

Bloque 1. Conceptos y técnicas básicas en programación



1. Introducción
2. Datos y expresiones. Especificación de algoritmos
3. Estructuras algorítmicas básicas
4. Iteración y recursión
5. Iteración y recursión sobre secuencias
6. Iteración y recursión sobre tablas

Notas:



1. Introducción
2. Datos y expresiones. Especificación de algoritmos
3. Estructuras algorítmicas básicas
4. Iteración y recursión
5. Iteración y recursión sobre secuencias
6. Iteración y recursión sobre tablas
 - Concepto de tabla. Sintaxis. Operaciones sobre tablas. Recorrido de tablas. Búsqueda en tablas. Algoritmos sencillos de ordenación en tablas.

Formas de acceso a una colección

En el capítulo anterior hablamos de la importancia de las colecciones de datos

Las dos formas más comunes de acceder a una colección de datos

- **acceso secuencial**: para acceder a un elemento de la colección es preciso haber accedido a los anteriores
- **acceso directo**: se puede acceder a un elemento concreto de la colección directamente, a través de su posición, o índice
 - por ejemplo, acceso a las imágenes de un DVD
 - acceso a un elemento de un vector
 - la estructura que admite este acceso se llama **tabla** o también **vector**

Definición de la tabla

La **tabla** representa una colección de cero o más objetos de un determinado tipo, en la que se dispone de acceso directo

- cada elemento se identifica mediante un **índice**

El **índice** es un valor de un tipo discreto, en un rango determinado

- nos restringiremos a índices enteros en un rango, pues hay lenguajes (Java, C, C++) que nos obligan a ello
 - en algunos lenguajes (Java, C, C++), el rango tiene otra limitación: son números entre 0 y un valor positivo

Declaración de tablas

Para declarar una tabla es preciso indicar

- su nombre
- el rango de valores del índice
- el **tipo** de elemento de cada casilla de la tabla

Sintaxis

tipo [1..N] nombre;

entero [1..50] a;	Declara una tabla de 50 variables de tipo entero
String [0..29] listaNom;	Declara una tabla de 30 objetos de tipo String
Alumno [0..19] alumno;	Declara una tabla de 20 objetos de la clase Alumno

Uso de los elementos de la tabla

- **tabla[i]** es el nombre del elemento *i*-ésimo de la tabla


```
a[i] // elemento i de la tabla a (es un entero)
listaNom[3] // cuarto elemento de la tabla
// listaNom (es un String)
alumno[19] // ultimo elemento de la tabla
// alumno (es un objeto de Alumno)
```
- estos elementos se pueden usar como variables
 - en expresiones


```
j := j*a[3]+4*a[7];
imprimir(listaNom[2]);
```
 - y para asignarle un valor


```
a[5] := 38;
listaNom[3] := "Esto es un texto";
```

Operaciones sobre tablas

Las tablas suelen disponer de una forma de obtener su tamaño

Si a es una tabla

- llamaremos $a.\text{longitud}$ al número de casillas de a

Algunas de las operaciones más comunes que haremos en las tablas son las mismas que en la secuencia

- recorrido
- búsqueda

El acceso directo dará otras posibilidades (ordenación, ...)

En ocasiones tendremos que convertir una secuencia a una tabla, y viceversa

Recorrido de tablas

Esquema de recorrido: especificación

```
método recorrido (tipo[1..N] t)
  {Pre:}
  Recorrido
  {Post: tratados t[1]..t[N]}
fmétodo
```

Recorrido Iterativo (mientras)

```

esquema recorridoIter(tipo[1..N] t)
  {Pre:}
  var entero i:=1 fvar;
  inicializaciones;
  {Inv: tratados t[1]..t[i-1], 1<=i<=N+1}
  mientras i<N+1 hacer
    tratar t[i];
    i:=i+1;
  fmientras
  {Post: tratados t[1]..t[N]}
fesquema

```

Recorrido Iterativo (para)

```

esquema recorridoIter(tipo[1..N] t)
  {Pre:}
  inicializaciones;
  para i:=1 hasta N hacer
    {Inv: tratados t[1]..t[i-1], 1<=i<=N}
    tratar t[i];
  fpara
  {Post: tratados t[1]..t[N]}
fesquema

```

Recorrido Iterativo (para)

```
// cuando el índice empieza en cero
esquema recorridoIter(tipo[0..N-1] t)
  {Pre:}
  inicializaciones;
  para i:=0 hasta N-1 hacer
    {Inv: tratados t[0]..t[i-1], 0<=i<=N-1}
    tratar t[i];
  fpara
  {Post: tratados t[0]..t[N-1]}
fesquema
```

