Programación 1

Tema 12

Algoritmos con vectores

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Algoritmos con vectores

En este capítulo se presentan algunos de los algoritmos básicos para resolver problemas clave en el tratamiento de estructuras de datos indexadas. Estos algoritmos deben formar parte de los conocimientos básicos de programación de cualquier estudiante universitario de Ingeniería Informática. Debe conocerlos por su nombre, debe saber programarlos y debe saber utilizarlos adecuadamente.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Índice

- Algoritmos de **recorrido**
- Algoritmos de **búsqueda**
  - Secuencial
  - Binaria
- Algoritmos de **distribución**
- Algoritmos de **ordenación**
  - Por **selección**

Problemas de tratamiento de estructuras de datos indexadas

Estos son algunos de los problemas de tratamiento de información a los que se enfrenta con mayor frecuencia un programador cuando trabaja con estructuras de datos indexadas:

- **Problemas de recorrido**. Estos problemas requieren el tratamiento de todos los elementos de la estructura. Por ejemplo, mostrar por pantalla todos los elementos, copiarlos en otra estructura de datos, determinar cuántos elementos satisfacen una determinada propiedad *P*, determinar cuál de los elementos presenta un valor mínimo o máximo, etc. Para resolver estos problemas se requiere tratar todos y cada uno de los elementos de la estructura mediante un **algoritmo de recorrido**.
- **Problemas de búsqueda**. Estos problemas plantean la búsqueda de un elemento de la estructura que satisfaga una determinada propiedad *P*. Cuando haya varios elementos que la satisfagan, la localización de uno de ellos bastará para resolver el problema. Los problemas de búsqueda se resolverán mediante un **algoritmo de búsqueda binaria** o mediante un **algoritmo de búsqueda secuencial**, en función de que los elementos de la estructura estén ordenados respecto de la propiedad *P* o no lo estén.
- **Problemas de distribución de datos**. Son problemas que requieren la reorganización de los elementos de la estructura de forma que, en una parte de ella se sitúen los que satisfagan una propiedad *P* y en la parte opuesta los que no la satisfagan. Para resolver estos problemas se aplican **algoritmos de distribución**.
- **Problemas de ordenación de datos**. Estos problemas requieren la reorganización de los elementos de la estructura de forma que queden ordenados según un determinado criterio. Para resolver estos problemas se aplican **algoritmos de ordenación**.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Problemas de tratamiento de estructuras de datos indexadas

En este capítulo se van a presentar algoritmos de recorrido, de búsqueda, de distribución y de ordenación sobre estructuras de datos indexadas. Se presentará un esquema general de cada uno de los tipos de algoritmos, independiente del tipo de dato de los elementos que integran la estructura indexada con la que trabajan. Posteriormente se ilustrará cada uno de los esquemas algorítmicos con uno o más ejemplos de aplicación a problemas concretos.

En la presentación de cada uno de los esquemas algorítmicos se va a trabajar con [vectores] de elementos de un tipo genérico que denominaremos `Dato`. Estos esquemas son generales y son válidos tanto para trabajar con estructuras indexadas de registros como de elementos de tipos básicos (**`int`**, **`double`**, **`char`**, **`bool`**, etc.).

Los ejemplos concretos de algoritmos ilustrativos que van a ser presentados se aplicarán a [vectores] cuyos elementos sean datos de tipo [`Persona`], presentados en el capítulo anterior.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Tipos de datos. Esquemas

```
/*
 * Definición de un tipo de dato genérico «Dato» sobre el cual
 * se van a plantear los esquema algorítmicos que se presentan
 * en este tema.
 */
 struct Dato {
    tipoCampo1 c1;              // campo 1º del registro
    tipoCampo2 c2;              // campo 2º del registro
    ...
    tipoCampok ck;              // campo k-ésimo del registro
 };
```

Tipos de datos. Ejemplos

```
const int MAX_LONG_NOMBRE = 24;
const int MAX_LONG_APELLIDOS = 24;

struct Persona {
      char nombre[MAX_LONG_NOMBRE];
      char apellidos[MAX_LONG_APELLIDOS];
      Nif nif;
      Fecha nacimiento;
      bool estaCasado;
      bool esMujer;
};
```

Índice

- **Algoritmos de recorrido**
- Algoritmos de búsqueda
  - Secuencial
  - Binaria
- Algoritmos de distribución
- Algoritmos de ordenación
  - Por selección

Algoritmos de recorrido

Los algoritmos de recorrido de una estructura indexada resuelven problemas que exigen un tratamiento individualizado de cada uno los elementos de la estructura.

