PROGRAMACIÓN

Escuela de Informática. Curso 2001/2002

Tema 5. VECTORES. RECORRIDO Y BÚSQUEDA

Isabel Galiano, Francisco Marqués y Natividad Prieto

Índice General

1 Introducción **2**

2 Vectores unidimensionales **2**
2.1 Declaración de vectores 2
2.2 Acceso a las componentes 4
2.3 Ejemplo 4

3 Vectores multidimensionales **6**
3.1 Declaración de vectores multidimensionales 6
3.2 Acceso a las componentes 7
3.3 Ejemplo 7

4 Problemas de recorrido y búsqueda en vectores **9**
4.1 Problemas de recorrido 9
4.2 Problemas de búsqueda 10
4.3 Búsqueda con garantía de éxito, centinelas 11
4.4 Problemas combinados 12

5 Complejidad de los algoritmos de recorrido y búsqueda **15**
5.1 Comparación de los costes de los algoritmos 15
5.2 Uso de la notación asintótica 16
5.3 Análisis por casos: caso mejor y caso peor 18
5.4 Regla general para el estudio de costes 19
5.5 Eficiencia de los algoritmos de recorrido 19
5.6 Eficiencia del algoritmo de búsqueda secuencial 19

6 Problemas propuestos **20**

1 Introducción

En ocasiones es conveniente poder almacenar y referenciar variables que representan una colección de valores, de forma que sea posible tratar de forma uniforme cada uno de los valores elementales que forman la colección.

Imagínese, por ejemplo, que una estación metereológica dispone de las medidas diarias de la pluviosidad en una determinada zona a lo largo de un mes. A partir de dichas medidas la estación debe generar un informe indicando los días de máxima y mínima pluviosidad, el total llovido a lo largo de todo el mes, la pluviosidad media de cada día, listar los días en que la pluviosidad duplicó al valor medio y, por último, listar de forma ordenada, y ascendentemente, las distintas medidas y los días en que éstas se produjeron.

Para poder efectuar cada uno de los procesamientos indicados, es necesario poder tratar de forma individual cada una de las medidas; pero haciendo uso de algún tipo de notación que soslaye el problema de tener que escribir repetidamente los mismos cálculos con nombres de variables prácticamente idénticos.

Las matemáticas resuelven el problema anterior con el uso de índices. Así, si se considerara la variable *lluvia*, por ejemplo, como la colección de valores de la pluviosidad a lo largo de 30 días, entonces se podría referenciar cada uno de los valores individuales mediante un subíndice que indicara el día en que se produce dicha medida, así: $lluvia_1$, $lluvia_2$, ..., $lluvia_{20}$, harían referencia, respectivamente, a la pluviosidad registrada el primer, segundo y vigésimo día del mes. Igualmente, $lluvia_i$, representaría la cantidad de lluvia registrada el día i-ésimo, siendo i un valor entero, número de día, comprendido entre 1 y 31.

En el lenguaje java (así como en la mayoría de los lenguajes de programación) se introduce el concepto de vector, o colección de valores del mismo tipo, con nombre común, y referenciables uno a uno, mediante un índice. En el resto del tema se examinará cómo definir y utilizar adecuadamente vectores en java.

2 Vectores unidimensionales

Como ya se ha visto, un **vector** es una colección de componentes *homogénea*, esto es, todos del mismo tipo de datos, que tiene un nombre o identificador común. Las características principales de una variable vector son:

- Es posible acceder y modificar cada una de las componentes individuales por su posición dentro del grupo o colección
- El número de componentes de un vector se establece inicialmente, no siendo posible su modificación posterior.

2.1 Declaración de vectores

Mediante la declaración de un vector se define, en primer lugar, el tipo de cada una de sus componentes. Por ejemplo:

```
double lluvia[];  // lluvia es un vector de valores "double"
```

Obsérvese que mediante la declaración anterior no se ha establecido aún el número de componentes del vector. Igual que en el resto de declaraciones de variable, existe un valor por omisión para la variable `lluvia` llamado `null`, que representa un vector sin ningún valor.

Para establecer el número de elementos del vector hace falta, en el lenguaje java, una operación explícita que estableza inicialmente dicho número, lo que se consigue haciendo uso de un operador especial, `new`, que permite asignar espacio inicialmente al vector. El operador `new` recibe el número y tipo de los elementos del vector.

Por ejemplo, mediante la instrucción siguiente, que se ejecutaría después de la declaración ya vista, se establece que el número de elementos del vector `lluvia` sea 31, reservando con ello el compilador espacio consecutivo para 31 componentes individuales de tipo `double`.

```
lluvia = new double[31];
```

Como se ha dicho, entre las dos instrucciones mostradas (la de declaración, y la de creación de espacio inicial) el valor de la variable pluviosidad es `null`; como es habitual en el lenguaje, es posible aunar ambas instrucciones en tan solo una:

```
double lluvia[] = new double[31];
// lluvia es un vector de 31 valores "double"
```

En general, una declaración de tipo vector suele ser de la forma:

```
tipoBase vector[] = new tipoBase[num];
// "vector" es un vector de "num" valores de "tipoBase"
// los valores individuales se inicializan adecuadamente
```

Una última característica significativa que cabe reseñar es que el valor `num`, mediante el cual se define el número de elementos del vector en consideración puede ser una expresión numérica. Con ello es posible parametrizar el tamaño de los vectores como función de alguna característica del problema.

