Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

Tabla de contenidos

2.1. Actividades

2.1.1. Resolución de ejercicios sobre definición de datos en C

Plan de trabajo

1. Resuelve los cuatro primeros ejercicios. Comprueba las soluciones con un compañero, posteando en el foro o consultando con los profesores antes de la siguiente clase.

2.1.2. Manejo de caracteres

Plan de trabajo

Una variable de tipo `char` representa un caracter. Sin embargo, un ordenador sólo guarda código numérico, así que caracteres como 'A', 'a', 'B', 'b', etc., tienen asociado un único número con el que quedan representados. Debido a esto, y teniendo en cuenta el conjunto de caracteres ASCII, las siguientes dos sentencias son equivalentes:

```
char x = 'A';
char x = 65;
```

Pudiéndose además hacer operaciones del tipo:

```
char y = x-5
```

Teniendo en cuenta esto, escribid un programa llamado `to_uppercase.c` que, dada una cadena inicializada de no más de 80 caracteres en minúscula, que convierta todas las letras introducidas a mayúsculas. Pista: Calcula la distancia que hay entre 'a' y 'A' en la notación ASCII (que además es la misma entre 'b' y 'B', 'c' y 'C', etc.).

2.1.3. Manejo de cadenas (*Strings*)

Plan de trabajo

Dado un array de caracteres como el siguiente: `char str_1[] = "Este es un ejercicio de C.";` escribe un programa que copie la cadena desde el array `str_1` a otro, copiando carácter a carácter utilizando un bucle. Al final, imprime las dos cadenas por pantalla.

2.1.4. Manejo de tablas (*Arrays*)

 **Plan de trabajo**

En este ejercicio calcularéis la media entre los elementos de dos arrays. Se pide para ello que imprimáis por pantalla los elementos de dos arrays y que calculéis e imprimáis la media entre cada par de elementos. Para ello, cread en vuestro entorno de desarrollo un fichero llamado `basic_arrays.c`, e implementad en él lo siguiente:

1. Una función que imprima por pantalla los elementos de un array de enteros: `void print_array(int array[]);`
2. Una segunda función que calcule la media entre cada par de elementos de un array y que la vaya imprimiendo por pantalla: `void calculate_average(int array1[], int array2[]);`. Para la primera posición de cada array, sumará los dígitos y los dividirá por dos, y ese resultado lo sacará por pantalla, y así con el resto de posiciones.
3. Una función `main` que declare e inicialice dos arrays de 10 enteros con los dígitos que queráis, por ejemplo `int array1[] = {1,5,7,3,12,...};`, y que haga uso de la primera función para imprimir los arrays y de la segunda para calcular la media.

Cuando acabes, muestra el fichero al profesor.

2.1.5. Manejo de structuras y uniones

 **Plan de trabajo**

En un nuevo fichero llamado `using_structs_and_unions.c`, implementa un programa que tenga las siguientes características:

1. Una estructura denominada `struct survey_information` que pueda almacenar el nombre de una persona y un campo que me permita saber bien la calle de Leganés donde vive (si es de Leganés), o bien el distrito de Madrid donde reside (si no es de Leganés). Pista: Necesitarás en esta estructura un campo que te permita saber si es de Leganés o no.
2. Función `void data_enter(struct survey_information *ptr_survey)`: Esta función inicializará el campo nombre de la estructura a la que apunta `ptr`. Después, inicializará también si la persona de la que se trata es o no de Leganés, indicando la calle en el primer caso, o el barrio en el segundo (escoge la opción que más te guste).
3. Función `void data_display(struct survey_information *survey)`: Que imprima el nombre del usuario y su calle si es de Leganés, o su distrito si no lo es.

Escribe un `main` que primero declare e inicialice los datos del usuario y que luego los muestre por pantalla con las funciones anteriores.

2.1.6. Ordenación por inserción y búsqueda dicotómica

 **Plan de trabajo**

SAUCEM S.L. está preparando una aplicación para desbancar en las aplicaciones móviles a goodreads y shelfari, redes sociales para amantes de los libros.

Este tipo de aplicaciones asignan a usuarios y autores un identificador único, un entero. Además, usan como identificador único de un libro, su ISBN, que actualmente es un entero de 13 dígitos.

SAUCEM S.L. quiere ir implementando la búsqueda de un libro por ISBN que ofrecen estas plataformas pero todavía no tiene claro cómo serán la estructura interna de los datos de su aplicación, así que, como primer paso, nos encomienda que implementemos dicha búsqueda con un array de enteros y que, antes, ordenemos el array.

Para ello deberemos primero ordenar el array que nos den y, después, buscar un elemento en dicho array.

Los pasos que debes seguir para implementar lo que nos encomienda SAUCEM S.L. son:

1. Define en el fichero `insertionSortandSearch.c`, un tamaño de array. Para ello usa la directiva `#define`, escribiendo por ejemplo `#define SIZE 5` antes (y fuera) de tu `main`. Declara en tu `main` un array de tipo `int` de tamaño `SIZE`.