En primer lugar se presenta un esquema general válido para cualquier algoritmo de recorrido. A continuación se ilustra su aplicación a la resolución de tres problemas concretos que requieren el desarrollo de algoritmos de recorrido.

En los esquemas que se van a presentar en este capítulo se omiten ciertos fragmentos de código que dependen de cada algoritmo concreto y se sustituyen por una frase explicativa de la misión de cada fragmento, escrita entre corchetes.

Un algoritmo de recorrido de una estructura de datos indexada responde al siguiente esquema general.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmos de recorrido. Esquema

```
/*
 * Pre:  n ≥ 0 y «T» tiene al menos «n» componentes.
 * Post: Se han tratado los datos de las primeras «n»
 *       componentes del vector «T».
 */
void recorrer(const Dato T[], const int n) {
    [ Acciones previas al tratamiento de los datos de «T» ]
    for (int i = 0;  i < n; i++) {
        /* Se han tratado las primeras i-1 componentes de «T» */
        [ Trata ahora el elemento T[i] ]
        /* Se han tratado las primeras i componentes de «T» */
    }
    [ Acciones posteriores al tratamiento de las primeras «n» componentes de «T» ]
}
```

Ejemplo. Mostrar

```
/*
 * Pre:  n ≥ 0 y «T» tiene al menos «n»
 *       componentes.
 * Post: Presenta por pantalla un listado de
 *       la información de las personas de
 *       las primeras «n» componentes del
 *       vector «T».
 */
void mostrar(const Persona T[],
             const int n);
```

Ejemplo. Mostrar

```
void mostrar(const Persona T[],
             const int n) {
    for (int i = 0; i < n; i++) {
        // Se han mostrado las personas de
        // las primeras i-1 componentes de «T»
        mostrar(T[i]);
        cout << endl;
        // Se han mostrado las personas de
        // las primeras «i» componentes de «T»
    }
}
```

La función `mostrar(Persona)` es la presentada en la transparencia 47 del tema anterior.

Ejemplo. Mostrar

```
void mostrar(const Persona T[],
             const int n) {
    for (int i = 0; i < n; i++) {
        mostrar(T[i]);
        cout << endl;
    }
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores
La función `mostrar(Persona)` es la presentada en la transparencia 47 del tema anterior.

Ejemplo. Contar

```
/*
 * Pre:  n ≥ 0 y «T» tiene al
 *       menos «n» componentes.
 * Post: Ha devuelto el número
 *       de solteros de  las
 *       primeras «n»
 *       componentes del vector
 *       «T».
 */
int numSolteros(const Persona T[],
                const int n);
```

Ejemplo. Contar

```
int numSolteros(const Persona T[], const int n) {
   /* Aún no se ha identificado ningún soltero. */
   int cuenta = 0;
   for (int i = 0; i < n; ++i) {
      /* cuenta == nº de solteros de las primeras i-1
       * componentes de «T» */
      if (!T[i].estaCasado) {
            cuenta = cuenta + 1;
       }
      /* cuenta == nº de solteros de las primeras «i»
       * componentes de «T» */
    }
    /* cuenta == nº de solteros de las primeras «n»
     * componentes de «T» */
    return cuenta;
}
```

Ejemplo. Contar

```
int numSolteros(const Persona T[],
                const int n) {
    int cuenta = 0;
    for (int i = 0; i < n; i++) {
        if (!T[i].estaCasado) {
            cuenta = cuenta + 1;
        }
    }
    return cuenta;
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores

Ejemplo.
Determinación de mínimos o máximos

```
/* Pre:  n > 0 y «T» tiene al menos
 *       «n» componentes.
 * Post: Ha devuelto la persona de
 *       más edad de entre las
 *       primeras «n» componentes del
 *       vector «T».
 */
Persona masEdad(const Persona T[],
                const int n);
```

Ejemplo. Determinación de mínimos o máximos

```
Persona masEdad(const Persona T[], const int n) {
    // indMayor == índice de la persona de más edad;
    // incialmente: primera componente del vector «T»
    int indMayor = 0;
    for (int i = 1; i < n; i++) {
        // indMayor == índice de la persona de más edad de entre
        // las primeras «i» - 1 componentes de «T»
        if (esMayorQue(T[i], T[indMayor])) {
            indMayor = i;
        }
        // indMayor == índice de la persona de más edad de entre
        // las primeras «i» componentes de «T»
    }
    // indMayor == índice del más viejo en  las primeras «n»
    // componentes del vector «T»
    return T[indMayor];
}
```

La función `esMayorQue(Persona, Persona)` es la presentada en la transparencia 49 del tema anterior.

Ejemplo.
Determinación de mínimos o máximos

```
Persona masEdad(const Persona T[],
                const int n) {
    int indMayor = 0;
    for (int i = 1; i < n; i++) {
        if (esMayorQue(T[i], T[indMayor])) {
            indMayor = i;
        }
    }
    return T[indMayor];
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores
La función `esMayorQue(Persona, Persona)` es la presentada en la transparencia 49 del tema anterior.

Índice

- Algoritmos de recorrido
- **Algoritmos de búsqueda**
  - Secuencial
  - Binaria
- Algoritmos de distribución
- Algoritmos de ordenación
  - Por selección

Algoritmos de búsqueda

Los algoritmos de búsqueda en una estructura indexada resuelven problemas que exigen el análisis de, al menos, una parte de los datos de la estructura. En función de las circunstancias se podrán aplicar diferentes algoritmos de búsqueda.

- Algoritmos de **búsqueda binaria** en el caso de que los elementos sobre los que se plantea la búsqueda estén ordenados respecto del criterio de búsqueda. Estos métodos son más **eficientes** que los que se citan a continuación. La mayor eficiencia se acentúa en el caso de que la búsqueda se plantee sobre un número de elementos suficientemente grande.
- Algoritmos de **búsqueda secuencial**. Son aplicables en todos los casos. En el caso de que los elementos sobre los que se plantea la búsqueda estén ordenados sólo está justificado el uso de estos algoritmos si el número de elementos sobre los que se plantea la búsqueda es reducido.

Se presentarán esquemas generales de los algoritmos de búsqueda secuencial y binaria, seguidos de ejemplos de aplicación de cada uno de ellos a problemas concretos de búsqueda. Se comienza por el de búsqueda secuencial por la mayor simplicidad de su diseño.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Esquema algorítmico de búsqueda secuencial

La estrategia de una búsqueda secuencial es revisar los sucesivos elementos de[l vector] comenzando por el de menor índice (aunque se podría comenzar también por el de mayor índice). La búsqueda concluye al encontrar el primer elemento que satisface el criterio de búsqueda o, en caso negativo, al haber explorado la totalidad de[l vector] sin éxito.

El esquema de una búsqueda secuencial partiendo desde el índice de valor inferior se muestra a continuación.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmos de búsqueda. Esquema

```
/*
 * Pre:  n ≥ 0 y «T» tiene al menos «n» componentes.
 * Post: Si entre los datos de  las primeras «n»
 *       componentes del vector «T» hay uno que
 *       satisface el criterio de búsqueda, entonces
 *       ha devuelto el índice de dicho elemento en
 *       el vector; si no lo hay, ha devuelto un
 *       valor negativo.
 */
int busqueda(const Dato T[], const int n);
```

Algoritmos de búsqueda Esquema (1/2)

