



Práctica Final: Cifras y Letras

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada



Estructuras de Datos

Grado en Ingeniería Informática

Doble Grado en Ingeniería Informática y ADE

Doble Grado en Ingeniería Informática y Matemáticas

Índice de contenido

1. Introducción.....	3
2. Objetivo.....	3
2.1. Tareas a realizar.....	3
2.1.1. El problema de las cifras.....	3
2.1.2. El problema de las letras.....	4
3. TDA y Programas.....	5
3.1 Letras.....	5
3.1.1. Test diccionario.....	5
3.1.2. Letras.....	5
3.1.3. Cantidad de Letras.....	7
3.1.4. Fichero diccionario.....	8
3.1.5. Fichero Letras.....	8
3.1.6. Módulos a desarrollar.....	8
3.2 Cifras.....	11
3.2.1. Estructuras lineales o jerárquicas.....	11
4. Práctica a entregar.....	11
5. Valoración.....	11
6. Referencias.....	11

1½ Prueba de cifras

Objetivo: **19**

2 4 8 2 6 5

1½ Prueba de letras

z a r p a c r s e



1. Introducción

Los objetivos de este guión de prácticas son los siguientes:

- Resolver un problema eligiendo la mejor estructura de datos para las operaciones que se solicitan

Los requisitos para poder realizar esta práctica son:

1. Haber estudiado el Tema 1: Introducción a la eficiencia de los algoritmos
2. Haber estudiado el Tema 2: Abstracción de datos. Templates.
3. Haber estudiado el Tema 3: T.D.A. Lineales.
4. Haber estudiado el Tema 4: STL e Iteradores.
5. Haber estudiado estructuras de datos no lineales

2. Objetivo.

El objetivo de esta práctica es llevar a cabo el análisis, diseño e implementación de un proyecto. Con tal fin el estudiante abordará un problema donde se requiere estructuras de datos que permiten almacenar grandes volúmenes de datos y poder acceder a ellos de la forma más eficiente.

2.1. Tareas a realizar

El estudiante deberá implementar varios programas simulando el juego “Cifras y Letras”. Este juego tiene dos partes. En la primera se trata de conseguir a partir de un conjunto de cifras específicas, y usando operaciones elementales, un número de 3 cifras concreto. La segunda parte consiste en dada una serie de letras escogidas de forma aleatoria, obtener la palabra existente en un diccionario formada a partir de ellas de mayor longitud o que tenga la mayor puntuación.

Se deberán llevar a cabo las siguientes tareas:

1. Definir los T.D.A. que se consideren necesarios para la solución del problema propuesto.
2. Probar los módulos con programas test.

Se puede usar la STL en todos los módulos

A continuación se detalla el juego y los programas que se deberán desarrollar.

2.1.1. El problema de las cifras

Esta parte del juego trata de: dado un conjunto de 6 enteros sacados aleatoriamente del conjunto:

$$C = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100\}$$

conseguir otro entero aleatorio de 3 cifras usando para ello sólo las operaciones de suma, resta, producto y división entera, teniendo en cuenta que solo se puede usar cada número (de los 6) como mucho una vez, aunque es posible que no todos se usen para conseguir el número de 3 cifras.

Ejemplo:

- Se sacan aleatoriamente del conjunto C los números:

6, 8, 10, 9, 4, 75

y se pide que con ellos se consiga el número (también aleatoriamente generado):

835

Una posible solución (que no tiene por qué ser única) es esta:

- $8/4 = 2$
- $9+2 = 11$
- $75*11 = 825$
- $825+10 = 835$

En este caso, se han usado solo 5 de los 6 números y sin usar ninguno más de una vez (el número 6 no ha hecho falta en esta solución) y sólo se ha usado operaciones de +, * y / (la resta en este caso tampoco ha hecho falta) para conseguir llegar al número exacto.

No pueden tenerse resultados temporales negativos, es decir, pasos intermedios como $4-8=-4$ y usar ese -4 no están permitidos, como tampoco está permitido hacer una división no exacta, es decir no puede hacerse $75/11$ y redondear.

Obviamente puede que sea imposible que con 6 números aleatorios se pueda conseguir el número aleatorio de 3 cifras y en ese caso hay dos salidas:

- a) la más simple: que el algoritmo diga que no hay solución
- b) la más interesante: que la salida del algoritmo sea conseguir el número más próximo posible al que nos piden

Evidentemente en la generación aleatoria podría haber repeticiones y salir p.ej. estos 6 números:

$$\{1, 3, 5, 3, 100, 5\}$$

donde el 3 y el 5 aparecen 2 veces. No pasa nada, esto es válido, simplemente que para alcanzar la solución contamos con dos treses y 2 cincos, nada más, pero sigue estando presente la restricción de la no repetición, es decir que contamos con 6 números, un 1, dos 3, dos 5 y un 100, es decir se puede usar cada número de la serie SOLO UNA VEZ (como mucho, una vez el 1, una vez el primer 3, una vez el primer 5, una vez el segundo 3, una vez el 100 y una vez el segundo 5).

