unidades de tiempo que un algoritmo tarda en ejecutarse para un problema de tamaño *n*.

Como el tiempo de ejecución varía mucho de un ordenador a otro, **T(*n*) se toma como el número de instrucciones simples** (asignaciones, comparaciones, operaciones aritméticas) **que se ejecutan** (suponiendo un ordenador ideal donde cada una de estas instrucciones consume una unidad de tiempo).

Si T(n) es el coste de un algoritmo, podemos asumir que n >= 0 y que T(n) es positiva para cualquier valor de n.

- Mejor invertir en un buen algoritmo que en una buena máquina.

Supongamos un algoritmo con $T(n) = 10^{-4} * 2^n$ segundos

n	Tiempo
10	0.1 segundos
20	2 minutos
30	24 horas
38	> 1 año

Si ahora ejecutamos el algoritmo en una máquina 100 veces más rápida.

$T(n) = 10^{-6} * 2^n$ segundos

n	Tiempo
10	0.001 segundos
20	1 segundo
30	20 minutos
38	35 días
45	1 año

Si ahora conseguimos un algoritmo mejor para resolver el mismo problema: $T(n) = 10^{-2} * n^3$ segundos

n	Tiempo
10	10 segundos
20	80 segundos
200	1 día
1500	1 año

Otro ejemplo:

Supongamos que para un cierto problema hemos hallado dos algoritmos A y B que lo resuelven. Evaluamos sus costes respectivos y obtenemos:

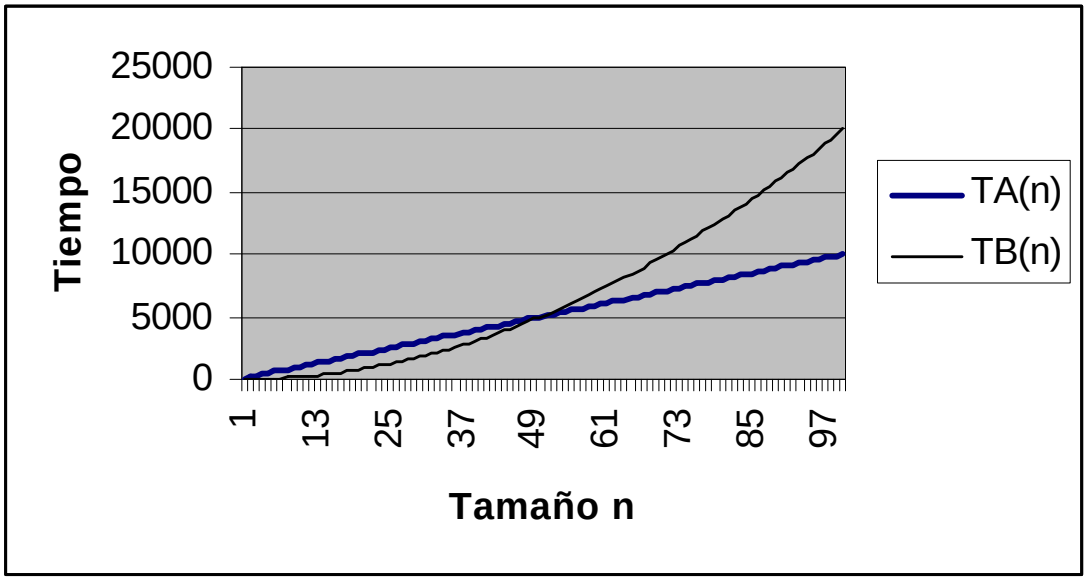
$T_A(n) = 100 * n$

$T_B(n) = 2 * n^2$

¿Cuál de los dos algoritmos es más eficiente?

Si n < 50, B es más eficiente que A, aunque no demasiado.

Si n > = 50, A es mucho más eficiente que B. La diferencia es más apreciable cuanto mayor se hace *n*.



