Práctica entregable de los bloques 3 y 4: LATEX, git, python, Numpy, Matplotlib, gdb y profiling

Victor Espín Belmonte y Jaime Ortiz Aragón

April 2020

1 Planteamiento del problema

La resolución de los Bloques III y IV de la asignatura constan de desarrollar un programa propuesto en Phyton y C, ayudándonos de las estructuras que se nos han proporcionado en las presentaciones de teoría *LatexTeoria.pdf, GitTeoria.pdf, PythonTeoria.pdf y teii-bloque4.pdf* así como en las estructuras del código en *bloque4-codigo.tgz*. La implementación del código la hemos ido recopilando en la aplicación **github.com** con el fin de haber podido mostrar nuestro progreso, el cual hemos ido realizando a la par por videollamadas de **Skype**.

En concreto, el problema a resolver en esta tarea consistía en dada una lista de tamaño variable de entrada, controlar los elementos repetidos que contiene, en nuestro caso *int*, para proporcionar como salida una lista sin repetidos.

Estos valores pueden variar en el intervalo [**0-99999**] y la longitud de la lista puede ser de hasta **200000** valores, siendo el valor máximo el que se establece como referencia en el enunciado de la tarea.

2 Resolución del problema

2.1 Generación aleatoria de números

El primer paso que realizamos para la confección del problema consistió en comprobar los argumentos que debía recibir el programa : los argumentos de entrada para generar el fichero con los números de la lista completa son el nombre del fichero y el número de enteros que contiene.

Para generar en *Phyton* el fichero con la lista de números aleatorios hicimos uso de la librería *Numpy*, que nos permite mediante la función *np.random.randint(valormax)* generar un número en el intervalo [**0-valormax**]. Para que esto generara la lista, simplemente debíamos realizar dicha operación para cada elemento, con su escritura en el fichero pasado como parámetro.

Este código se encuentra en el fichero *Fichero.py* en la carpeta de *Phyton*.

```
import sys
import string
import numpy as np

def main():
    # Control de argumentos de línea de comandos:
    if len(sys.argv) != 3:
        print("Uso: {} fichname number".format(sys.argv[0]))
        sys.exit(0)
    nombre = sys.argv[1]
    numeros= int(sys.argv[2])
    fichero = open(nombre,'w')
    for i in range(0,numeros):
        num = np.random.randint(99999)
        fichero.write(str(num) + '\n')
    fichero.close()
if __name__ == '__main__':
    main()
```

Aquí podemos ver la captura del codigo. Como se extraen los argumentos y como se genera el fichero con el bucle for que genera números aleatorios.

2.2 Programa principal en *Phyton*

El programa principal, que se encuentra en el fichero *Practica3.py*, recibe como parámetros el nombre del fichero que se generó en el apartado **2.1**, el fichero en

el que va a escribir la lista de números sin repeticiones y el fichero pdf en el que se va a guardar la gráfica que se nos pide representar.

```
try:
    ficheroPDF = sys.argv[3]
    comprobacion = ficheroPDF.split(".")
    if (len(comprobacion)!=2 or comprobacion[1] != "pdf"):
        raise ValueError()
except:
    print("Debe de introducir un fichero pdf")
    sys.exit(-1)

try:
    ficheroSalida = sys.argv[2]
    #comprobamos que el fichero no tenga ninguna extension, si la tiene
    #pedimos que el usuario meta un nombre simple
    comprobacion = ficheroSalida.split(".")
    if (len(comprobacion)!=1):
        raise ValueError()
except:
    print("Debe de introducir simplemente un nombre")
    sys.exit(-2)
try:
    ficheroEntrada = sys.argv[1]
    with open(ficheroEntrada,'r') as reader:
        for line in reader:
            numeros.append(int(line))
    if (len(numeros)!=200000):
        raise ValueError()
except:
    print("El fichero de entrada debe existir")
```

En esta imagen podemos ver como se trata que los parámetros de entrada sean o no correctos. Primero vemos como comprobamos utilizando split que la extensión del fichero para el pdf sea .pdf, si esto no es así acabaremos con un error. Después comprobamos que el fichero de salida, en el que se escriben los números sin repeticiones, no tenga ninguna extensión rara también con split. Si el fichero tiene una extensión abortamos. Finalmente para comprobar que el fichero de entrada tiene doscientos mil números simplemente metemos todas las líneas en una lista y comprobamos que su longitud sea de doscientos mil elementos.