Recorrido recursivo

```
esquema recorridoRec(tipo[1..N] t, entero i)
  {Pre: tratados t[1]..t[i-1]}
  si i<N+1 entonces
    tratar t[i];
    recorridoRec(t,i+1)
  fsi
  {Post.- tratados t[1]..t[N]}
fesquema
```

Ejemplo de recorrido: Suma de elementos

```
método sumaTabla(entero[1..N] t) retorna entero
  {Pre:}
  suma
  {Post: valor retornado=t[1]+...+t[N]}
fmétodo
```

Solución Iterativa

Invariante: s contiene la suma de los elementos $t[1]..t[i-1]$,
 $1 \leq i \leq N+1$

Cota: $N+1-i$

Terminación: $i==N+1$

Inicializaciones:

```
i:=1;
s:=0;
```

Cuerpo del bucle:

```
s:=s+t[i];
i:=i+1;
```

Solución iterativa

```

método sumaTabla(entero[1..N] t) retorna entero
  {Pre:}
  var
    entero i:=1;
    entero s:=0;
  fvar
    {Inv: s es la suma de elementos t[1]..t[i-1],
      0<=i<=N-1}
  mientras i<N+1 hacer
    s:=s+t[i];
    i:=i+1;
  fmientras
  retorna s;
  {Post: valor retornado=t[1]+...+t[N]}
fmétodo

```

Máximo de una tabla

```

// con bucle para
método máximo(entero[1..N] t) retorna entero
  {Pre: N>=1 (la tabla tiene al menos un elemento)}
  var
    entero m:=t[1];
  fvar
    {Inv: m es el máximo de t[1]..t[i-1]}
  para i:=2 hasta N hacer
    si t[i]>m entonces
      m:=t[i]
    fsi
  fpara
  retorna m;
  {Post: valor retornado= máximo de t[1],...,t[N]}
fmétodo

```

Búsqueda en tablas

Búsqueda de un elemento que satisface una propiedad **P**

```

esquema buscar (elem[1..N] t) retorna entero
  {Pre:}
  {Post:
    valor retornado= primer k,  $1 \leq k \leq N$ , tal que
      t[k] satisface P
    si ningún valor satisface P, valor retornado=-1
  }
fesquema
  
```

Solución iterativa

```

esquema buscar (elem[1..N] t) retorna entero
  var
    booleano encontrado:= falso;
    entero índice:=-1;
    entero i:=1;
  fvar
    {Inv: si encontrado, t[índice] cumple P
      si no encontrado, índice=-1 y ninguno de los
        elementos t[1]..t[i-1] cumple P
       $1 \leq i \leq N+1$ }
  
```

Solución iterativa (cont.)

```

mientras no encontrado y  $i < N+1$  hacer
  encontrado := P(t[i]);
  si encontrado entonces
    índice := i;
  fsi
  i := i+1;
fmientras;
retorna índice;
fesquema

```

Búsqueda binaria o dicotómica

```

método búsquedaBinaria(elem[1..N] t, elem x)
retorna entero
  {Pre: t ordenada crecientemente}
  {Post: valor retornado =
    -1 si x no está en la tabla
     $1 \leq i \leq N$  si  $x = t[i]$ }
fmétodo

```

Solución Iterativa

```
método búsquedaBinaria(elem[1..N] t, elem x)
retorna entero
var
    entero inicio:=1;
    entero fin:=N;
    entero medio;
fvar

// comprobar si la tabla esta vacía
si t.longitud==0 entonces
    retorna -1; // elemento no encontrado
fsi
{Inv: x > elementos anteriores a inicio,
     x < elementos posteriores a fin,
     1<=inicio<=fin<=N}
```

Solución Iterativa (cont.)

```
mientras inicio<fin hacer
    medio:=(inicio+fin) div 2;
    si t[medio]<x entonces
        inicio:=medio+1;
    si no
        fin:=medio;
fsi
fmientras;
si (t[inicio]==x) entonces
    retorna inicio;
si no
    retorna -1; // elemento no encontrado
fsi
fmétodo
```

Solución recursiva

```

método búsquedaBinaria(elem[1..N] t, elem x,
entero inicio, entero fin) retorna entero
  {Pre: t ordenada crecientemente,
    x > elementos anteriores a inicio,
    x < elementos posteriores a fin,
    1<=inicio<=fin<=N}
  var entero medio fvar
  si inicio<fin entonces
    // caso recursivo
    medio:= (inicio+fin) div 2;
    si t[medio]<x entonces
      retorna búsquedaBinaria(t,x,medio+1,fin);
    si no
      retorna búsquedaBinaria(t,x,inicio,medio)
  fsi