Así, por ejemplo, se podría efectuar la siguiente declaración:

```
...
int numDías;
// la variable numDias contiene el número de días del mes
...
double lluvia[] = new double[numDías];
```

2.2 Acceso a las componentes

Tras efectuar la declaración del vector `lluvia` se dispone de 31 componentes de tipo `double` numeradas desde la 0 hasta la 30 y accesibles mediante la notación: `lluvia[0]` (componente primera), `lluvia[1]` componente segunda, y así sucesivamente hasta la última componente, que se denota: `lluvia[30]`.

Con cada una de las componentes del vector `lluvia` es posible efectuar todas las operaciones que podrían realizarse con variables individuales de tipo `double`, por ejemplo, dadas las declaraciones anteriores se podría hacer:

```
lluvia[3] = 35.167;
System.out.println("Pluviosidad medida: " + lluvia[3]);
lluvia[5] = lluvia[3]*2;
int i = 6;
lluvia[i+1] = lluvia[i-1];
```

En java existe, además, la posibilidad de inicializar simultáneamente todas las componentes de un vector, en lugar de hacerlo individualmente componente a componente. Sea, por ejemplo, el segmento siguiente:

```
double ejemplo[] = new double[5];
// inicialización
ejemplo[0] = 1.3; ejemplo[1] = 2.678; ejemplo[2] = 124.4;
ejemplo[3] = 21.367; ejemplo[4] = 34.8; ejemplo[5] = 24.6;
```

es posible utilizar, para realizar las asignaciones iniciales anteriores, un *inicializador de vector*, esto es: una lista de expresiones separadas por comas y encerradas entre llaves; así:

```
int i = 1;
double ejemplo[] = {1.3*i, 2.678*i, 124.4*i, 21.367*i, 34.8*i, 24.6*i};
```

En general, si se considera la declaración de caracter general del tipo `vector`, efectuada antes, la expresión `vector[expInt]`, hace referencia al elemento de número `expInt` del vector, siendo `expInt` una expresión numérica entera cualquiera. La expresión es correcta si `expInt` tiene, al ser evaluada, un valor válido, esto es: comprendido entre 0 y `num-1`; de no ser así se dará o un error durante la compilación, o una excepción `ArrayIndexOutOfBoundsException`, durante la ejecución del programa.

2.3 Ejemplo

El programa que se muestra a continuación lee los datos de pluviosidad almacenados en cierto fichero `"plfeb.dat"` para, calcular su valor medio y, a continuación, escribir los días del mes en que la pluviosidad fue, al menos, el doble de la media:

```
import nsIO.*;
class lluviasM {

  static final int MAX_DIAS = 32;

  public static void main(String args[]){
  // definición de entrada y salida ...
  input fent = new input("plfeb.dat");
  output fsal = new output();

  // definición del vector lluvia
  double lluvia[] = new double[MAX_DIAS];

  // lectura y acumulación
  int nD = 1;
  double media = 0;
  while (fent.more() && nD < MAX_DIAS) {
    lluvia[nD] = fent.readdouble();
    media += lluvia[nD];
    nD++;
    }
  nD--;

  // escritura de la media
  fsal.writeln("Numero total dias: "+nD);
  if (nD>0) {
     media /= nD;
     fsal.writeln("Pluviosidad media: "+media+" l.\n");
  }

  // valores mayores que el doble
  double doble = 2*media;
  for (int i = 1; i <=nD; i++)
    if (lluvia[i]>doble)
      fsal.writeln("Dia: \t"+i+"\t"+lluvia[i]+" l.");

  // cerrar ficheros
  fent.close();
  fsal.close();
  }
}
```

3 Vectores multidimensionales

Si se deseara extender el tratamiento del ejemplo anterior, en el tratamiento de la pluviosidad de una zona, para abarcar no sólo los días de un mes sino los de todo un año, se podría definir, por ejemplo, un vector de 366 elementos, que mantuviera de forma correlativa los datos de pluviosidad de una zona día a día. Con ello, por ejemplo, el dato correspondiente al día 3 de febrero ocuparía la posición 34 del vector, mientras que el correspondiente al 2 de julio ocuparía el 184. Otra aproximación para la representación de los datos del problema anterior consistiría en utilizar un vector de 12 componentes, vectores a su vez, cada uno con 31 elementos. Lo que permitiría una descripción más ajustada a la realidad y, sobre todo, simplificaría los cálculos de la posición real de cada día en la estructura de datos.

En java las componentes de un vector pueden ser, a su vez, otros vectores. Las componentes de estos últimos vectores pueden ser tanto datos elementales, con lo que los vectores son *bi-dimensionales*, (también denominados *matrices*), o pueden ser vectores de nuevo. En general, no hay límite al anidamiento que puede presentar una estructura de ese tipo, con lo que se puede obtener vectores de tantas dimensiones como se desee (denominados *n-dimensionales*).