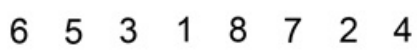
2. Vamos a usar la función `void random_filling(int *array, int size, int max, int min);` para rellenar el array con valores aleatorios. Deberás descargar e incluir el fichero `random_filling.h` en tu código. Además deberás bajarte la biblioteca `random_filling.o`.

   La función `void random_filling(int *array, int nmemb, int max, int min)` recibe el array, el número de miembros de dicho array y los valores máximo y mínimo.

3. Implementa la función `void print_array_integers(int *array, int nmemb)` que imprime un array de enteros de tamaño `nmemb`. Usa esta función en tu `main` para comprobar que se ha rellenado bien el array.

4. Como SAUCEM S.L. nos pide que antes de buscar, ordenemos, vamos a implementar un algoritmo de ordenación. Uno de los más sencillos: el algoritmo de ordenación por inserción.

   Este gif, tomado de la Wikipedia, muestra cómo funciona el algoritmo.

   El algoritmo recorre el array desde `i` igual a `1` hasta `nmemb-1`.

   Guarda el elemento que está en la posición `i` en una variable temporal `tmp` (para no perderlo), dejando así un hueco.

   Va comparando todos los elementos desde la posición `j=i-1` hasta la posición `0` con el valor guardado en`tmp`.

   Si `tmp` es menor que `array[j]`, mueve el hueco a la posición `j`. Es decir, copia `array[j]` en `array[j+1]`.

   Si `tmp` es mayor o igual, copia `tmp` al hueco.

   Deberás implementar dicho algoritmo en la función `void insertion_sort(int *array, int nmemb)`.

   Para ir viendo cómo funciona el algoritmo, imprime por pantalla mensajes del estilo (en este ejemplo `SIZE` es igual a 3):

```
$ ./insertSortDichSearch
Filling with 34 to 3456

Before sorting:
Array = 580 2106 776
**********
 i=1
Array = 580 2106 776
Ordering array[1]=2106
array[0+1]=array[1]
Array = 580 2106 776
**********
 i=2
Array = 580 2106 776
Ordering array[2]=776
j=1
array[1+1] =array[1]
Array = 580 2106 2106
array[0+1]=array[2]
Array = 580 776 2106

After sorting:
Array = 580 776 2106
```

5. Borra la línea donde defines el valor de `SIZE`.

   Ahora compila el ejercicio de esta manera `gcc -Wall -g -o programa insertionSortandSearch.c random_filling.o -DSIZE=15` y ejecútalo.

   Compílalo y ejecútalo para varios valores de SIZE, comprobando que funciona.

6. Ahora queremos deshacernos de los comentarios que pusimos para entender el código. En vez de borrarlos o comentarlos, queremos dejarlos ahí para un futuro, pero que no salgan siempre que ejecutamos.

   En la función `insertion_sort`, antes de cada `printf`, conjunto de printfs o llamada a la función `print_array_integers` escribe `#ifdef INFO_SORT`. Justo después `#endif`

   Por ejemplo, así:

   ```
   #ifdef INFO_SORT
   printf ("**********\n i=%i\n",i);
   print_array_integers(array,size);
   #endif
   ```

   Compila con `gcc -Wall -g -o programa insertSortDichSearch.c random_filling.o -DINFO_SORT -DSIZE=5` Ejecuta y comprueba que aún salen los comentarios.

   Vuelve a compilar con `gcc -Wall -g -o programa insertSortDichSearch.c random_filling.o -DSIZE=10` Ejecuta y comprueba que ya no salen.

7. Ahora queremos buscar un elemento dentro del array ya ordenado. Además, también queremos saber cuántas iteraciones hacen falta para encontrar dicho elemento.

   En este apartado implementaremos la búsqueda por fuerza bruta. Es decir, recorrer el array hasta

que encontremos el número. Cuando lo encontremos, paramos y devolvemos el índice donde lo hemos encontrado. Devuelve un `-1` si no lo hemos encontrado.

La función se llamará `int search_brute_force(int busco,int *array,int nmemb, int *numb_itero)`. Recibe el número a buscar, el array, el número de elementos del array y un puntero donde dejar el número de iteraciones. Devuelve el índice. Si no lo encuentra, devuelve un `-1`.

La función no supone que el array esté ordenado.

Para comprobar que funciona, define un entero en tu main, llama a la función `search_brute_force` e imprime dónde lo ha encontrado y el número de iteraciones necesario para encontrarlo.

```
$ ./insertSortDichSearch
Filling with 2 to 2000
Before sorting:
Array = 321 1212 435 404 178 1605 1330 465 1445 893 1046 743 1271 974 97 169 729 566 1717 787

After sorting:
Array = 97 169 178 321 404 435 465 566 729 743 787 893 974 1046 1212 1271 1330 1445 1605 1717
We want to look for a number: 890
With Brute Force:
No Found. Number of iterations 20

$ ./insertSortDichSearch
Filling with 2 to 2000
Before sorting:
Array = 321 1212 435 404 178 1605 1330 465 1445 893 1046 743 1271 974 97 169 729 566 1717 787

After sorting:
Array = 97 169 178 321 404 435 465 566 729 743 787 893 974 1046 1212 1271 1330 1445 1605 1717
We want to look for a number: 1271
With Brute Force:
Found in index 15. Number of iterations 16
```

8. Ahora implementaremos el algoritmo de búsqueda binaria o dicotómica.

   Este algoritmo supone (no como el anterior) que el array está ordenado. Es quizá el más intuitivo de los algoritmos de búsqueda.