Es mucho más importante la forma de la función (*n* en lugar de n^2) que las constantes que intervienen (2 frente a 100).

Además, como el tiempo de ejecución del algoritmo depende del ordenador en el que lo ejecutemos (aparecerá multiplicado por una determinada constante), no es necesario que nos preocupemos por esos factores multiplicativos a la hora de analizar la complejidad de un algoritmo.

1.1. Tamaño del problema

Tamaño del problema: depende generalmente de los datos de entrada:

- En un algoritmo para calcular la cantidad de dígitos de un número, el tamaño del problema es el número de cifras.
- En un algoritmo de ordenación, es el número de elementos que hay que ordenar.
- En un algoritmo de búsqueda, es el número de elementos entre los que hay que buscar el elemento dado.
- En un algoritmo que actúa sobre un conjunto, es el número de elementos que pertenecen al conjunto.

Con mucha frecuencia, el coste de un algoritmo depende de los datos de entrada particulares sobre los que operes, y no sólo del tamaño de esos datos. En esos casos, podemos distinguir entre:

- $T_{sup}(n)$: **coste del caso peor**: coste máximo del algoritmo
- $T_{inf}(n)$: **coste del caso mejor:** coste mínimo del algoritmo
- $T_{med}(n)$: **coste del caso medio:** coste promedio

¿Qué nos interesa medir?:

- *Operaciones elementales* (OE): aquellas que el ordenador realiza en tiempo acotado por una constante:
  - operaciones aritméticas básicas, asignaciones, saltos (llamadas a funciones, retorno de funciones, etc), comparaciones y accesos a estructuras indexadas básicas (vectores y matrices)
- *Operaciones significativas*

Medidas significativas de problemas comunes

Problema: búsqueda de un elemento
Medida: número de comparaciones con un elemento
Tamaño del problema (n): número de elementos

Problema: multiplicación de dos matrices de números reales
Medida: número de sumas y productos de reales
Tamaño del problema (n): dimensión de las matrices

Problema: resolución de un sistema de ecuaciones lineales
Medida: número de operaciones aritméticas
Tamaño del problema (n,m): número de ecuaciones y de incógnitas

Problema: ordenación de una serie de elementos
Medida: número de comparaciones y de intercambios
Tamaño del problema (n): número de elementos

Problema: recorrido de un árbol binario
Medida: número de enlaces procesados
Tamaño del problema (n): número de nodos del árbol

RESUMIENDO:

- El tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones (elementales o significativas) que realiza el algoritmo para un tamaño de entrada dado.

Ejemplo: ¿Cuál de los siguientes algoritmos es mejor para un tamaño *n*?

- ¿Qué nos interesa medir para comparar objetivamente los dos algoritmos? ¿Operaciones elementales? ¿Comparaciones con respecto a los elementos del vector?

```
int Pertenece1 (int A[n], int x) {
        int encontrado=0;
        for (int i=0; i<n; i++)
                if (x== A[i]) encontrado=1;
        return encontrado;
}
```

```
int Pertenece2 (int A[n], int x) {
        int i=0, encontrado=0;
        while (i < n && !encontrado) {
                if (x == A[i]) encontrado=1;
                else i++;
        }
        return encontrado;
}
```

El segundo algoritmo no sólo depende del tamaño de la entrada, sino también del orden en el que aparezcan los elementos.

2. Notaciones asintóticas

Para expresar las cotas de complejidad se utiliza una notación asintótica. La más usual es la notación **O (O grande)** que proporciona una cota superior de la complejidad del algoritmo.

Definición formal de la notación O (O grande)

Sea f(*n*) una función definida sobre los números enteros positivos *n*. Diremos que T(*n*) es O(f(*n*)) sii $\exists$ n_0, c >= 0 t.q. $\forall$n >= n_0, T(n) <= c * f(n).

Es decir, a partir de un cierto valor, T(*n*) está **acotada superiormente** por f(*n*) multiplicada por una constante.