```
int busqueda(const Dato T[], const int n) {
    int i = 0;
    // Espacio inicial de búsqueda: las componentes del vector «T» indexadas
    // entre «i» (== 0) y «n» - 1, ambas inclusive
    bool encontrado = false;
    while (!encontrado && i < n) {
        // Sin éxito tras buscar en las primeras i-1 componentes de «T»
        [ Analiza el elemento T[i] ]
        if (T[i] satisface el criterio de búsqueda) {
            encontrado = true;
        }
        else {
            i = i + 1;
        }
    }
    // encontrado || i ≥ n
    ...
}
```

Algoritmos de búsqueda Esquema (2/2)

```
int busqueda(const Dato T[], const int n) {
    ...
    // encontrado || i ≥ n

    // Discriminación del éxito de la búsqueda
    if (encontrado) {
        return i;
    }
    else {
        return -1;
    }
}
```

Ejemplo. Buscar por DNI

```
/*
 * Pre:  n > 0 y «T» tiene al menos «n»
 *       componentes.
 * Post: Si entre las personas almacenados en
 *       las primeras «n» componentes del vector
 *       «T» hay una cuyo DNI es igual a
 *       «dniBuscado», entonces ha devuelto el
 *       índice de dicho elemento en el vector;
 *       si no lo hay, ha devuelto un negativo.
 */
int buscar(const Persona T[], const int n,
           const int dniBuscado);
```

Ejemplo. Buscar por DNI

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {
    int i = 0;
    bool encontrado = false;
    /* Búsqueda */
    while (!encontrado && i < n) {
        if (T[i].nif.dni == dniBuscado) {
            encontrado = true;
        }
        else {
            i = i + 1;
        }
    }
    // encontrado || i ≥ n
    ...
}
```

Ejemplo. Buscar por DNI

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {
    ...
    // encontrado || i ≥ n
    /* Discriminación del éxito */
    if (encontrado) {
        return i;
    }
    else {
        return -1;
    }
}
```

Ejemplo. Buscar por DNI

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {
    int i = 0; bool encontrado = false;
    while (!encontrado && i < n) {
        if (T[i].nif.dni == dniBuscado) {
            encontrado = true;
        }
        else {
            i = i + 1;
        }
    }
    if (encontrado)
        return i;
    else
        return -1;
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores

Algoritmos de búsqueda
Esquema con garantía de éxito

Una pequeña mejora de la eficiencia de un algoritmo de búsqueda secuencial se logra si está garantizada la presencia dentro el espacio de búsqueda de un elemento, al menos, que satisfaga el criterio. En tal caso el esquema algorítmico de búsqueda secuencial con garantía de éxito es el siguiente.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmos de búsqueda
Esquema con garantía de éxito