Se nos pide que resolvamos el problema de dos formas distintas y tras estudiar órdenes de complejidad, nos decantamos por utilizar conjuntos y diccionarios. Ambos presentan un **O(n)** en sus operaciones para eliminar los repetidos. Por definición, un conjunto es una colección de valores que no presenta repetidos, por lo que si pasamos una lista a un conjunto, en este podremos encontrar todos los valores de la lista eliminando los repetidos. Esto lo realizamos en la línea *list(set(aux))*, que transforma la lista *aux* en un conjunto y posteriormente en una lista de nuevo, con los valores del conjunto.

Por otra parte, en Python, un diccionario es una colección no-ordenada

de valores que son accedidos a través de una clave. Estos tipos pueden ser listas, cadenas, tuplas, otros diccionarios, objetos, etc. Como es de esperar, una tupla no puede estar repetida, ya que cada clave hace referencia a una tupla del diccionario. Para conseguir pasar una lista con elementos repetidos a un diccionario solo debemos usar *dict.fromkeys(aux).keys()*. Esto lo haremos en un bucle while en el que se irán procesando cada vez mas elementos de la lista, concretamente dos mil más en cada iteración.

```
try:
    ficheroPDF = sys.argv[3]
    comprobacion = ficheroPDF.split(".")
    if (len(comprobacion)!=2 or comprobacion[1] != "pdf"):
        raise ValueError()
except:
    print("Debe de introducir un fichero pdf")
    sys.exit(-1)

try:
    ficheroSalida = sys.argv[2]
    #comprobamos que el fichero no tenga ninguna extension, si la tiene
    #pedimos que el usuario meta un nombre simple
    comprobacion = ficheroSalida.split(".")
    if (len(comprobacion)!=1):
        raise ValueError()
except:
    print("Debe de introducir simplemente un nombre")
    sys.exit(-2)
try:
    ficheroEntrada = sys.argv[1]
    with open(ficheroEntrada,'r') as reader:
        for line in reader:
            numeros.append(int(line))
    if (len(numeros)!=200000):
        raise ValueError()
except:
    print("El fichero de entrada debe existir")
```

En la imagen vemos como transformamos la lista en un set y en un diccionario. De esta forma eliminamos elementos repetidos. También podemos ver como calculamos los tiempos en los que esta transformación se lleva a cabo.

Otra solución que podríamos haber adoptado son el ordenar la lista e ir buscando si el elemento n es igual al $n+1$ para ir eliminando. Otra más podría haber sido tener un array de **0-valormax**, y cuando vayamos encontrando un valor, marcarlo como encontrado en el array.

Como cabe de esperar, el orden de magnitud es mucho mayor al adoptado en nuestras soluciones.

```
def plot_values(x, y,y2,y3,width=0.5):

    plt.plot(x,y,color = 'g', label="Tiempo set")
    plt.plot(x,y2,color = 'r', label="Tiempo Dic")
    plt.plot(x,y3,color = 'b', label="Tiempo C")
    plt.title('Tiempos de uso')
    plt.legend()
    plt.xlabel('Numero de Valores')
    plt.ylabel('Tiempos')
    return plt
```

En esta función auxiliar podemos observar como utilizando matplotlib podemos dibujar una gráfica. En las tres primeras lineas dibujamos las líneas correspondientes a los tres métodos. Seguidamente colocamos el titulo y una leyenda. Para finalizar con la gráfica le damos nombre a los dos ejes.

```
fichero = open(ficheroSalida,'w')
for numero in sinRepeticiones1:
        fichero.write(str(numero) + '\n')
```

Antes de pasar a explicar como funciona el uso de la librería ctypes para utilización de código C en python, vemos como para terminar el programa se escribe la lista de números sin repeticiones en el fichero de salida.

Finalmente hay que indicar que antes de poder ejecutar el programa como tal habrá que ejecutar el programa Fichero.py con un nombre de fichero y 200000 como segundo argumento. También habrá que hacer make en la carpeta de C para que se cree la librería necesaria.

2.3 Programa principal en C

Para pasar el código de *Phyton* a *C* nos hemos basado en el fichero que se nos ofreció *mult.py*.

Dado que se nos pide que la reserva de memoria se haga en *Python*, debemos conseguir hacer desde aquí una llamada al código *C*, que se encontrará en la carpeta C correspondiente.