3.1 Declaración de vectores multidimensionales

El código siguiente declara en java una matriz (vector bi-dimensional) de elementos de tipo double, y la inicializa para que sea de 4 por 4 elementos:

```
double matriz [][] = new double[4][4];
```

La siguiente declaración es equivalente a la anterior (se define matriz como un vector de vectores de tipo `double`), aunque en la práctica es menos utilizada:

```
double [] matriz [] = new double[4][4];
```

En realidad los vectores multidimensionales en java son vectores de vectores, por lo que es posible declarar al comienzo tan solo una dimensión e inicializar posteriormente el resto. Conisidérese, por ejemplo, esta otra forma de declaración para el ejemplo anterior en su primera versión:

```
double matriz [][] = new double[4][];
// se inicializa matriz a un vector de 4  componentes,
// vectores, a su vez, aún no inicializados. (Valen null)
matriz[0] = new double[4];
matriz[1] = new double[4];
matriz[2] = new double[4];
matriz[3] = new double[4];
```

Obviamente, es posible inicializar cada uno de los subvectores con un tamaño diferente (aunque el tipo base elemental debe ser siempre el mismo para todos los componentes). Considérese el ejemplo siguiente:

```
double matriz [][] = new double[4][];
// se inicializa matriz a un vector de 4  componentes,
matriz[0] = new double[3];  // matriz[0] tiene 3 elem.
matriz[1] = new double[4];  // matriz[1] tiene 4 elem.
matriz[2] = new double[14];  // matriz[2] tiene 14 elem.
matriz[3] = new double[10];  // matriz[3] tiene 10 elem.
```

Como ocurría en la declaración de los vectores unidimensionales, el número de elementos con que se inicializa un vector o subvector cualquiera en el caso multidimensional puede ser una expresión numérica entera con valor, en el momento de la creación, mayor o igual que 0.

3.2 Acceso a las componentes

El acceso a las componentes de un vector multidimensional se efectúa de forma similar al caso unidimensional, pero tomando en consideración la existencia de múltiples dimensiones. Además, el índice de todos los subvectores se encuentra comprendido entre 0 y la dimensión del mismo menos 1. Por ejemplo:

```
double matriz [][] = new double[4][4];

for (int i=0; i<4; i++) matriz[i][i] = 1;
// los elementos de la diagonal principal valen 1
```

Como ocurría con los vectores unidimensionales, a los n-dimensionales se les puede asignar también un valor mediante el uso de un *inicializador de vector*, como, por ejemplo, se realiza a continuación:

```
// definición e inicialización de "matriz"
double matriz[][] = {{0,1,2},{10,11,12},{20,21,22,23},{33,11}};
```

3.3 Ejemplo

El programa que se muestra a continuación lee los datos de pluviosidad almacenados en cierto fichero `"pluvio.dat"`. Los datos en el fichero se han almacenado escribiendo en cada linea del mismo el dia de la medida, el mes, y la medida total. Las medidas no tienen por que estar ordenadas cronológicamente. Un ejemplo de algunas lineas del fichero de entrada es el siguiente:

```
...
24 11 312.12
15  3   6.756
14  8  12.5
15  1  31.3
16  3 212.0
```

```
17  3  87.9
18  3   3.56
23  6  11.11
...
```

A partir del fichero de entrada anterior se desea generar un nuevo fichero, denominado `"pluvio2.dat"` que tenga las mismas medidas que aparecen en el fichero de entrada pero ordenadas cronológicamente. De forma resumida, el programa lee del fichero de entrada en una matriz y escribe, recorriendo la matriz, en el fichero de salida. Nótese como se utiliza un vector especial `diasM[]` para almacenar el número total de días de cada mes.

```
import nsIO.*;
class lluvias {
  public static void main (String Args[]){

     input fe = new input("pluvio.dat");
     output fs = new output("pluvio2.dat");

     // matriz de días del mes; el mes 0 tiene 0 días
     int diasM[] = {0,31,29,31,30,31,30,31,31,30,31,30,31};

     // matriz de datos, declaración
     double lluvia[][] = new double[13][];

     // matriz de datos, inicialización (notar que lluvia[0]=null)
     for (int i=1; i<=12; i++) lluvia[i] = new double[diasM[i]+1];

     // lectura de datos:
     while (fe.more()) {
        int dia = fe.readint();
        int mes = fe.readint();
        double plu = fe.readdouble();

        if (1<=mes && mes<=12 && 1<=dia && dia<=diasM[mes])
           lluvia[mes][dia] = plu;
     }
     // escritura de los valores de la matriz distintos de 0
     for (int mes=1; mes<=12; mes++)
       for (int dia=1; dia<=diasM[mes]; dia++)
          if (lluvia[mes][dia]!=0)
             fs.writeln(dia+"\t"+mes+"\t"+lluvia[mes][dia]);
     // cerrar ficheros
     fe.close();
     fs.close();
```

```
   }
}
```

4 Problemas de recorrido y búsqueda en vectores

Muchos de los problemas que se plantean cuando se utilizan vectores pueden clasificarse en dos grandes grupos de problemas genéricos: los que conllevan el *recorrido de un vector* y los que suponen la *búsqueda de un elemento*, de entre los del vector, que cumple cierta característica dada. La importancia de este tipo de problemas proviene de que surgen no sólo en el ámbito de los vectores sino también en muchas otras organizaciones de datos de uso frecuente. Las estrategias básicas de resolución que se verán a continuación son también extrapolables a esos otros ámbitos.

4.1 Problemas de recorrido

Se clasifican como *problemas de recorrido* a todos aquellos que para su resolución exigen algún tratamiento de todos los elementos del vector. El orden para el tratamiento de los elementos del vector puede organizarse de muchas maneras: ascendentemente, descendentemente, simultáneamente ascendente y descendente, etc.