```
/*
 * Pre:  n > 0, «T» tiene al menos «n» componentes y entre los datos de las
 *       primeras «n» componentes del vector «T» hay uno que satisface el
 *       criterio de búsqueda.
 * Post: Ha devuelto el índice de un elemento de las primeras «n» componentes
 *       del vector «T» que satisface el criterio de búsqueda.
 */
int busqueda(const Dato T[], const int n) {
    /* Búsqueda secuencial con garantía de éxito */
    int i = 0;
    while (no satisface T[i] el criterio de búsqueda) {
        /* Se han descartado las primeras «i» componentes de «T» */
        i = i + 1;
        /* Espacio búsqueda: componentes de «T» indexadas en [i, n-1] */
    }
    return i;
}
```

Búsquedas en vectores ordenados

- Adivinar un número del 1 al 10
- Preguntas disponibles:
  - ¿Es el número *i*?, con $i \in \mathbb{N}$
- El número es el 6

Búsquedas en vectores ordenados

- Adivinar un número del 1 al 10000
- Preguntas disponibles:
  - ¿Es el número *i*?, con $i \in \mathbb{N}$

Búsquedas en vectores ordenados

- Adivinar un número del 1 al 10000
- Preguntas disponibles:
  - ~~¿Es el número *i*?~~
  - ¿Es mayor que *i*?
  - ~~¿Es menor que *i*?~~

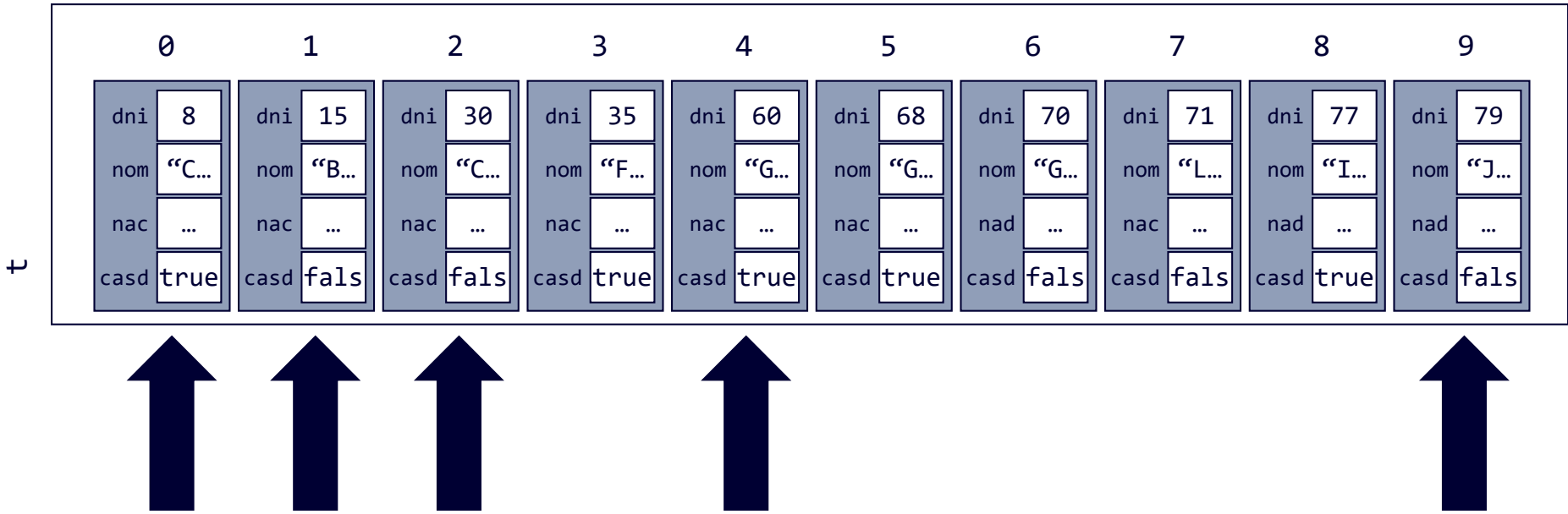
con $i \in \mathbb{N}$

Búsquedas en vectores ordenados

		[1, 10000]
1.	¿Es mayor que 5000? No	→ [1, 5000]
2.	¿Es mayor que 2500? Sí	→ [2501, 5000]
3.	¿Es mayor que 3750? Sí	→ [3751, 5000]
4.	¿Es mayor que 4375? Sí	→ [4376, 5000]
5.	¿Es mayor que 4688? Sí	→ [4689, 5000]
6.	¿Es mayor que 4844? Sí	→ [4845, 5000]
7.	¿Es mayor que 4922? No	→ [4845, 4922]
8.	¿Es mayor que 4883? No	→ [4845, 4883]
9.	¿Es mayor que 4864? Sí	→ [4865, 4883]
10.	¿Es mayor que 4874? No	→ [4865, 4874]
11.	¿Es mayor que 4869? Sí	→ [4870, 4874]
12.	¿Es mayor que 4872? No	→ [4870, 4872]
13.	¿Es mayor que 4871? No	→ [4870, 4871]
14.	¿Es mayor que 4870? No	→ [4870, 4870]

Búsqueda binaria

dniBuscado = 30



Esquema algorítmico de búsqueda binaria

Un algoritmo de búsqueda binaria sólo es aplicable si los elementos sobre los que se realiza la búsqueda están **ordenados** según el criterio que guía la búsqueda.

La estrategia de una **búsqueda binaria** en un[vector] ordenada, también denominada **búsqueda dicotómica**, consiste en dividir en cada iteración en espacio de búsqueda en dos mitades y seleccionar en cuál de ellas debe proseguirse con la búsqueda. La búsqueda concluye al reducirse el espacio de búsqueda a un solo elemento.

El número de iteraciones necesario para acotar la búsqueda a un solo elemento es aproximadamente igual a $\log_2 n$, siendo n el número de elementos sobre el que se plantea inicialmente la búsqueda.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmo de búsqueda binaria
Esquema