Como curiosidad, indicar que existen combinaciones de 6 números de C con las que puede obtenerse cualquier número de 3 cifras (y sería interesante pensar en cómo podrían conseguirse tales combinaciones) P.ej. a partir del conjunto $C=\{2, 6, 7, 9, 50, 75\}$ se puede conseguir cualquier número de 3 cifras.). Se denominan combinaciones mágicas.

2.1.2. El problema de las letras

Esta parte del juego consiste en dada una serie de letras escogidas de forma aleatoria obtener la palabra existente en el lista_palabras formada a partir de ellas de mayor longitud.

Por ejemplo dadas las siguientes letras:

O D Y R M E T

una de las mejores soluciones sería METRO.

El tamaño del conjunto de letras de partida lo decide el usuario y dichas letras pueden repetirse

3. TDA y Programas

3.1 Letras

3.1.1. Test diccionario

Este programa permitirá comprobar el buen funcionamiento del T.D.A Diccionario, representado como un set. Para ello el código fuente “testdiccionario.cpp” deberá funcionar con el T.D.A Diccionario que se desarrolle (ver sección 3.1.6).

```

1.  #include <fstream>
2.  #include <iostream>
3.  #include <string>
4.  #include <vector>
5.  #include <set>
6.  int main(int argc, char * argv[]){
7.      if(argc != 2){
8.          cout << "Los parametros son: " << endl;
9.          cout << "1.- El fichero con el diccionario" << endl;
10.         return 1;
11.     }
12.     ifstream f_diccionario(argv[1]);
13.     if(!f_diccionario){
14.         cout << "No puedo abrir el fichero " << argv[1];
15.         return 0;
16.     }
17.     Dictionary diccionario;
18.     f_diccionario >> diccionario;
19.     for (Dictionary::iterator it = diccionario.begin(); it !=
20.         diccionario.end(); ++it){
21.         cout << *it << endl;
22.     }
23.     cout<<endl;
24.     cout<<"Numero de palabras en el diccionario "<<
        diccionario.size()<<endl;
25.
26.     //Buscamos todas las palabras con prefijo zurr
27.     string prefijo = "zurr";
28.
29.     pair<Dictionary::iterator, Dictionary::iterator >ret =
        diccionario.range_prefix(prefijo);
30.     for (Dictionary::iterator it = ret.first; it != ret.second; +
        +it){
31.         cout<<*it<<endl;
32.     }
33.     //Borando una palabra
34.     string w = "zurrusco";
35.     if (diccionario.erase(w)){
36.         cout<<"Palabra borrada "<<diccionario.size()<<endl;
37.     }
38.     else
39.         cout<<"Palabra noborrada "<<diccionario.size()<<endl;
40.
41.     int longitud;
42.     cout<<"Dime la longitud de las palabras que quieres
        ver ";
43.     cin>>longitud;
44.     vector<string>
        v=diccionario.getWordsLength(longitud);
45.     for (int i=0;i<v.size();i++)
46.         cout<<v[i]<<endl;
47.
48.     string input;
49.     cout<<"Dime una palabra ";
50.     cin>>input;
51.     if (diccionario.find(input)!=diccionario.end())
52.         cout<<"La palabra esta"<<endl;
53.     else
54.         cout<<"La palabra no esta"<<endl;
55.     return 0;
56. }

```

En este código, se carga el diccionario en memoria y luego se imprime por la salida estándar. A continuación se muestran todas las palabras del diccionario de una longitud dada. Y finalmente dada una palabra por el usuario, el programa indica si existe tal palabra en el diccionario o no. El estudiante creará por lo tanto el programa testdiccionario, que se deberá ejecutar en la línea de órdenes de la siguiente manera:

```
prompt% testdiccionario spanish
```

El único parámetro que se da al programa es el nombre del fichero donde se almacena el diccionario.