En nuestro fichero *Listas.py* utilizamos la biblioteca de *Ctypes*.

```
if __name__ == "Listas":
    # Cargamos la biblioteca compartida en ctypes.
    LIBList = ctypes.CDLL (os.path.abspath(os.path.join(os.path.dirname(__file__),  "../C/libListas.so.1")))
```

Para cargar la biblioteca compilada por nuestro makefile usaremos la librería os de python y le pasaremos la ruta.

```
# Objeto correspondiente a la función dentro de la biblioteca.
funcLista = LIBList.Listas

# Prototipo de la función: De la lista sin elementos duplicados,
#le pasamos un puntero a int. Observa que ctypes no define punteros a datos que
# no sean c_char, c_wchar y c_void, por lo que hay que crearlos con POINTER.
funcLista.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int,
                      ctypes.POINTER(ctypes.c_int)]

# Valor devuelto por la función (se puede eliminar, pues es el
# comportamiento por defecto).
funcLista.restype = ctypes.POINTER(ctypes.c_int)

# Puesto que ctypes espera que el primer parámetro sea instancia
# de punteros a c_int (es decir, instancias LP_c_int), según el
# prototipo de la función, no podemos usar directamente lista
# ya que es de tipo list y no instancias de
# LP_c_int. Lo que hacemos es definir salida como un array de valores
# c_int, con tantos elementos como tiene lista.
longitud=(ctypes.c_int*1)()
 # Llamada a la función de la biblioteca compartida.
salida1=funcLista((ctypes.c_int * len(lista))(*lista), len(lista),longitud)
```

En la primera línea podemos ver como se carga la función en c dentro de la biblioteca. Después definimos los parámetros de la función. Estos parámetros tienen que estar dentro de los tipos de ctypes. En este caso utilizaremos ctypes.Pointer(ctypes.cint) para el primer y el tercer argumento, lo que nos indica que serán punteros a enteros como veremos más adelante. Para el segundo parámetro simplemente indicaremos que será un ctypes.cint indicando que es un entero.

Siguiendo con la imagen podemos ver como definimos el tipo de retorno de la función como un puntero de enteros, que será la lista sin repeticiones. Para terminar con esta imagen vemos como definimos que el tercer parámetro sea un array de una sola posición, esto lo hacemos así por que necesitamos almacenar la longitud de la lista sin repeticiones para recorrerla posteriormente. También podemos ver como llamamos a la función con todos sus parámetros correspondientes.

```
# Vamos a devolver una lista. Para eso, hacemos una copia de la lista de
# entrada. Podríamos devolver directamente «salida», pero sería un objeto de
# ctypes tal y como lo hemos definido, con lo que una simple orden
# «print(salida)» no nos mostraría su contenido sino su tipo. Queda más
# "elegante" devolver algo como la entrada.
listaout=[]
aux = longitud[0]
# Copiamos a dicho vector de salida el resultado de la función.
for i in range(aux):
  listaout.append(salida1[i])

# Devolvemos el vector de salida.
return listaout
```

Finalizando con Listas.py tenemos como devolver la lista sin repeticiones. Extraemos la longitud de la lista sin repeticiones del tercer parámetro de la función. Con este dato recorremos toda la lista de salida introduciendo los datos en una lista de python para devolverla posteriormente

```
int compare_ints(const void* a, const void* b)
{

    return (*(int *)a - *(int *)b);

}
```

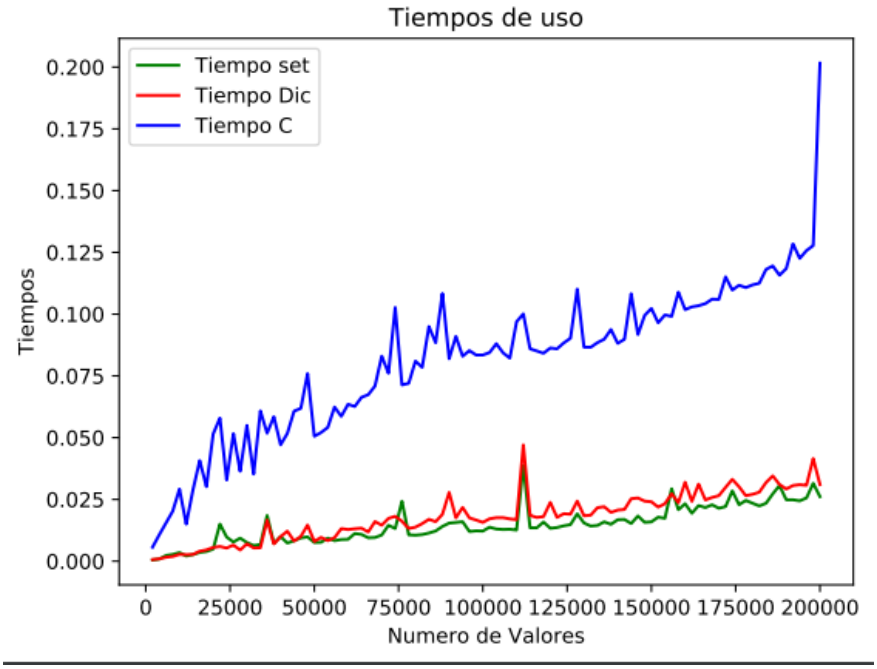
Antes de pasar a la función principal en C tenemos esta función auxiliar que pasaremos a la función qsort como parámetro para ordenar la lista, ya que esto nos bajara el orden de la función bastante. Esta función lo único que hace básicamente es devolver la resta entre los dos números para saber cual es mayor de los dos.

```
int* Listas(int * lista, const int size,int * longitudFinal)
{

    int temp[size];
    int j=0;
    qsort(lista,size,sizeof(int),compare_ints);
    for(int i=0; i<size-1;i++)
        if(lista[i] != lista[i+1])
            temp[j++]=lista[i];
    temp[j++] = lista[size-1];
    int * salida = (int*)malloc(j*sizeof(int));
    for(int i=0;i<j;i++) {
        salida[i] = temp[i];
    }
    longitudFinal[0] = j;
    return salida;
```

