7. Algoritmos de ordenación

Ordenación de vectores

En programación es muy habitual ordenar vectores:

- Para mostrar los datos en orden
- Para facilitar algunas operaciones con los datos, en particular, para mejorar el rendimiento de las búsquedas de datos
- ...

La idea de la ordenación es simple: dado un vector `v` (posiblemente con repeticiones)

	0	1	2	3	4	5	6	7	8	9
v	7	3	9	2	5	1	3	4	1	6

queremos ordenar sus elementos, por ejemplo en orden ascendente, de acuerdo a algún orden definido para ellos:

	0	1	2	3	4	5	6	7	8	9
v	1	1	2	3	3	4	5	6	7	9

El problema de la ordenación es de importancia capital en programación y se ha estudiado exhaustivamente.

- Se han propuesto multitud de métodos de ordenación, en particular, de ordenación de vectores

¿Qué métodos se nos ocurren?

Ordenación por inserción

Algoritmo simple y natural. Descripción:

- Para cada elemento e del array v
  - Se inserta e ordenadamente entre sus elementos anteriores

Animación: `https://en.wikipedia.org/wiki/Insertion_sort`

Características:

- simple de implementar
- *in-place*: se ejecuta sobre el propio array de entrada (sin arrays auxiliares)
- adaptativo: más eficiente cuanto más ordenado esté el array de partida
- estable: respeta el orden de los elementos iguales

Insertion sort: Implementación en C#

```
static void insercion(int [] v){
  int n = v.Length; // número de eltos del vector

  // v[0] ya está ordenado, para cada uno desde 1 hasta n:
  for (int i=1; i<n; i++){    // inv: v[0..i-1] está ordenado
    // insertamos v[i] ordenadamente en el subvector v[0..i-1]
    int tmp = v[i];           // guardamos v[i]
    int j = i-1;
    // desplazamos eltos a la dcha abriendo hueco para v[i]
    while ((j>=0) && (v[j]>tmp)) {
      v[j+1]=v[j];
      j--;
    }
    v [j + 1] = tmp;
  }
}
```


Medidas de tiempo (cronómetro)

```
// v vector de prueba (aleatorio o escogido)

// copiamos/clonamos v en w para no perder el
// original (las ordenacions son "in place")
w = (int[]) v.Clone();

// tiempo de de inicio
TimeSpan start = new TimeSpan(DateTime.Now.Ticks);

<<método de ordenación>>

// tiempo de parada
TimeSpan end = new TimeSpan(DateTime.Now.Ticks);
// resta de tiempos
double elapsed = end.Subtract(start).TotalMilliseconds;
```


Ordenación por selección

También es un algoritmo de ordenación simple y natural. Descripción:

- Buscamos el elemento menor y lo colocamos en la posición 0
- Buscamos el menor de los restantes y lo ponemos en la posición 1
- Buscamos el menor de los restantes y lo ponemos en la posición 2
- ...

Animación: https://en.wikipedia.org/wiki/Selection_sort

Características:

- simple de implementar
- *in-place*: se ejecuta sobre el propio array de entrada (sin arrays auxiliares)
- es estable?: es fácil hacer que lo sea (cómo?)
- es adaptativo?: más eficiente cuanto más ordenados esté el array de partida NO

Selection sort: Implementación en C#

```
static void seleccion(int [] v){
  int n = v.Length; // num de eltos del vector

  // en cada posición i=0..n-1 ponemos el menor de v[i..n-1]
  // vamos hasta i=n-2 porque al final v[n-1] ya está en su sitio
  for (int i = 0; i < n-1; i++) {
    // inv: el subarray 0..i-1 está ordenado
    // buscamos el menor en v[i..n)
    int min = i;
    for (int j = i + 1; j < n; j++)
      if (v [j] < v [min]) // con menor estricto es estable
        min = j;
    // ponemos el menor en v[i] y v[i] en la pos del menor
    swap (ref v [i], ref v [min]);
  }
}
```

```
static void swap(ref int x, ref int y){
  int tmp = x;
  x = y;
  y = tmp;
}
```

Método de la burbuja: bubble sort

Se recorre el array de principio a fin, comparando los elementos dos a dos (contiguos) e intercambiándolos si no están en orden... y se repite el proceso hasta que el array esté completamente ordenado (Wikipedia).

También puede recorrerse de fin a principio ⇝ se van colocando los menores al principio del array:

```
static void burbuja(int [] v){
  int n = v.Length;

  // damos n vueltas, en cada una v[i] queda en su sitio
  for (int i=0; i<n; i++)
    // inv: v[0..i-1] ordenado (en su posición final)
    // recorremos con j=n-1 hasta i+1 comparando v[j-1] y v[j]
    for (int j=n-1; j>i; j--)
      if (v[j-1]>v[j])
        swap(ref v[j-1],ref v[j]);
}
```

¿Es estable, adaptativo? ¿Cómo funciona si el vector de partida ya está ordenado? ¿Puede mejorarse?

Burbuja mejorado

Mejora: controlar si ha habido intercambio en cada vuelta; si no lo hay es porque el vector ya está ordenado ⇝ parar.

```
static void burbujaMejorado(int [] v){
  int n = v.Length;

  bool cont=true; // control de parada
  int i=0;  // ahora el for es un while con "cont" en la condición de parada
  while (i<n-1 && cont) {
    cont = false; // control de intercambio, inicialmente false
    for (int j=n-1; j>i; j--)
      if (v[j-1]>v[j]) {
        swap(ref v[j-1],ref v[j]);
        // si hay intercambio cont a true
        cont = true;
      }
    i++;
  }
}
```

¿Cuánto mejora el algoritmo? ¿Mejora en cualquier caso?