Ejemplo: $T(n) = 3n^2 + 2n$ es $O(n^2)$

$n_0 = 0$ c = 5 $\forall n >= n_0$, $T(n) <= c * n^2$

$3n^2 + 2n <= 5n^2$

Definición formal de la notación W

Sea g(*n*) una función definida sobre los números enteros positivos *n*.

T(*n*) es Ω(g(*n*)) sii $\exists$ n_0, c >= 0 t.q. $\forall$n >= n_0, T(n) >= c * g(n).

Es decir, a partir de un cierto valor, T(*n*) está **acotada inferiormente** por g(*n*) multiplicada por una constante.

Ejemplo: $T(n) = 3n^2 + 2n$ es $\Omega(n^2)$

$n_0 = 0$ c = 3 $\forall n >= n_0$, $T(n) >= c * n^2$

$3n^2 + 2n >= 3n^2$

Definición formal de la notación asintótica q

Sea g(*n*) una función definida sobre los números enteros positivos *n*.

T(*n*) es θ(g(*n*)) sii ∃ n_0, c_1, c_2 >= 0 t.q.

$$\forall n >= n_0,\ c_1 * g(n) <= T(n) <= c_2 * g(n).$$

Es decir, a partir de un cierto valor, T(*n*) está **acotada superior e inferiormente (orden exacto)** por g(*n*).

Ejemplo: $T(n) = 3n^2 + 2n$ es $\theta(n^2)$

2.1. Reglas de simplificación y cálculo de la complejidad

- Regla de la suma

Supongamos P1 y P2, dos fragmentos de programa con tiempos de ejecución $T_1(n)$ y $T_2(n)$. Si $T_1(n)$ es O(f(n)) y $T_2(n)$ es O(g(n)), entonces la ejecución de los dos fragmentos de programa es de orden O(máx(f(n), g(n)).

$T_1(n) + T_2(n)$ es máx(O(f(n)), O(g(n)))

- Regla del producto

Supongamos P1 y P2, dos fragmentos de programa con tiempos de ejecución $T_1(n)$ y $T_2(n)$. Si $T_1(n)$ es O(f(n)) y $T_2(n)$ es O(g(n)), entonces:

$T_1(n)*T_2(n)$ es O(f(n)*g(n))

- En particular, si T(n) es O(c f(n)), con c constante positiva, entonces, T(n) es O(f(n)).

- Si el tiempo de ejecución sigue una función polinómica, los términos de orden inferior no importan
  Si $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, $T(n)$ es $O(n^k)$

- Si se realiza una operación para n/2, n/4, . . ., $n/2^p$, con $n = 2^p$, entonces el número de operaciones será $p = \log n$, y la complejidad es $O(\log n)$.

- La base de los logaritmos no importa
  Usamos siempre $O(\log n)$ sin indicar la base ($\log_b n = \log_c n * \log_b c$).

2.2. Complejidad de las instrucciones básicas (algoritmos iterativos)

Instrucciones de asignación, lectura, escritura, comparaciones (Operaciones elementales)

O(1)

Secuencia

S1; S2 **$O(\text{máx}(t_{S1}, t_{S2}))$**

Instrucción condicional

Si B entonces S1 si no S2 finsi **$O(\text{máx}(t_B, t_{S1}, t_{S2}))$**

Bucles

Mientras B hacer
S **$O(\text{max\_iter} * (T_S + T_B))$**
Finmientras (**max_iter** en el caso peor)

- Es posible que S dependa también del número de iteración en el que nos encontremos. En ese caso: $T(n)$ es $O(\sum_{i=1}^{f2(n)} t_s(n, i))$

2.3. Problemas

Para cada problema:

¿Cuál es el coste?