```
/*
 * Pre:  n > 0, «T» tiene al menos «n» componentes y los
 *       elementos de las primeras «n» componentes del vector
 *       «T» están ordenados de menor a mayor valor.
 * Post: Si entre los datos almacenados en  las primeras «n»
 *       componentes del vector «T» hay uno que satisface el
 *       criterio de búsqueda, entonces ha devuelto el índice de
 *       dicho elemento; si no lo hay, ha devuelto un valor
 *       negativo.
 */
int buquedaBinaria(const Dato T[], const int n);
```

Algoritmo de búsqueda binaria
Esquema

```
int buquedaBinaria(const Dato T[], const int n,
                   datoBuscado) {
    int inf = 0;
    int sup = n - 1;
    /* Búsqueda */
    while (inf < sup) {
        int medio = (inf + sup) / 2;
        if (el valor de T[medio] es inferior al de datoBuscado) {
            inf = medio + 1;
        }
        else {
            sup = medio;
        }
    }
    ...
}
```

Algoritmo de búsqueda binaria
Esquema

```
int buquedaBinaria(const Dato T[],
                   const int n, datoBuscado) {
    ...
    /* Discriminación del éxito */
    if (T[inf] satisface el criterio de búsqueda) {
        return inf;
    }
    else {
        return -1;
    }
}
```

Algoritmo de búsqueda binaria
Ejemplo

```
/*
 * Pre:  n > 0, «T» tiene al menos «n» componentes y los
 *       datos de las primeras «n» componentes del vector
 *       «T» están ordenados por valores del DNI
 *       crecientes.
 * Post: Si entre las personas almacenados en las
 *       primeras «n» componentes del vector «T»
 *       hay uno cuyo DNI es igual a «dniBuscado»,
 *       entonces ha devuelto el índice de dicho elemento
 *       en el vector; si no lo hay, ha devuelto un valor
 *       negativo.
 */
int buscar(const Persona T[], const int n,
           const int dniBuscado);
```

Algoritmo de búsqueda binaria
Ejemplo

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {
    // Espacio de búsqueda: establecimiento en T[0..n-1]
    int inf = 0;
    int sup = n - 1;

    /* Búsqueda */
    ...

    /* Discriminación del éxito */
    ...
}
```

Algoritmo de búsqueda binaria
Ejemplo

```
...
/* Búsqueda */
// Espacio de búsqueda: T[0..n-1]
while (inf < sup) {
    // Espacio de búsqueda: T[inf..sup]
    int medio = (inf + sup) / 2;
    if (dniBuscado > T[medio].nif.dni) {
        // Espacio de búsqueda: T[medio+1..sup]
        inf = medio + 1;
    }
    else {
        // Espacio de búsqueda: T[inf..medio]
        sup = medio;
    }
    // Espacio de búsqueda: T[inf..sup]
}
// inf >= sup
// Espacio de búsqueda: T[inf]
...
```

Algoritmo de búsqueda binaria
Ejemplo

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {

    ...