En el siguiente ejemplo se muestra un segmento de código en java para determinar, a partir de una definición de datos similar a la del último ejemplo resuelto, la cantidad máxima llovida en un sólo día a lo largo del mes de noviembre (mes 11), así como el día en que ésta se produjo:

```
....
// la matriz "lluvia" ya ha sido leída del fichero y
// contiene los datos del mismo
....
int maxD = 1;
for (int i = 2; i<= diasM[11]; i++)
   if (lluvia[11][i] > lluvia[11][maxD]) maxD = i;
// maxD es el día de máxima pluviosidad
System.out.println("Día: "+maxD+" lluvia:"+lluvia[11][maxD]);
....
```

Naturalmente, puede organizarse el cálculo anterior mediante un recorrido de los elementos del vector en sentido descendente, en lugar de ascendente, así el segmento siguiente es similar al anterior

```
....
// la matriz "lluvia" ya ha sido leída del fichero y
// contiene los datos del mismo
....
int maxD = 0;
```

```
double máximo = 0;
for (int i = diasM[11]; i>= 1; i--)
   if (lluvia[11][i] > máximo) {
           máximo = lluvia[11][i];
           maxD = i;
   }
// maxD es el día de máxima pluviosidad
System.out.println("Día: "+maxD+" lluvia:"+lluvia[11][maxD]);
....
```

Nótese que, además de una diferente inicalización en los dos segmentos de código, se ha seguido una estrategia diferente para dar valor inicial al máximo inicial.

4.2 Problemas de búsqueda

Se denominan problemas de búsqueda a los que, de alguna manera, conllevan el determinar si existe algún elemento del vector que cumpla una propiedad dada. Con respecto a los problemas de recorrido presentan la diferencia de que no es siempre necesario tratar todos los elementos del vector, ya que el elemento buscado puede encontrarse inmediatamente, encontrarse tras haber recorrido todo el vector, o incluso no encontrarse.

Consíderese el problema siguiente: "Determinar, a partir de la matriz `lluvia` un día del mes de marzo en que la pluviosidad haya superado los 100 litros".

El siguiente segmento de código, posible solución del problema, determina el día pedido si es que éste existe o señala el hecho de que no exista:

```
....
// la matriz "lluvia" ya ha sido leída del fichero y
// contiene los datos del mismo
....
int i;
for (i=1; i<=diasM[3] && lluvia[3][i]<=100; i++);
// o bien i > diasM[3] o bien lluvia[3][i] > 100
if (i<=diasM[3]) System.out.println("día: "+i);
   else System.out.println("el día no existe");
....
```

Nótese que el orden de evaluación de la condición de la iteración, así como el uso de la conjunción en cortocircuito son relevantes, de ser diferentes el segmento podría originar un error de intento de acceso a un elemento inexistente (fuera de rango).

Otra posible solución del problema es la siguiente:

```
....
// la matriz "lluvia" ya ha sido leída del fichero y
// contiene los datos del mismo
....
```

```
for (i=1; i<diasM[3] && lluvia[3][i]<=100; i++);
// o bien i = diasM[3] o bien lluvia[3][i] > 100
if (lluvia[3][i]>100) System.out.println("día: "+i);
   else System.out.println("el día no existe");
....
```

Nótese que en este segmento de código tanto el orden de evaluación de la condición como el tipo de operador conjunción utilizado es irrelevante ya que nunca se referencia una posición fuera de rango.

Naturalmente, cualquiera de las dos iteraciones anteriores podría efectuarse descendentemente. Proponemos al lector que repita los dos ejemplos anteriores mediante un tratamiento descendente.

Una estrategia seguida en ocasiones consiste en utilizar una variable booleana que indique en todo momento si la propiedad buscada ha sido ya establecida o no. Considérese, por ejemplo, el problema: "Determinar si en el mes de marzo hubo al menos tres días consecutivos con una pluviosidad mayor a 100 litros cada uno de ellos". El siguiente segmento es una de las soluciones posibles del mismo:

```
....
// la matriz "lluvia" ya ha sido leída del fichero y
// contiene los datos del mismo
....
boolean tresD = false;  // se han encontrado o no los tres días
int conta = 0;                  // num. días consecutivos con lluvia
int i = 1;                          // variable del bucle

while (i<=diasM[3] && !tresD) {
     if (lluvia[3][i] > 100) conta ++;
        else conta = 0;
     tresD = conta == 3;
     if (!tresD) i++;
}
// o bien i > diasM[3], o bien tresD == true
if (tresD) System.out.println(i-2+" "+i-1+" "+i);
   else System.out.println("No existen");
....
```

4.3 Búsqueda con garantía de éxito, centinelas

Ocasionalmente, se conoce de antemano que el elemento que se desea encontrar existe seguro en el vector (aunque tal vez no se sabe dónde se encuentra exactamente). Este tipo de búsquedas se denominan, por ello, *con garantía de éxito*. Cuando se dan, presentan la ventaja de que se puede simplificar en las mismas la condición de la iteración, reduciéndola generalmente en una conjunción. Con ello se consigue un menor tiempo de ejecución del bucle.

Muchas veces es posible "forzar"la existencia de un elemento como el buscado, utilizando una posición especial del vector al principio o al final del mismo, de forma que la búsqueda se realice con *garantía de éxito*. Ese tipo de elementos, que se insertan deliberadamente en el vector se denominan *centinelas*.