   El algoritmo calcula el índice que está en la mitad del array. Comprueba si el valor buscado es el de dicho índice. Si no lo es, comprueba si está entre el índice inicial y la mitad, o entre la mitad y el índice final. En cualquiera de los dos casos, actualiza los valores de inicial y final para buscar, en la primera mitad o en la segunda mitad. Lo repite sucesivamente hasta que encuentra el valor.

   Implementa la función `int search_dichotomic_iterative(int busco,int *array,int ind_min,int ind_max, int *numb_itero)`.

   Para entender mejor el funcionamiento puedes imprimir, al igual que hicimos en la ordenación, los pasos intermedios tanto del algoritmo de búsqueda por fuerza bruta como del algoritmo de búsqueda dicotómica. Para ello, no te olvides de usar `#ifdef INFO_SEARCH` y `#endif` en el código y de compilar con `-DINFO_SEARCH`.

   La salida con comentarios debería ser similar a ésta:

   ```
   $ ./insertSortDichSearch
   ```

```
Filling with 2 to 8900
Before sorting:
Array = 1424 5390 1931 1795 788

After sorting:
Array = 788 1424 1795 1931 5390
We want to look for a number: 1931

***************We are in Brute Force Algorithm***********
In array[0]=788
In array[1]=1424
In array[2]=1795
Found !! in i=3
Number of iterations: 4

******************We are in Dichotomic Search Algorithm**********
Searching between indexes array[0] = 788 and array[4] = 5390
Comparing with array[2]=1795
Searching between indexes array[3] = 1931 and array[4] = 5390
Comparing with array[3]=1931
Found !! in i=3
Number of iterations: 2

******************Results******************
With Brute Force:
Found in index 3. Number of iterations 4
With Dichotomic Search:
Found in index 3. Number of iterations 2
```

9. Por último queremos comprobar qué ocurre cuando tenemos arrays muy grandes. Por ejemplo de 200000 o 300000 elementos.

   Como imprimir todos los elementos llevaría mucho tiempo, no los vamos a imprimir por pantalla. Usa `#ifdef IMPRESION` en el código y `-DIMPRESION` al compilar, si quieres que aparezcan.

   Lo primero que nos damos cuenta es lo mucho que tarda en rellenar el array. Si ves que tarda mucho, compila con `-DSIZE=20000`

```
$ ./insertSortDichSearch
Filling with 2 to 30000

We want to look for a number: 5678

With Brute Force:
Found in index 3738. Number of iterations 3739

With Dichotomic Search:
Found in index 3739. Number of iterations 11

$ ./insertSortDichSearch
Please, introduce a min: 2

Please, introduce a max: 30000
Filling with 2 to 30000

We want to look for a number.
```

```
Please, introduce a number: 5679

With Brute Force:
No Found. Number of iterations 20000

With Dichotomic Search:
No Found. Number of iterations 14
```

Puedes observar que el número de iteraciones es muy inferior en el algoritmo de búsqueda dicotómica. Esto es debido a la complejidad de los algoritmos.

La complejidad computacional mide el número de pasos (que se traduce en tiempo) que se necesitan para resolver un problema en el caso medio, en el mejor y en el peor caso.

En el caso de los algoritmos de ordenación, lo que contamos son las operaciones necesarias en el mejor y en el peor caso para ordenar un array.

En un fichero en tu directorio de trabajo deberás dejar como entregable, en un fichero llamado `cuestion.txt` tu estimación de la complejidad de los tres algoritmos implementados en esta práctica: los dos algoritmos de búsqueda y el algoritmo de ordenación por inserción.

10. ACTIVIDAD ADICIONAL: implementación de la función `void random_filling(int *array, int nmemb, int max, int min)` que recibe el array, el número de miembros de dicho array y los valores máximo y mínimo.

    La función deberá hacer uso de la función `int rand(void);` de la biblioteca `stdlib.h` que devuelve un número aleatorio entre `0` y `RAND_MAX`.

    Deberá realizarse la transformación para que el número generado esté entre `min` y `max`. Es decir, en vez de estar en [`0`, `RAND_MAX`] que esté en [`min`, `max`].

    Para ello se usará la fórmula: `min + rand()/RAND_MAX * (max-min)`. Ten cuidado porque es necesario hacer castings para no perder precisión (o perder el número!!). Ponlos en el sitio adecuado.

    Usa tu función `void random_filling` en vez de la que te hemos proporcionado y comprueba que funciona.