```

Solución recursiva (cont.)

```

  si no
    // caso directo
    si (t[inicio]==x) entonces
      retorna inicio;
    si no
      retorna -1; // elemento no encontrado
  fsi
fsi
fmétodo

```

Solución recursiva (cont.)

```
// método para hacer la búsqueda completa
método búsquedaBinaria(elem[1..N] t, elem x)
retorna entero
  {Pre: t ordenada crecientemente}
  // comprobar si la tabla esta vacía
  si t.longitud==0 entonces
    retorna -1; // elemento no encontrado
  fsi
  retorna búsquedaBinaria(t,x,1,t.longitud);
fmétodo
```

Crear una secuencia a partir de una tabla: solución iterativa

```
método tablaASecuencia(elem[1..N] t)
retorna Secuencia<elem>;
  {Pre:}
  var
    entero i:=1;
    Secuencia<elem> s= nueva secuencia vacía;
  fvar
    {Inv: Escritos t[1],...,t[i-1], 1<=i<=N+1}
    mientras i<N+1 hacer
      s.escribirAlFinal(t[i]);
      i:=i+1;
    fmientras;
  retorna s;
  {Post: s=[T[1],T[2],...,T[N]]}
fmétodo
```

Ordenación en tablas

Es frecuente tener la necesidad de ordenar los elementos almacenados en una tabla

- por ejemplo, para hacer búsquedas eficientes

Es preciso que los elementos tengan una relación de orden definida

Existen diversos algoritmos de ordenación

- se distinguen por su dificultad, por su eficiencia, y por sus necesidades de memoria
- la eficiencia se suele medir como una función aproximada del número de datos a ordenar (n)

Principales algoritmos de ordenación

Algoritmo	Eficiencia promedio
Ascenso de burbuja	n^2
Ordenación por inserción	n^2
Ordenación por selección	n^2
Ordenación rápida (quick sort)	$n \log n$
Ordenación por cajas (bin sort)	$n+n^0$ cajas $\rightarrow n$ según el caso
Ordenación por cifras (key sort)	$n*n^0$ cifras $\rightarrow n$ según el caso

Los primeros no se utilizan, por poco eficientes

Los dos últimos pueden ocupar mucha memoria

Una buena solución es la ordenación rápida

Ordenación por selección

```
método ordenaSelección(elem[1..N] t)
  {Pre: t=T}
  {Post: t está ordenada crecientemente,
        t es una permutación de T}
fmétodo
```

Solución iterativa

```
método ordenaSelección(elem[1..N] t)
  {Pre: t=T}
  var
    entero i:=1; entero pmin;
  fvar
    {Inv: t[1..i-1] está ordenada crecientemente
        t[1..N] es una permutación de T,
        1<=i<=N, t[i-1]<= min{t[k], k>=i}}
  mientras i<N hacer
    pmin:=posMin(t,i,N);
    //{t[pmin]=min{t[k], k>=i}
    intercambiar(t[i], t[pmin]);
    i:=i+1;
  fmientras
fmétodo
```

Solución iterativa (cont.)

```

método posMin(elem[1..N] t, entero i, entero j)
retorna entero
  {Pre:  $1 \leq i \leq j \leq N$ }
  var
    elem m:=t[i]; entero indice:=i;
  fvar
    {Inv t[m]=min{t[f]:  $i \leq f < k$ }
    para k:=i+1 hasta j hacer
      si t[k]<m entonces
        m:= t[k]; indice:=k;
      fsi
    fpara
    retorna indice;
  {Post: t[valor retornado]=min{t[f]:  $i \leq f \leq j$ }
fmétodo

```

Ordenación por inserción

```

método ordenaInserción(elem[1..N] t)
  {Pre:  $N \geq 0$ ,  $t = T$ }
  {Inv: t[1..i-1] ordenada crecientemente,
    t[1..i] es una permutación de T[1..i],
     $2 \leq i \leq N+1$ }
  para i:=2 hasta N hacer
    insertar(t,i);
  fpara
  {Post: t ordenada crecientemente y
    es una permutación de T}
fmétodo

```

Ordenación por inserción (cont.)

```

método insertar(elem[1..N] t, entero i)
  {Pre: t[1..i-1] ordenada creciente y t =T, 1<=i<=N}
  var
    elem x:=t[i];
    entero j:=i;
  fvar
  mientras j>1 y x<t[j-1] hacer
    t[j]:= t[j-1];
    j:=j-1;
  fmientras;
  t[j]:=x;
  {Post: t[1..i] ordenada creciente,
    t[1..i] permutación de T[1..i],
    t permutación de T}

fmétodo

```