Supóngase, por ejemplo, que se desea determinar si existe algún día del tercer mes que presente una pluviosidad en el rango comprendido entre 50 y 100 litros. Como todos los vectores `lluvia[i]` tienen la posición 0 no utilizada, puede hacerse uso de la misma para incluir inicialmente en la misma el valor 75 litros que, en este caso, actuará como un centinela, así:

```
....
// la matriz lluvia ya ha sido leída del fichero y
// contiene los datos del mismo
....
int i;
lluvia[3][0] = 75;  // valor centinela
for (i=diasM[3]; lluvia[3][i]<50 || lluvia[3][i]>100; i--);
// i == 0 se ha encontrado solo el centinela, ...
if (i != 0) System.out.println("día: "+i);
   else System.out.println("el día no existe");
....
```

4.4 Problemas combinados

La solución de problemas reales exige a veces la realización combinada de técnicas de recorrido y búsqueda; mientras que en otras ocasiones se hace necesaria la obtención de resultados intermedios, mediante un tratamiento preliminar, mediante los cuales se progresa hacia la solución del problema.

En lo que sigue se muestran dos ejemplos que representan ambas situaciones; para resolverlos se utilizará un fichero ("Alumnos.dat") con los nombres de los alumnos matriculados en el curso. Cada linea de dicho fichero, correspondiente a un alumno, contiene en mayúsculas los apellidos y el nombre del mismo.

Problema 1.- Dado el fichero "Alumnos.dat"determinar, si existen, nombres duplicados, esto es: determinar todas las parejas de alumnos que tengan exactamente el mismo nombre.

Para evitar lecturas innecesarias del fichero se introducirán los datos inicialmente en un vector que será recorrido posteriormente para averiguar la existencia de duplicados.

Dicho recorrido posterior consistirá en una iteración de búsqueda, anidada dentro de otra de recorrido. Mediante la iteración externa se recorrerán, uno a uno, todos los alumnos, mientras que en la interna se buscará un nombre igual al que se esté tratando en ese momento en la iteración externa:

```
import nsIO.*;
```

```
class dupAlum {

  static final int MAX_ALUM = 200;  // máximo número alumnos

  public static void main(String args[]) {
    // definición de entrada y salida ...
    input fent = new input("Alumnos.dat");
    output fsal = new output();

    //definición del vector alum
    String alum[] = new String[MAX_ALUM];

    //lectura y creación:
    int numA = 0;
    while (fent.more() && numA < MAX_ALUM) {
      alum[numA] = fent.readline();
      numA++;
    }
    // hay un total de numA alumnos, pero
    // el último está en numA-1

    for (int i = 0; i<numA-1; i++) {
       int j;
       for (j = i+1; j<numA && !alum[i].equals(alum[j]); j++);
       // O bien j = numA y no está duplicado,
       // o bien j < numA y entonces alum[i] y alum[j] son iguales
       if (j < numA) { fsal.writeln(alum[i]);
                       fsal.writeln("duplicado en: "+i+" y "+j);
       }
    }
  }
}
```

Si este programa se ejecuta con la entrada:

```
MARQUES HERNANDEZ, FRANCISCO
PRIETO SANZ, NATIVIDAD
CASANOVA FAUS, ASSUMPCIO
ROSSO, PAOLO
LOPEZ PEREZ, JUAN
PRIETO SANZ, NATIVIDAD
PRIETO SANZ, NATIVIDAD
FACTOR JIMENEZ, NICOLAS
```

se obtiene la salida:

```
PRIETO SANZ, NATIVIDAD
duplicado en: 1 y 5
PRIETO SANZ, NATIVIDAD
duplicado en: 5 y 6
```

Considérese a continuación:

Problema 2.- Dado el fichero "Alumnos.dat"escribir de forma ordenada alfabéticamente los nombres de los alumnos junto con la posición que ocupa cada uno de ellos en el fichero inicial.

Para resolver este problema se puede seguir la estrategia siguiente:

1. introducir los datos del fichero en un vector que será tratado posteriormente
2. iterar tantas veces como alumnos y, en cada iteración
   (a) obtener el alumno alfabéticamente mayor (de los que quedan)
   (b) escribirlo
   (c) marcar el alumno para no considerarlo en iteraciones posteriores

El siguiente es un programa posible para resolver el problema siguiendo la estrategia anterior:

```
import nsIO.*;
class ordAlum  {
  static final int MAX_ALUM = 200;  // máximo número alumnos
  public static void main(String args[]) {
    // definición de entrada y salida ...
    input fent = new input("Alumnos.dat");
    output fsal = new output();

    // definición del vector alum
    String alum[] = new String[MAX_ALUM];

    // lectura y creación:
    int numA = 0;
    while (fent.more() && numA < MAX_ALUM) {
      alum[numA] = fent.readline();
      numA++;
    }
    // hay un total de numA alumnos, pero
    // el último está en numA-1

    // creación de un mínimo ficticio:
    alum[numA] = " ";
```

```
    // definición e inicialización del vector de marcados
    boolean marcas[] = new boolean[numA];
    for (int i=0; i<numA; i++) marcas[i]=false;

    // Iterar tantas veces como alumnos:
    for (int i=0; i<numA; i++) {
         int maxPos = numA;           // posición máximo
         // encontrar el máximo
         for (int j=0; j<numA;j++)
             if (!marcas[j] && alum[j].compareTo(alum[maxPos])>0)
                  maxPos = j;
         // máximo en maxPos, escribirlo y marcarlo, ...
         fsal.writeln(alum[maxPos]);
         marcas[maxPos]=true;
    }
  }
}
```