    /* Discriminación del éxito */
    if (T[inf].nif.dni == dniBuscado) {
        return inf;
    }
    else {
        return -1;
    }
}
```

Algoritmo de búsqueda binaria
Ejemplo

```
int buscar(const Persona T[], const int n,
           const int dniBuscado) {
    int inf = 0;
    int sup = n - 1;
    while (inf < sup) {
        int medio = (inf + sup) / 2;
        if (dniBuscado > T[medio].nif.dni)
            inf = medio + 1;
        else
            sup = medio;
    }
    if (T[inf].nif.dni == dniBuscado)
        return inf;
    else
        return -1;
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores

Índice

- Algoritmos de recorrido
- Algoritmos de búsqueda
  - Secuencial
  - Binaria
- **Algoritmos de distribución**
- Algoritmos de ordenación
  - Por selección

Algoritmos de distribución

El problema de distribuir los elementos de un[vector *T* de *n* elementos indexados de 0 a *n* − 1 (denotado como *T[0..n−1]*)] , con relación a una determinada **propiedad** *P* que satisfacen *k* de los *n* elementos de[l vector], consiste en permutar los elementos de[l vector] de forma que los elementos reubicados en las *k* primeras posiciones, *T[0..k−1],* sean los que satisfacen la propiedad *P*, mientras que los restantes elementos, *T[k..n−1]*, sean los que no la satisfacen.

Se propone un esquema algorítmico iterativo para la distribución distribución de los datos de *T[0..n-1].* En cada iteración se logra ubicar uno o dos elementos en la parte de[l vector] (parte inferior o superior) que le o les corresponde. La iteración concluye cuando todos los elementos han sido ubicados en la parte que les corresponde.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmo de distribución. Esquema

```
/*
 * Pre:  n > 0, «T» tiene al menos «n» componentes y sea k el
 *       número de elementos de las primeras «n» componentes del
 *       vector «T» que satisfacen una determinada propiedad P.
 * Post: Las primeras «n» componentes del vector «T» son una
 *       permutación de los datos iniciales de «T» en la que
 *       todos los elementos de  las primeras «k-1» componentes
 *       del vector «T» satisfacen la propiedad P y ninguno de
 *       los elementos de las componentes del vector «T» con
 *       índices entre «k» y n-1, ambos inclusive, la satisface.
 */
void distribuir(Dato T[], const int n);
```

Algoritmo de distribución. Esquema

```
void distribuir (Dato T[], const int n) {
    int inf = 0;
    int sup = n - 1;
    while (inf < sup) {
        if (T[inf] satisface P) {
            inf = inf + 1;
        }
        else if (T[sup] no satisface P) {
            sup = sup - 1;
        }
        else {
            Dato aux = T[inf];  T[inf] = T[sup];  T[sup] = aux;
            inf = inf + 1;  sup = sup - 1;
        }
    }
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores

Algoritmo de distribución. Ejemplo

```
/*
 * Pre:  n > 0 y «T» tiene al menos «n» componentes.
 * Post: Las primeras «n» componentes del vector «T»
 *       son una permutación de los datos iniciales
 *       de «T» en la que todas las personas
 *       solteras tienen un índice en el vector
 *       menor que cualquier persona casada.
 */
void distribuir(Persona T[], const int n);
```

Algoritmo de distribución. Ejemplo

```
void distribuir(Persona T[], const int n) {
    int inf = 0;    int sup = n - 1;
    while (inf < sup) {
        if (!T[inf].estaCasado) {
            inf++;
        }
        else if (T[sup].estaCasado) {
            sup--;
        }
        else {
            permutar(T[inf], T[sup]);
            inf++; sup--;
        }
    }
}
```

Código documentado: en las transparencias anteriores y en https://github.com/prog1-eina/tema-12-algoritmos-vectores

Algoritmo de distribución. Ejemplo