3.1.2. Letras

Este programa construye las palabras de longitud mayor (o de puntuación mayor) de un diccionario a partir de una serie de letras seleccionadas de forma aleatoria. El programa letras se deberá ejecutar en la línea de órdenes de la siguiente manera:

```
prompt% letras spanish letras.txt 8 L
```

Los parámetros de entrada son los siguientes:

1. El nombre del fichero con el diccionario
2. El nombre del fichero con las letras
3. El número de letras que se deben generar de forma aleatoria
4. Modalidad de juego:
 - Longitud: Si el parámetro es *L* se buscará la palabra más larga.
 - Puntuación: Si el parámetro es *P* se obtendrá la palabra de mayor puntuación.

Tras la ejecución en pantalla aparecerá lo siguiente:

```
Las letras son: T      I      E      O      I      T      U      S
Dime tu solucion:tieso
tieso  Puntuacion: 5

Mis soluciones son:
otitis Puntuacion: 6
tiesto Puntuacion: 6
Mejor Solucion:tiesto
¿Quieres seguir jugando[S/N]?N
```

En primer lugar el programa genera 8 letras. Estas letras se escogen, de forma aleatoria, entre las dadas en el fichero letras (ver sección 3.1.5). Una vez generadas las letras, el programa pide al usuario su solución, en el ejemplo la solución dada es “tieso”. A continuación se muestra la solución del usuario junto con su puntuación. Y finalmente se muestran las soluciones dadas por el programa. Para generar de forma aleatoria las letras con la que construir la palabra, el estudiante creará una *Bolsa de Letras* (contenedor de letras que se disponen de forma aleatoria) en la que el número de veces que aparece cada letra, en la *Bolsa de Letras*, viene dado por la columna *Cantidad* del fichero de letras.

En el caso de que el usuario haya escogido jugar en modo “*Puntuacion*”, como resultado se

*****Puntuaciones Letras*****		Z	10
A	1		
B	3	Las letras son:	
C	3	N	S A O T O A I
D	2	Dime tu solucion:sonata	
E	1		
F	4		
G	2	sonata Puntuacion: 6	
H	4	Mis Soluciones son:	
I	1	asiano Puntuacion: 6	
J	8	atonia Puntuacion: 6	
L	1	ostion Puntuacion: 6	
M	3	sonata Puntuacion: 6	
N	1	sotana Puntuacion: 6	
O	1	sotani Puntuacion: 6	
P	3	sotano Puntuacion: 6	
Q	5	tisana Puntuacion: 6	
R	1	toison Puntuacion: 6	
S	1	Mejor Solucion:toison	
T	1	¿Quieres seguir jugando[S/N]?N	
U	1		
V	4		
Y	4		

obtendrán las palabras que acumulen una mayor puntuación. Para obtener la puntuación de la palabra simplemente tenemos que sumar las puntuaciones de las letras en la palabra (en el fichero de Letras la puntuación de cada letra viene dada en la columna Puntos).

En ambas versiones, se le preguntará al usuario si quiere seguir jugando. En caso afirmativo se generará una nueva secuencia de letras aleatorias para jugar de nuevo. En otro caso el programa termina.

En la sección se detallan los formatos de los ficheros de entrada al programa.

3.1.3. Cantidad de Letras

El programa cantidad_letras obtiene la cantidad de instancias de cada letra (ver fichero letras en la sección 3.1.5). El programa se deberá ejecutar en la línea de órdenes de la siguiente manera:

```
prompt% cantidad_letras spanish letras.txt salida.txt
```

Los parámetros de entrada son los siguientes:

1. El nombre del fichero con el diccionario
2. El nombre del fichero con las letras
3. El fichero donde escribir el conjunto de letras con la cantidad de apariciones calculada.

Este programa una vez haya cargado el fichero diccionario en memoria y el conjunto de letras, obtiene para cada letra en el conjunto el número de veces que aparece en el diccionario, es decir encuentra la frecuencia de aparición (Fabs.). Finalmente se obtiene el tanto por ciento, sobre el total de las frecuencias, del número de veces que aparece cada letra y se asigna a dicha letra una puntuación de 1 a 10 de acuerdo a su frecuencia de aparición, de forma que cuanto menos aparezca, mayor puntuación tendrá. Este valor será la cantidad (Puntos).

A modo de ejemplo para la anterior ejecución, el fichero salida.txt obtenido es:

#Letra	Cantidad	Puntos
A	16	1
B	2	3
C	6	3
D	5	2
E	10	1
F	2	4
G	2	2
H	1	4
I	9	1
J	1	8
L	5	1
M	4	3
N	7	1
O	10	1
P	3	3
Q	1	5
R	10	1
S	5	1
T	6	1
U	4	1
V	2	4
X	1	8
Y	1	4
Z	1	10

3.1.4. Fichero diccionario

El fichero diccionario se componen de un conjunto de palabras, cada una en un línea. Este conjunto de palabras serán las palabras que se consideren como válidas.