En este caso, ante una entrada similar a la utilizada en el ejemplo anterior, se obtiene la siguiente salida:

```
ROSSO, PAOLO
PRIETO SANZ, NATIVIDAD
PRIETO SANZ, NATIVIDAD
PRIETO SANZ, NATIVIDAD
MARQUES HERNANDEZ, FRANCISCO
LOPEZ PEREZ, JUAN
FACTOR JIMENEZ, NICOLAS
CASANOVA FAUS, ASSUMPCIO
```

5 Complejidad de los algoritmos de recorrido y búsqueda

5.1 Comparación de los costes de los algoritmos

Como ya se ha introducido en el tema anterior, el coste de un algoritmo se expresa como una función $T(n)$ no decreciente de la talla del problema. Por lo tanto comparar costes es comparar funciones no decrecientes de las que nos interesa sólo su tasa de crecimiento. Por ejemplo, si $T(n)$ es un polinomio, entonces el monomio de mayor grado del mismo es el que da el aspecto de la curva de crecimiento.

En la siguiente tabla se muestra en orden creciente de tasa de crecimiento distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos.

Función	Nombre
c	constante
$logn$	logarítmica
n	lineal
$nlogn$	nlogn
n^2	cuadrática
n^3	cúbica
2^n	exponencial

Supongamos que se está descargando un fichero de Internet, de forma que hay un retraso inicial de 2 seg. para establecer la conexión y después la descarga se realiza a razón de 1.6 K/seg. Si el tamaño del fichero es de N kbytes, el tiempo de descarga viene descrito por la fórmula lineal:

$$T(N) = \frac{N}{1.6} + 2$$

La descarga de un fichero de 80 k requiere 52 seg., la descarga de un fichero el doble de grande requiere del orden de 102 seg, es decir casi el doble. Esta propiedad, por la cual el tiempo de ejecución es esencialmente proporcional al tamaño de la entrada, caracteriza a un algoritmo lineal. Como puede verse en las gráficas que se presentan a continuación, las curvas no lineales conducen a tiempos de ejecución mayores. Esto indica que el coste lineal es, de entre todos los que se muestran en estas gráficas, el más favorable.

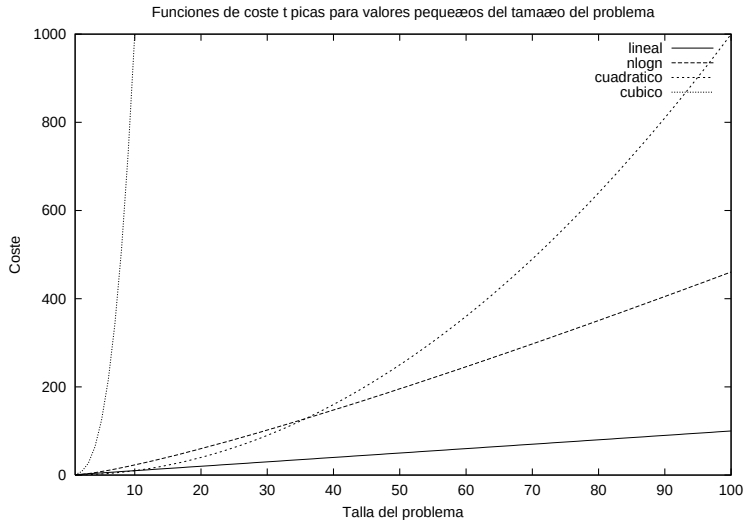
Una función cúbica es una función cuyo término dominante es n^3 multiplicado por alguna constante. Una función cuadrática tiene como término dominante n^2 por alguna constante, y una función lineal tiene como término dominante n por alguna constante. El término dominante de la función $nlogn$ es n veces el logaritmo de n; la función logaritmo crece muy lentamente, más lentamente que cualquier raíz.

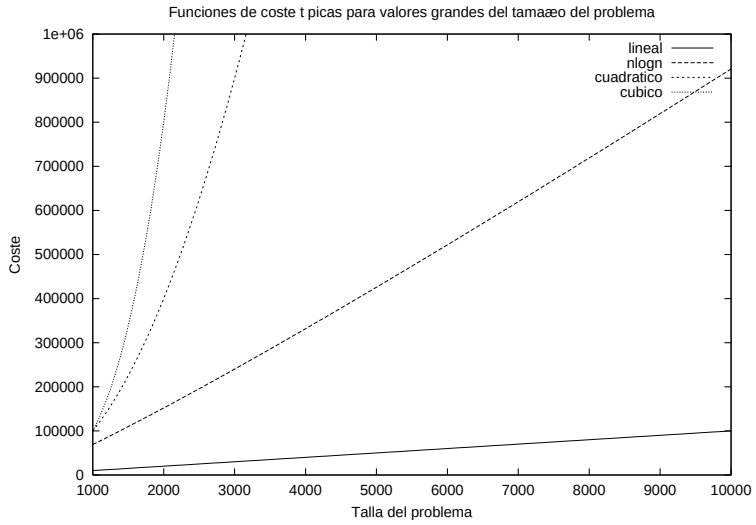
En las figuras siguientes se presentan cuatro funciones típicas en el análisis de algoritmos: lineal, $nlogn$, cuadrático y cúbico, sobre distintos tamaños del problema, la primera desde 1 hasta 100 y la segunda desde 1.000 hasta 10.000.