¿Cuántas operaciones elementales se realizan?

```
for (int i=1; i<=n;i++)
        x ++;
```

```
for (int i=1; i<=n;i++)
        for (int j= 1; j<=n;j++)
                for (k = 1; k<=n; k++)
                        x ++;
```

```
i = 1;
do {
        x ++;
        i = i + 2;
} while ( i > n);
```

```
for (int i = 1;i<=n; i++)
        for (int j=1; j<=i; j++)
                x ++;
```

```
x = 1;
while (x < N)
        x = 2 * x;
```

2.4. Funciones de complejidad más usuales

O(1) Complejidad constante: deseable, pero difícil o imposible de conseguir

O(log n) Complejidad logarítmica: habitualmente de solución recursiva; muy buena

O(n) Complejidad lineal: bastante buena y usual

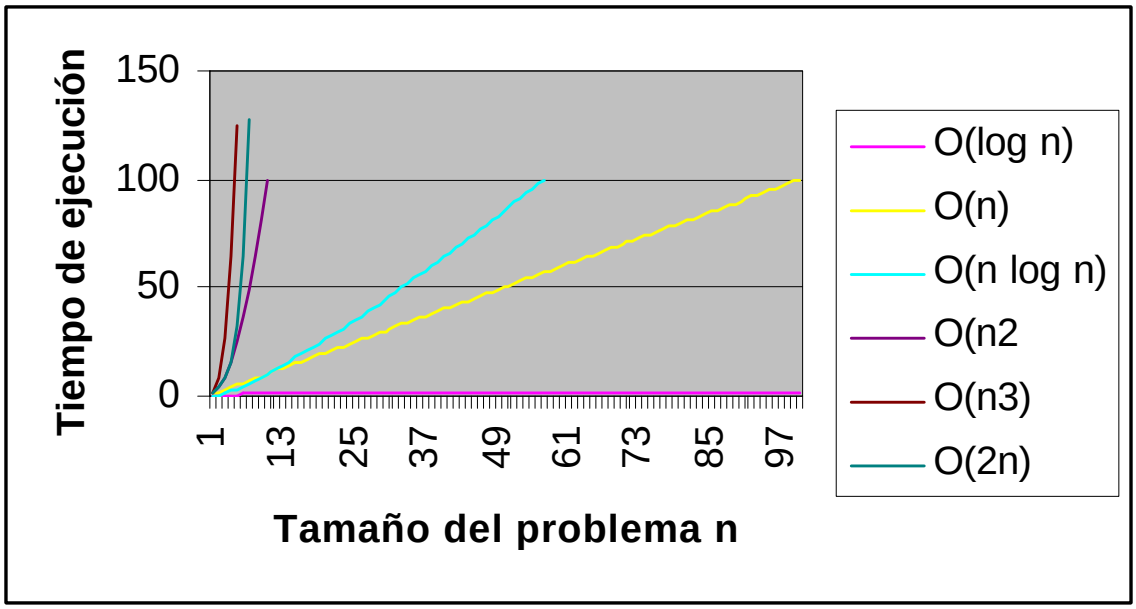
O(n*log n) Frecuente en problemas recursivos; bastante buena

O(n^2) Complejidad cuadrática: aparece en bucles y similares

O(n^3) *Complejidad cúbica:* para muchos datos crece en exceso

O(n^k) *Complejidad polinómica (k > 3):* si K crece en exceso, la complejidad tiende a ser bastante alta

O(2^n) *Complejidad exponencial: Explosión combinatoria:* no es práctico cuando n es grande. Debe evitarse.



3. Cálculo del caso medio

En muchos algoritmos, el tiempo de ejecución depende, no sólo del tamaño de los datos de entrada, sino también de los valores específicos. Así, suelen estudiarse tres casos **para un mismo algoritmo con un tamaño n determinado**:

- **Caso mejor:** traza que realiza **menos** instrucciones.

- **Caso peor:** traza que realiza **más** instrucciones.