```
/*
 * Pre:  uno = A y otro = B
 * Post: uno = B y otro = A
 */
void permutar(Persona& uno, Persona& otro) {
    Persona aux = uno;
    uno = otro;
    otro = aux;
}
```

Índice

- Algoritmos de recorrido
- Algoritmos de búsqueda
  - Secuencial
  - Binaria
- Algoritmos de distribución
- **Algoritmos de ordenación**
  - Por selección

Algoritmos de ordenación

El problema de ordenar los elementos de un[vector], *T[0..n−1]*, con relación a un determinado **criterio**, consiste en permutar los elementos de[l vector] para queden ordenados según dicho criterio.

Existe una variedad de algoritmos de ordenación de [vectores]. En este curso de programación vamos a presentar el algoritmo de **ordenación de [vectores] por selección**. Se ha optado por él por su simplicidad de diseño, aunque no sea el algoritmo de ordenación de [vectores] más eficiente.

Un algoritmo de ordenación por selección tiene una estructura iterativa. En la iteración *i*-ésima se selecciona el menor (o, en su caso, el mayor) de los datos de[l vector] *T[i..n−1]* y se permuta con el elemento *T[i]*. Es inmediato observar dentro del bloque de código a iterar un algoritmo de recorrido para seleccionar el dato menor de[l] sub[vector] *T[i..n−1]*.

Fuente: **Javier Martínez.** «Capítulo 12. Algoritmos básicos de trabajo con estructuras de datos indexadas». *Curso de Programación 1*. 2017

Algoritmos de ordenación
Especificación

```
/*
 * Pre:  n > 0 y «T» tiene al menos «n» componentes.
 * Post: El contenido de las primeras «n»
 *       componentes del vector «T» es una
 *       permutación del contenido iniciales de «T»
 *       en la que todos ellos están ordenados según
 *       un determinado criterio C.
 */
void ordenar(Dato T[], const int n);
```

Ordenación por selección
Esquema

```
void ordenacion (Dato T[], const int n) {
    for (int i = 0; i < n - 1; i++) {
        // Los elementos de las primeras «i» - 1 componentes del vector
        // «T» ya están ordenados según el criterio C.
        int iMenor = i;
        for (int j = i + 1; j < n; j++) {
            if (T[j] es menor que T[iMenor] según C) {
                iMenor = j;
            }
        }
        // T[iMenor] es el menor de T[i..n-1]. Permuta T[i] y T[iMenor]
        Dato aux = T[i];
        T[i] = T[iMayor];
        T[iMayor] = aux;
        // Los elementos de las primeras «i» componentes del vector
        // «T» ya están ordenados según el criterio C.
    }
    // Los elementos de las primeras «i» - 1 componentes del vector
   // «T» ya están ordenados según el criterio C.
}
```

Ordenación por selección
Ejemplo

```
/*
 * Pre:  n > 0 y «T» tiene al menos «n»
 *       componentes.
 * Post: El contenido de las primeras «n»
 *       componentes del vector «T» es una
 *       permutación del contenido inicial de
 *       «T» en la que todas ellas están
 *       ordenados de forma que tienen valores
 *       de DNI crecientes.
 */
void ordenar(Persona T[], const int n);
```

Ordenación por selección
Esquema

```
void ordenar(Persona T[], const int n) {
    // Ordenación de un vector por el método de selección
    for (int i = 0; i < n - 1; i++) {
        /* Las personas de las primeras i-1 componentes de «T» son los de menor
         * valor de DNI y ya están ordenadas */

        // Selección de la persona con menor valor de DNI de T[i..n-1]
        int iMenor = i;
        for (int j = i + 1; j < n; j++) {
            // T[iMenor] es el de menor DNI de T[i..j-1]
            if (T[j].nif.dni < T[iMenor].nif.dni) {
                iMenor = j;
            }
            // T[iMenor] es el de menor DNI de T[i..j]
        }
        // T[iMenor] es el de menor DNI de T[i..n-1]. Permuta T[i] y T[iMenor]
        permutar(T[i], T[iMenor]);

        /* Las personas de las primeras i-1 componentes del vector «T» son los
         * de menor valor de DNI y ya están ordenadas */
    }
}
```

Esquema de búsqueda de mínimo (en solo una parte del vector)

Algoritmo de ordenación por selección

- Select-sort with Gypsy folk dance
  - Extraído de *I Programmer*
    http://www.i-programmer.info/news/150-training-a-education/2255-sorting-algorithms-as-dances.html

Índice

- Algoritmos de recorrido
- Algoritmos de búsqueda
  - Secuencial
  - Binaria
- Algoritmos de distribución
- Algoritmos de ordenación
  - Por selección