5.2 Uso de la notación asintótica

Cuando se analiza el coste de un algoritmo, el objetivo es conocer sobre qué curva de coste se encuentra dicho algoritmo. También es necesario poder comparar estas curvas de costes con el fin de poder decidir si un determinado algoritmo es mejor, peor o equivalente a otro. Es por esto que lo que nos interesa es medir el índice de crecimiento de las funciones de coste y expresarlo en notación asintótica. Tres razones apoyan esta decisión:

- Para valores de n suficientemente grandes el valor de la función está completamente determinado por el término dominante.
- El valor exacto del coeficiente del término dominante no se conserva al cambiar de entorno de programación.


Funciones de coste t picas para valores pequeæos del tamaæo del problema
lineal
nlogn
cuadratico
cubico
Coste
Talla del problema
0
200
400
600
800
1000
10
20
30
40
50
60
70
80
90
100


Funciones de coste t picas para valores grandes del tamaæo del problema
lineal
nlogn
cuadratico
cubico
Coste
Talla del problema
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1e+06
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

- El uso de la notación asintótica nos permite establecer un orden relativo entre funciones comparando términos dominantes:

$$logaritmico \prec lineal \prec nlogn \prec cuadratico \prec cubico \prec exponencial$$

5.3 Análisis por casos: caso mejor y caso peor

Como ya se ha comentado, el coste de un algoritmo es una función no decreciente de la talla del problema. Además, para un tamaño fijo del mismo, el coste del algoritmo puede depender de la configuración de la entrada del problema, de lo que llamaremos *instancia del problema.*

Una *instancia de un problema* representa todas las configuraciones diferentes de la entrada, de una talla determinada, para las que el comportamiento del algoritmo es el mismo en cuanto a costes.

Cuando en el coste de un algoritmo se detectan instancias, a partir de las instancias significativas se definen los siguientes conceptos:

- *Coste del algoritmo en el caso peor:* es la complejidad del mismo para la instancia del problema que presente el peor coste. Lo denotaremos por $T^p(n)$.
- *Coste del algoritmo en el caso mejor:* es la complejidad del mismo para la instancia del problema que presente el coste menor. Lo denotaremos por $T^m(n)$.
- *Coste promedio del algoritmo:* a la media de los costes de todas las instancias del problema.

Los estudios más útiles son el coste del algoritmo en el peor caso y el coste promedio. En general, el estudio del coste promedio de los algoritmos es difícil de realizar, tanto analítica como experimentalmente, la principal dificultad reside en conocer la distribución de probabilidad sobre las instancias del problema.

Ejemplos: Algoritmos de recorrido y búsqueda El problema de recorrido de un vector de n elementos presenta una única instancia; por lo tanto no se puede distinguir entre casos mejor y peor.

Sin embargo, el problema de la búsqueda secuencial presenta varias instancias:

- que el elemento se encuentre en la posición 0 del vector,
- que se encuentre en la posición 1,
- ...
- que se encuentre en la última posición $(n-1)$
- que no se encuentre

En total, tenemos $n+1$ posibilidades, las n primeras corresponden a casos de búsqueda con éxito, la última al caso de la búsqueda sin éxito. Si el algoritmo implementado realiza la búsqueda secuencial ascendente, el caso mejor, el más favorable, correspondería a la primera instancia (el elemento a buscar se encuentra en la primera posición). El caso peor se daría para la última instancia (búsqueda sin éxito).

5.4 Regla general para el estudio de costes

Para el estudio del coste temporal de los algoritmos se seguirán los siguientes pasos:

1. Determinar el tamaño del problema; esto es, estudiar de qué parámetros va a depender el coste.

2. Analizar si, para un tamaño del problema fijo, existen instancias significativas para el coste.

3. Obtener la función de coste. Si existen instancias significativas, el estudio de costes se particularizará para el caso peor y el caso mejor.

Una forma de obtener la función de coste de un determinado algoritmo iterativo consiste en contar el número de veces que se repite lo que llamaremos una *instrucción crítica* o *instrucción barómetro*. Una instrucción crítica en un determinado algoritmo es aquella que se ejecuta por lo menos con tanta frecuencia como cualquier otra del algoritmo. En realidad no hay problema si hay alguna instrucción que se ejecuta un número de veces constante más que el barómetro, ya que quedará absorbida en la notación asintótica.

5.5 Eficiencia de los algoritmos de recorrido

Sea `v` un vector definido de n elementos, el siguiente algoritmo describe el recorrido de dicho vector:

```
for (int i=0; i<n; i++) tratar(v[i]);
```

Supongamos que `tratar(e)` es una operación de coste constante que se realiza sobre el elemento `e`.

El coste del algoritmo será una función del número de elementos del vector $T(n)$. El algoritmo no presenta instancias significativas. La función de coste $T(n)$ se puede aproximar contando el número de veces que se repite la instrucción `tratar(v[i])`, que puede ser una instrucción crítica para este segmento de programa. También se podría haber tomado como instrucción crítica el incremento de la variable de control del bucle `i` o la evaluación de la guarda `i<n`, que incluso se realiza una vez más que el resto. Así el coste del algoritmo será: $T(n) = nLineal$

5.6 Eficiencia del algoritmo de búsqueda secuencial

Sea `v` un vector definido de n elementos, el siguiente algoritmo describe el proceso de búsqueda secuencial ascendente sin centinela. Se trata de buscar el primer elemento que cumpla una cierta propiedad `P(e)`.

```
int i=0;
while ((i<n) && !(P(v[i])))  i++;
if i<n return i; else return -1;
```

Supongamos que la comprobación de la condición de búsqueda `P(e)` es una operación de coste constante.

