Cuaderno de problemas
Fundamentos de algorítmia.

Esquemas algorítmicos básicos

Prof. Isabel Pita

27 de febrero de 2023

Índice

1. Búsqueda binaria.

Este algoritmo busca un valor en una colección ordenada de valores. Los valores pueden ser de cualquier tipo en el que haya definido un orden y la estructura de datos en que están guardados debe permitir el acceso directo (en tiempo constante) a los valores.

Existen diversas versiones de la búsqueda dependiendo del valor que se devuelve. El algoritmo puede devolver sencillamente cierto o falso dependiendo de si el valor buscado se encuentra en la colección o no. Otra posibilidad, que nos proporciona más información, es devolver la posición en que se encuentra o en que se debería encontrar el valor buscado. Esta posibilidad permite dos variantes en el caso de que haya valores repetidos, dependiendo de si se devuelve la posición mas a la izquierda o mas a la derecha en que se encuentra el valor, en el caso de que se encuentre y esté repetido. En el algoritmo que se muestra a continuación se devuelve la posición en que está o debería estar el valor y si este se encuentra repetido se devuelve la posición más a la izquierda. Si se quiere otra opción se debe adaptar ligeramente el algoritmo, manteniendo el esquema básico.

Se presenta el algoritmo aplicado a vectores ordenados de menor a mayor. Si los valores están contenidos en otro tipo de datos habrá que adaptarlo al tipo de estructura utilizado, teniendo en cuenta que el tipo de datos debe permitir acceso directo a los elementos.

```
int binarySearchIz (std::vector<T> const& v, T elem, int ini, int fin) {
    if (ini >= fin) return ini;  // vector vacio
    else if (ini+1 == fin) return ini; // vector 1 elemento
    else {
        int mitad = (ini + fin -1) / 2;
        if (v[mitad] < elem) return binarySearchIz(v,elem,mitad+1,fin);
        else return binarySearchIz(v,elem,ini,mitad+1);
    }
}
```

La llamada inicial a la función es:

```
binarySearchIz(v,elem,0,(int)v.size());
```

El algoritmo está implementado en la librería `algorithm` de la `STL` mediante la función `lower_bound`. Otras variantes están implementadas mediante las funciones `binary_seach` y `upper_bound`.

2. Búsqueda secuencial.

Este algoritmo comprueba si existe un valor en una colección no ordenada de elementos. Se aplica también cuando la colección de valores está guardada en un tipo de datos que no permita el acceso directo (en tiempo constante) a sus elementos. Si la colección de valores está ordenada y el tipo que la almacena tiene acceso directo se debe aplicar el algoritmo de búsqueda binaria explicado a continuación.

Se presenta el algoritmo aplicado a una estructura de datos de tenga definidas las operaciones `size` para obtener el número de elementos en la colección y `[]` para acceder a sus elementos. El algoritmo devuelve la posición en que se encuentra el elemento o el número de valores de la colección si éste no se encuentra.

```
int i = 0;
while (i < v.size() && v[i] != valor)  ++i;
return i ;
```

El algoritmo está implementado en la librería `algorithm` de la `STL` mediante la función `search`.

3. Mezcla ordenada.

El algoritmo obtiene una colección ordenada de valores a partir de dos colecciones ordenadas. Ambas colecciones tienen que estar ordenadas con el mismo criterio, los valores pueden ser de cualquier tipo en el que haya definido un orden y la estructura de datos en la que están guardados puede ser cualquier tipo que nos permita acceder a los datos en el orden definido.

El algoritmo tiene diferentes versiones dependiendo del tratamiento que se de a los valores repetidos de las colecciones. El esquema se aplica con pequeñas variaciones para resolver otros problemas, como

por ejemplo la unión, intersección y la diferencia de dos colecciones ordenadas de valores. En todos los casos, el esquema básico que recorre primero las dos colecciones de valores hasta que termine una de ellas y a continuación trata la colección que no haya terminado en el primer bucle es común a todos ellos.

Se presenta el algoritmo aplicado a vectores. Si los valores están contenidos en otro tipo de datos habrá que adaptarlo al tipo de estructura utilizado. En cualquier caso, primero se recorren las dos colecciones de valores hasta que termine una de ellas y a continuación se trata la colección que no se haya terminado en el primer bucle.

En el siguiente algoritmo `v` y `w` son vectores de datos ordenados de menor a mayor. En la variable `u` se construye la mezcla ordenada. Debe ser del mismo tipo que `v` y `w` y estar vacía.

```
int i = 0, j = 0; // se recorren los dos vectores de izquierda a derecha
while (i < v.size() && j < w.size()) {
// mientras quedan elementos en los dos vectores se selecciona el menor
    if ( v[i] <= w[j] ){ u.push_back(v[i]); ++i; }
    else { u.push_back(w[j]); ++j; }
}
while ( i < v.size() ) {  // Quedan elementos en la primera coleccion
   u.push_back(v[i]); ++i;
}
while ( j < w.size() ) {  // Quedan elementos en la segunda coleccion
   u.push_back(w[j]); ++j;
}
```

Observar que de los dos últimos bucles sólo se ejecuta uno de ellos, dependiendo del vector que se terminó de recorrer en el primer bucle.

El algoritmo está implementado en la librería `algorithm` de la `STL` mediante la función `merge`.

4. Partición en dos partes.

Este algoritmo nos permite dividir los valores de una colección en dos partes: los que cumplen una determinada propiedad y los que no la cumplen. El algoritmo es conocido principalmente por su aplicación en el algoritmo de ordenación `quicksort`, en el que los valores se dividen entre los que son menores y los que son mayores que un valor dado llamado *pivote*.

Se da la implementación del algoritmo sobre un vector. El el lado izquierdo quedan los valores menores o iguales que el valor del *pivote*, en el lado derecho los mayores. Al terminar el algoritmo el índice `p` toma el valor de la posición del primer valor mayor que el *pivote*.

```
int p = 0, q = (int)v.size();
while (p < q) {
    if (v[p] <= pivote)  ++p; // elemento del indice izquierdo correcto
    else if (v[q-1] > pivote) --q;  // elemento del indice derecho correcto
    else {  // Ambos elementos fuera de sitio
        std::swap(v[p],v[q-1]);
        ++p; --q;
    }
}
return p;
```

El algoritmo está implementado en la librería `algorithm` de la `STL` mediante la función `partition`.

5. Partición en tres partes.

Este algoritmo nos permite dividir los valores de una colección en tres partes: los valores menores que uno dado, los valores iguales y los valores mayores. El algoritmo es conocido principalmente por su aplicación en el algoritmo de ordenación `quicksort`, en el que resulta más eficiente que la partición en dos partes si el vector que se ordena tiene muchos valores repetidos.

Se da la implementación del algoritmo sobre un vector. En el lado izquierdo quedan los valores menores que el valor del *pivote*, en el centro quedan los valores iguales y en el lado derecho los mayores. El índice `p` toma el valor de la posición del primer valor mayor o igual que el *pivote* y el índice `q` toma el valor de la

posición del primer valor estrictamente mayor que el pivote. Cualquiera de las tres partes puede quedar vacía.

```
p = 0; q = (int)v.size(); int k = 0;
while (k < q) {
    if (v[k] == pivote) ++k;
    else if ( v[k] < pivote)  {
        std::swap(v[p],v[k]);
        ++p; ++k;
    }
    else {
        std::swap(v[q-1],v[k]);
        --q;
    }
}
```