- **Caso medio:** traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de entrada dado, con las probabilidades de que éstas ocurran.

$$\sum_i p(i)T(i)$$

Error frecuente: confundir el caso mejor con el caso de tamaño de entrada n = 1 (el que menos instrucciones realiza en cualquier caso)

- **Caso mejor → demasiado optimista**
- **Caso peor → demasiado pesimista**
- **Caso medio →muy exacto; ¿difícil de calcular?**

3.1. Cálculo de series

$$\sum_{i=1}^{n} c = cn$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}$$

$$\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

$$\sum_{i=2}^{n} \frac{1}{i} \approx \log n$$

$$\sum_{i=1}^{n} \log i \approx n \log n - \frac{n}{2}$$

Ejemplo: Pasos para calcular la complejidad de un algoritmo

- Algoritmo que suma los elementos de una matriz cuadrada de tamaño n:

```
Suma = 0;
i = 0;
while (i < n) {
    j = 0;
    while (j < n) {
        Suma = Suma + Matriz[i][j];
        j ++;
    }
    i++;
}
```

- **¿Cuál es el tamaño del problema? (Puede depender de más de una variable.)**
  - La dimensión de la matriz.
- **¿Cuál es la medida significativa?** ¿Qué queremos medir?
  - Número de asignaciones totales
  - Número de operaciones aritméticas
  - Número de accesos a la matriz
  - Número de operaciones elementales (ejercicio)

- Si tomamos como medida significativa el *número de asignaciones totales*: $2+\sum_{i=1}^{n}2+\sum_{i=1}^{n}\sum_{j=1}^{n}2=2+2n+2n^2$
- Si tomamos como medida significativa el *número de operaciones aritméticas*: $\sum_{i=1}^{n}1+\sum_{i=1}^{n}\sum_{j=1}^{n}2=n+2n^2$
- Si tomamos como medida significativa el *número de accesos a la matriz*: $\sum_{i=1}^{n}\sum_{j=1}^{n}1=n^2$
- **¿Se pueden distinguir casos mejor y peor?** NO; todos son È(n^2)
- **Calcular T(n) o medidas asintóticas.**

3.2. Ejercicio: algoritmos de búsqueda

Calcular el **caso medio**, tomando como medida significativa el número de comparaciones con elementos del vector. Calcularlo también considerando el número total de operaciones elementales.

```
int Pertenece1 (int A[n], int x) {
        int encontrado=0;
        for (int i=0; i<n; i++)
                if (x== A[i]) encontrado=1;
        return encontrado;
}
```

```
int Pertenece2 (int A[n], int x) {
        int i=0, encontrado=0;
        while (i < n && !encontrado) {
                if (x == A[i]) encontrado=1;
                else i++;
        }
        return encontrado;
}
```

```
int BúsqBinaria (int A[n]; int x) {

        int encontrado = 0, izq, der, mitad;

        izq = 0;
        der = n-1;
        while ((izq<=der) && !encontrado) {
                mitad = (izq + der) / 2;
                if (x == A[mitad])
                        encontrado = 1;
                else   if (x < A[mitad])
                                der = mitad – 1;
                        else
                                izq = mitad + 1;
        }
        return encontrado;
}
```

3.3. Ejemplo: Máximo de un vector

Casos peor, mejor y medio de un algoritmo que calcula el mayor en un vector.

```
FUNCTION Máximo (L: array[1..N] of integer): Integer;
VAR max: Integer;
BEGIN
        max := L[1];
        For i := 2 to N do
                If max < L[i] then
                        max := L[i];
        Máximo := max
END;
```

Supongamos que todos los elementos son distintos

¿Tamaño del problema?

¿Medida significativa?

¿Número de operaciones significativas? Caso mejor, peor y medio.