El coste del algoritmo será una función del número de elementos del vector en el que se efectúa la búsqueda $T(n)$. Para un vector de n elementos, el coste del algoritmo va a depender de la instancia del problema. El algoritmo presenta $n+1$ instancias significativas.

Análisis del caso peor El caso peor se da cuando el bucle se realiza el mayor número posible de veces `n`; esto ocurre cuando no hay ningún elemento en el mismo que satisfaga la propiedad enunciada. La función de coste $T^p(n)$ se puede aproximar contando el número de veces que se repite la instrucción crítica `i++` en este caso. También se podría haber tomado como instrucción crítica la evaluación de la guarda `((i<n) && !P(v[i]))`, que incluso se realiza una vez más que el resto. Así:

$$T^p(n) = n \quad LINEAL$$

Análisis del caso mejor El caso mejor se da cuando el bucle se realiza el menor número posible de veces `n`; esto ocurre cuando el primer elemento satisface la propiedad enunciada. Así:

$$T^m(n) = 1 \quad CONSTANTE$$

6 Problemas propuestos

1. Sea `v` un vector de `num` elementos de cierto tipo base `tbase`. Constrúyase un programa para invertir los elementos del vector, esto es: al finalizar la ejecución del mismo el vector contendrá en su posición 0 el elemento que inicialmente contenía la posición `num-1`, en su posición 1 el elemento que inicialmente contenía la posición `num-2` y así sucesivamente.

2. Dado el fichero `"pluvio.dat"`, cuyo contenido y organización es ya conocido, realícese un programa para: "Obtener el mes que ha llovido más, de forma acumulada, a lo largo de todo el año, así como la cantidad llovida en dicho mes."

3. A partir de la información del fichero "pluvio.dat"es posible subdividir los días del año, en función de la cantidad llovida, teniendo en cuenta los intervalos siguientes: [0..10[, [10..20[, ... [90..100[, ... [190 a 200[y, por último, más de 200 litros. Escríbase un programa que escriba en una lista cada intervalo junto con el número total de días del año en que la cantidad llovida pertenezca a dicho intervalo.

4. La moda de un conjunto de valores es el valor que se repite mayor número de veces en el conjunto. Escríbase un programa para leer desde cierto fichero (que puede ser la entrada estándard) los valores de las edades de un grupo de personas y que presente, a continuación, la moda de ese conjunto de valores (basta con escribir una si hay varias).

5. Escríbase un programa para determinar si un vector de palabras (`String`) es capi-cua, esto es, para determinar si la primera y última palabra del vector son la misma, la segunda y la penúltima palabras también lo son, y así sucesivamente.

6. Sea `v` un vector de valores enteros. Realícese un programa para determinar la posición, si existe, de la primera subsecuencia del vector que comprenda, al menos tres números enteros consecutivos en posiciones consecutivas del vector.

7. El algoritmo siguiente encuentra la posición del elemento más pequeño en un vector de n enteros ($n \geq 1$):

```
int pos, min;
pos=0; min=v[pos];
for (int i=1; i<n; i++)
   if v[i]<min { pos=i; min=v[pos];}
// en pos está el elemento más pequeño
```

   (a) ¿De qué va a depender el coste temporal de la función `posMinimo`?. Y por lo tanto, ¿cuál es el tamaño del problema?.

   (b) ¿Existen instancias significativas?

   (c) ¿Cuál podría ser la instrucción crítica en este algoritmo?. Cuántas veces se repite?.

   (d) ¿Cuál es la complejidad temporal de `posMinimo` expresada en términos del número de veces que se repite la instrucción crítica?. ¿Cuál es la complejidad asintótica de la función?.

8. Estúdiese la complejidad asintótica de los algoritmos iterativos obtenidos de la implementación directa (sin mejoras especiales) de las definiciones de las siguientes operaciones:

   (a) Producto escalar de dos vectores de dimensión n.

   (b) Suma de dos matrices cuadradas de dimensión nxn.

   (c) Producto de matrices cuadradas de dimensión nxn.

9. El siguiente algoritmo calcula la suma de las componentes de un vector de n naturales ($n \geq 1$):

```
int s=0;
for (int i=0; i<n; i++)
   for (int j=0; j<v[i]; j++)
     s++;
// s es la suma de los elementos del vector
```

   (a) ¿Se puede considerar como instrucción crítica de esta función la instrucción más interna de los dos bucles `s++`?. ¿Por qué?.

(b) ¿Cuál es la instrucción crítica para este algoritmo?.

(c) ¿Cuál es el coste de este algoritmo en términos del número de veces que se repite la instrucción crítica?. ¿Cuál es su complejidad asintótica?.

10. Considérese los dos algoritmos siguientes:

```
Algoritmo 1:
   for (int i=0; i<n; i++) {
      w[i]=0;
      for (int j=0; j<n; j++)
         w[i]+=v[j];
   }
Algoritmo 2:
   w[0]=v[0];
   for (int i=1; i<n; i++) {
      w[i]=w[i-1]+v[i];
   }
```

(a) ¿Qué hacen estos dos algoritmos?.

(b) Calcúlese la complejidad asintótica de estos dos algoritmos.

(c) ¿A qué se debe que las complejidades obtenidas no coincidan?.