Un ejemplo de fichero diccionario es el siguiente:

```
a
aaronica
aaronico
ab
abab
ababillarse
ababol
abaca
abacera
abaceria
abacero
abacial
abaco
abad
abada
abadejo
abadenga
abadengo
```

3.1.5. Fichero Letras

Un ejemplo de fichero de letras es el que se muestra a la derecha.

El formato del fichero es el siguiente:

1. En primer lugar aparece una línea encabezada con el carácter **#** donde se describe las columnas del fichero (Letra Cantidad Puntos)
2. A continuación cada línea se corresponde con la información de una letra:
 - Valor de la letra
 - Número de veces que aparece la letra en la Bolsa de Letras
 - Puntos asignados a la letra.

#Letra	Cantidad	Puntos
A	12	1
E	12	1
O	9	1
I	6	1
S	6	1
N	5	1
L	1	1
R	6	1
U	5	1
T	4	1
D	5	2
G	2	2
C	5	3
B	2	3
M	2	3
P	2	3
H	2	4
F	1	4
V	1	4
Y	1	4
Q	1	5
J	1	8
X	1	8
Z	1	10

3.1.6. Módulos a desarrollar

Módulo Diccionario

Será necesario construir el T.D.A Diccionario. Un objeto del T.D.A. Diccionario almacena palabras de un lenguaje. El T.D.A Diccionario será representado como un **set** *instanciado a string* . Así en líneas generales el módulo Diccionario se detalla a continuación:

```
#ifndef __Diccionario_h__
#define __Diccionario_h__

#include <set>
class Dictionary {
private:

    set<string> words;

public:
    /**
     * @brief Constructor por defecto
     *
     * Crea un Dictionary vacío
     */
    Dictionary(){};
```

```

/**
 * @brief Destructor
 */
~Dictionary(){}

/**
 * @brief Limpia el Dictionary
 *
 * @post el diccionario queda con 0 palabras
 */
void clear();

/**
 * @brief Tamaño del diccionario
 *
 * @return Número de palabras guardadas en el diccionario
 */
unsigned int size() const;

/**
 * @brief Comprueba si el diccionario está vacío.
 * @return true si el diccionario está vacío, false en caso contrario
 */
bool empty() const { return words.size()== 0; }

/**
 * @brief Indica si una palabra esta en el diccionario o no.
 * Este método comprueba si una determinada palabra se encuentra o no en el diccionario
 * @param palabra: la palabra que se quiere buscar.
 * @return Booleano indicando si la palabra existe o no en el diccionario
 */
bool exists(const string &val);

/**
 * @brief Elimina una palabra del diccionario
 *
 * @param val Palabra a borrar del diccionario
 *
 * @return Booleano que indica si la palabra se ha borrado del diccionario
 */
bool erase(const string &val);

/**
 * @brief Sobrecarga del operador de entrada
 *
 * Permite leer las palabras de un fichero de texto e introducirlas en el
 * diccionario
 * @param is Flujo de entrada
 * @param dic Diccionario a rellenar
 * @return Flujo de entrada para poder encadenar el operador
 */
friend istream &operator>>(istream &is, Dictionary &dic);

```

```

/**
 * @brief Sobrecarga del operador de salida
 *
 * Permite imprimir el diccionario completo a un flujo de salida
 *
 * @param os Flujo de salida, donde imprimir el diccionario
 * @param dic Diccionario a imprimir
 * @return Flujo de salida, para poder encadenar el operador
 */
friend ostream &operator<<(ostream &os, const Dictionary &dic);

/**
 * @brief Indica el numero de apariciones de una letra.
 *
 *
 * @param c letra a buscar.
 * @return Un entero indicando el numero de apariciones.
 */
int getOccurrences(const char c)const;

/**
 * @brief Indica el numero totales de letras
 *
 *
 * @return Un entero indicando el numero totales de letras
 */

int getTotalLetters()const;

/**
 * @brief Iterador del diccionario. Obtiene las palabras ordenadas alfabéticamente
 */
class iterator {
private:
    ...

public:
    ...

    friend class Dictionary;
};

/**
 * @brief Iterador constante del diccionario. Obtiene las palabras ordenadas alfabéticamente
 */
class const_iterator {
private:
    set<string>::const_iterator it;

public:

    friend class Dictionary;
};