Ordenación de otros tipos de datos

Hemos visto algoritmos de ordenación para arrays de enteros

- Adaptación inmediata para tipos simples como (float, char, string,...)
- Fácil adaptación para cualquier otro tipo de datos ordenado

Por ejemplo, podemos ordenar polinomios de acuerdo a los exponentes de sus monomios. Adaptando el método de inserción:

```
static void ordenaInsercion(Polinomio p){
  for (int i=1; i<p.tam; i++){
    Monomio tmp = p.mon[i]; // insertamos el monomio i-esimo ordenadamente
    int j = i-1;      // desplazamos eltos a la decha abriendo hueco para él
    while ((j>=0) && (p.mon[j].exp > tmp.exp)) {
      p.mon[j+1]=p.mon[j];
      j--;
    }
    p.mon [j + 1] = tmp; // lo insertamos
  }
}
```

¿Puede ahora mejorarse la suma de polinomios, utilizando polinomios ordenados?

Mezcla ordenada

Para la suma de polinomios, teniendo los monomios ordenados, puede aplicarse el algoritmo de mezcla ordenada de dos vectores *ordenados* (muy utilizado en programación)

Idea:

- Se recorren simultáneamente los dos vectores de principio a fin
- En cada vuelta se elige el elemento más pequeño entre los apuntados en sendos vectores y se añade en otro vector
- ... hasta terminar el recorrido en un de los dos vectores.
- Se añaden los elementos restantes del vector que no ha terminado de recorrerse.

Concretando

- recorremos los vectores: `p1.mon` y `p2.mon` de tamaños `p1.tam` y `p2.tam`. Utilizamos los índices `i1, i2`
- generamos el vector suma `p3.mon`. Utilizamos el índice `j`

```java
static Polinomio suma(Polinomio p1, Polinomio p2){
  Polinomio p3;
  p3.mon = new Monomio[N*2]; // para asegurar que quepa

  // i1 recorre p1.mon, i2 recorre p2.mon, i3 recorre p3.mon
  int i1 = 0, i2 = 0, j = 0;

  // recorremos los dos pols hasta que uno termine (o ambos)
  while (i1 < p1.tam && i2 < p2.tam) {
    // añadimos en p3 el monomio de menor exponente
    if (p1.mon[i1].exp < p2.mon[i2].exp) {
      p3.mon[j] = p1.mon[i1];
      i1++; j++;
    } else if (p2.mon[i2].exp < p1.mon[i1].exp) {
      p3.mon[j] = p2.mon[i2];
      i2++; j++;
    } else { // si tienen el mismo exponente sumamos coeficientes
      double coef = p1.mon[i1].coef + p2.mon[i2].coef;
      if (coef != 0) {  // si el coef es cero, no añadimos nada, sino, la suma
        p3.mon[j].coef = coef;
        p3.mon[j].exp = p1.mon[i1].exp;
        j++;
      }
      i1++; i2++;
    }
  } // fin while
  // ... contiúa
```



```java
  // al final añadimos los monomios que queden de p1 o p2
  // A lo sumo se ejecuta uno de estos dos whiles
  // si p1.mon no se ha terminado se añaden sus monomios restantes
  while (i1 < p1.tam) {
    p3.mon[j] = p1.mon[i1];
    i1++; j++;
  }
  // idem con p2.mon
  while (i2 < p2.tam) {
    p3.mon[j] = p2.mon[i2];
    i2++; j++;
  }
  // actualizamos el tamaño de p3 y acabamos
  p3.tam = j;
  return p3;
} // fin método
```

Este algoritmo de mezcla es aplicable a vectores de cualquier tipo (ordenado).

Búsqueda en arrays ordenados

Problema: determinar si un elemento aparece en un array (de enteros, para simplificar)

- Con el array desordenado $\rightsquigarrow$ búsqueda secuencial (de la primera a la última componente).
- Pero si el array está ordenado, podemos mejorar esa búsqueda

Búsqueda binaria (o dicotómica) de un elemento e en un array v:

- Calculamos la posición media med de v
  - Si $e = v[med]$, fin (encontrado)
  - Si $e < v[med]$ buscamos en la mitad de la izquierda
  - Si $e > v[med]$ buscamos en la mitad de la derecha

¿Cómo/cuándo termina el algoritmo?

En general buscamos un elemento `e` en un subvector de `v`, delimitado por dos índices `ini` y `fin`:

- Inicialmente `ini=0` y `fin=n-1`
- Posición media `med = (ini+fin)/2` (por qué?)
- Buscar a la izquierda $\rightsquigarrow$ `fin = med-1`
- Buscar a la derecha $\rightsquigarrow$ `ini = med+1`

¿Cuándo termina la búsqueda?

- cuando $e == v[med]$
- o el subvector de búsqueda sea vacío $\rightsquigarrow$ `ini>fin`

¿Este método es mejor que la búsqueda secuencial? ¿Por qué?

Implementación:

```
static bool busquedaBinaria(int [] v, int e){
  int ini = 0, fin = v.Length - 1;
  bool enc = false;

  while (ini<=fin && !enc){
    int med = (ini + fin) / 2;
    if (e < v [med])
      fin = med - 1;
    else if (e > v [med])
      ini = med + 1;
    else
      enc = true;
  }

  return enc;
}
```


Ejercicios:

- Inserción en un array ordenado
- Eliminación en un array ordenado
- Mezcla de arrays ordenados