3.4. Ejemplo: MinMax

Casos peor, mejor y medio de los algoritmos que calculan el mayor y el menor números leídos en una secuencia.

```
Procedure MinMax (VAR Min, Max: Integer);
VAR x: Integer;
Begin
        if LeerNúmero(x) then
        Begin
                Min := x; Max:= x;
                while LeerNumero(x) do
                        if Min > x then
                                Min := x
                        else if Max < x then
                                Max := x
        End
End;

Procedure MinMax2 (VAR Min, Max: Integer);
VAR x,y: Integer; Salir: Boolean;
Begin
        if LeerNúmero(x) then
        Begin
                Min := x; Max:= x; y := x;
                Salir := False;
                while LeerNumero(x) do
                        if LeerNumero(y) then
                                if x < y then begin
                                        If Min> x then Min :=x;
                                        If Max < y then Max := y
                                end
                                else begin
                                        If Min > y then Min := y;
                                        If Max < x then Max := x
                                    end
                        else begin
                                if Min > x then Min := x;
                                if Max < x then Max := x
                            end
        End
End;
```

3.5. Ejemplo: algoritmos de ordenación de vectores

Calcular el número de comparaciones y movimientos para los casos mejor, peor y medio, en los algoritmos de la burbuja, inserción directa, selección directa e inserción binaria.

Algoritmo de la burbuja

```
for (i=1; i < N; i++) {
    for (j=0; j < N-i; j++) {
        if (V[j] > V[j+1]) {
            aux = V[j];
            V[j] = V[j+1];
            V[j+1] = aux;
        }
    }

}
```

Algoritmo de ordenación por inserción directa

```
for(i = 2; i<= N; i++) {
    X = V[i];
    V[0] = X; // centinela
    j = i - 1;
    while ( X < V[j] ) {
        V[j+1] = V[j];
        j = j - 1;
    }
    V[j+1] = X;
}
```

Algoritmo de ordenación por selección directa

```
for (i=0; i < N-1; i++) {
    k = i;
    Min = V[i];
    for (j=i+1; j<N; j++) {
        if (V[j] < Min) {
            k = j;
            Min = V[j];
        }
    }
    V[k] = V[i];
    V[i] = Min;
}
```

Algoritmo de ordenación por inserción binaria

```
for(i=1; i< N; i++) {
    X = V[i];
    Izq = 0;
    Der = i-1;
    while (Izq <= Der) {
        Medio = (Izq + Der)/2;
        if (X < V[Medio])
            Der = Medio - 1;
        else
            Izq = Medio + 1;
    }
    for(j = i - 1; j >= Izq; j--) {
        V[j+1] = V[j];
    }
    V[Izq] = X ;
}
```

3.6. Ejercicios

1.-

```
Begin
    i := N;
    While i>= 1 do begin
        j := 1;
        Repeat
            j := j*2
        Until j >= N;
        i := i div 2
    end
End;
```

2.-

```
Begin
    For i:= N downto 1 do begin
        j := 1;
        While j <= N do
            j := j*2;
    end
End;
```

3.-

```
Begin
    i := N;
    Repeat
        For j:= 1 to i do Write(j);
        i := i div 2
    Until i <= 1
End;
```

4.- Obtener el mejor y el peor caso:

```
Begin
    ReadLn(x);
    If x > 0 then
        For i := 1 to N do
            For j:= 1 to i do
                Write(j)
    Else
        For i := 1 to 5 do
            For j:= 1 to N do
                Write(i+j)
End;
```

5.- Obtener el mejor y el peor caso:

```
Begin
    ReadLn(x);
    If x = 0 then
        For i := 1 to N do
            For j:= 1 to M do
                Write(j)
    Else
        For i := 1 to N do begin
            j:= 1;
            While j <= N do
                j := j*2;
        end
End;
```

6.-

```
Begin
    Aux := L;
    While Aux <> NIL do
        Aux := Aux^.suce
End;
```

7.-

```
Function Auxiliar (Num: integer); integer;
Var i, j: integer;
Begin
    j := 0;
    for i := 1 to Num do
        j := j+i;
    Auxiliar := j
End;

Begin
    For i := 1 to N do begin
        a := Auxiliar(i);
        for j := 1 to a do
            b := j + a;
    end
End;
```