Finalmente para el algoritmo en C primero ordenamos la lista, después simplemente iremos recorriendo toda la lista y comprobando si el siguiente elemento es distinto al que tenemos. Si lo es simplemente tendremos que colocarlo en la lista de salida. Para terminar creamos una lista del tamaño preciso para eliminar los posibles residuos que nos hayan quedado.

2.4 Comparación de tiempos de operaciones

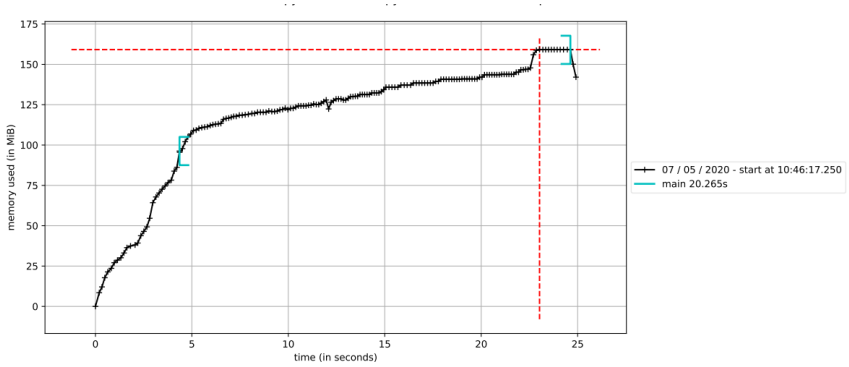


Como era de esperar aun que el orden de nuestro programa en c es el orden de qsort, que debería estar en torno a nlogn, las soluciones de python son mucho mas eficientes. Esto es así por que estarán implementadas de una mejor manera usando hash o con algoritmos mucho mas eficientes. Para hacer un buen uso de la biblioteca ctypes es necesario plantear muy bien en que problema puede ser beneficioso, dado que si hay algún método o librería ya disponible en python es probable que no sea rentable usarla.

```
84        100        1461.0     14.6      0.0          t0 = t.time_ns()
85        100     3854303.0  38543.0      7.3          list(set(aux))
86        100        1805.0     18.1      0.0          t_exec1 = (t.time_ns()-t0)/1.0e9
87
88        100         434.0      4.3      0.0          t2 = t.time_ns()
89        100     4922179.0  49221.8      9.3          dict.fromkeys(aux).keys()
90        100        5452.0     54.5      0.0          t_exec2 = (t.time_ns()-t2)/1.0e9
91
92        100         366.0      3.7      0.0          t3 = t.time_ns()
93        100  30088340.0 300883.4     56.9          sinRepeticiones1=Listas(aux)
94        100        1607.0     16.1      0.0          t_exec3 = (t.time_ns()-t3)/1.0e9
95
```

En esta imagen podemos ver el uso de la herramienta line profile para el calculo del tiempo de ejecución de las funciones. En la primera columna podemos ver el numero de linea en el programa. En la segunda columna el número de veces que entra a esa linea. Finalmente en las dos siguientes columnas podemos

ver el tiempo total que utiliza cada linea y el tiempo por entrada a esa linea respectivamente. Como podemos comprobar la llamada a la función en C es la que mas tiempo usa, un con algo mas de la mitad del tiempo de ejecución, mientras que el tiempo del set es algo menos que el de los diccionarios. Todo esto ya lo podíamos ver en el pdf que genera el programa automáticamente.



Finalmente para terminar con el estudio del programa tenemos el estudio de la memoria. Como podemos ver el gran uso de memoria se debe al main. Esto es así por que en esta función es donde se crean tanto los nuevos sets y los nuevos diccionarios a parte de que es donde se reserva toda la memoria que se va a utilizar en C mediante el uso de ctypes. Es por esto que es de lo más normal del mundo que haya un crecimiento tan alto en esta parte del programa.