```

```

/**
 * @brief Indica si una palabra esta en el diccionario
 *
 * @return iterador apuntando a la palabra si esta o end si no esta
 */

iterator find(const string & w);

/**
 * @brief Obtiene todas las palabras de una longitud
 * param longitud: valor de longitud de la palabras a devolver
 * @return un vector con palabra de la longitud dada
 */
vector<string> getWordsLength(int longitud);

/**
 * @brief Inserta una palabra en el diccionario
 *
 * @param val palabra a insertar en el diccionario
 * @return Booleano que indica si la inserción ha tenido éxito. Una palabra se inserta
 * con éxito si no existía previamente en el diccionario. El iterador apunta a la palabra
 */
pair<iterator,bool> insert(const string &val);

/**
 * @brief Busca un las palabras con un prefijo
 *
 * @param val prefijo a buscar
 * @return un pair con dos iteradores el primero apuntando a la primera palabra con el prefijo y
 * el segundo donde ya no contiene el prefijo. Si no existe el prefijo se devuelve los dos
 * iteradores apuntando a end()
 */

pair<iterator, iterator> range_prefix(const string &val);

/**
 * @brief Obtiene el iterador apuntando a la primera palabra del diccionario
 */
iterator begin() ;
const_iterator begin()const;

/**
 * @brief Obtiene el iterador apuntando al final del diccionario.
 */
iterator end() ;
const_iterator end()const;

};

#endif

```

El iterador sobre el T.D.A Diccionario que nos permita recorrer de manera ordenada todas las

palabras del diccionario. Una posible especificación y representación de este iterador es:

```
#ifndef __Diccionario_h__
#define __Diccionario_h__
#include <set>

class Diccionario{
private:
    set<string> datos;
public:
    ...

    class iterator{
    private:
        set<string>::iterator it;
    public:
        iterator ();
        string operator *();
        iterator & operator ++();
        bool operator ==(const iterator &i)
        bool operator !=(const iterator &i)
        friend class Diccionario;
    };

    ...
};

#endif
```

Módulo Letra y Conjunto de Letras

El T.D.A Letra almacena una letra. Una letra se especifica con tres valores:

1. El carácter de la propia letra
2. La cantidad de veces que puede aparecer.
3. La puntuación de una letra.

El T.D.A Conjunto_Letras permitirá tener en memoria un fichero Letras. Este T.D.A se define como una colección de letras, en las que no hay letras repetidas.

Módulo Bolsa de Letras

Este módulo será útil para el programa *letras*. El T.D.A Bolsa_Letras almacena caracteres correspondientes a una letra de un Conjunto de Letras. Este carácter aparece en la Bolsa_Letras repetido tantas veces como diga el campo cantidad de la letra. Por lo tanto en la Bolsa de Letras aparecen las letras de forma aleatoria. En el programa “*letras*” la secuencia de letras con las que se juega se cogen de la Bolsa de Letras.

3.2 Cifras

3.2.1 Estructuras lineales o jerárquicas

Basándose en estructuras lineales ó no lineales, resolver el juego, incluyendo la búsqueda de combinaciones mágicas. Se han de definir adecuadamente los TDA utilizados.

4. Práctica a entregar

El estudiante deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre “cifrasyletras.tgz” y entregarlo antes de la fecha que se publicará en la página web de la asignatura. Tenga en cuenta que no se incluirán ficheros objeto, ni ejecutables, ni la carpeta datos. Es recomendable que haga una “limpieza” para eliminar los archivos temporales o que se puedan generar a partir de los fuentes.

El estudiante debe incluir el archivo CmakeLists.txt para realizar la compilación. Tenga en cuenta que los archivos deben estar distribuidos en directorios:

letras	— include	Ficheros de cabecera (.h)
	— src	Código fuente (.cpp)
	— obj	Código objeto (.o)
	— lib	Bibliotecas
	— doc	Documentación
	— bin	Ficheros ejecutables
	— datos	Fichero de datos.

Para realizar la entrega, en primer lugar, realice la limpieza de archivos que no se incluirán en ella, y sitúese en la carpeta superior (en el mismo nivel de la carpeta “cifrasyletras”) para ejecutar:

```
prompt% tar zcv cifrasyletras.tgz letras
```

tras lo cual, dispondrá de un nuevo archivo cifrasyletras.tgz que contiene la carpeta letras así como todas las carpetas y archivos que cuelgan de ella.

La práctica se debe realizar obligatoriamente (salvo excepciones claramente justificadas) por parejas. Ambos miembros de la pareja deberán subir los ficheros a prado a través del enlace correspondiente a la entrega de la práctica final. Se deberá incluir un fichero readme.txt en el que se indique exactamente qué se ha hecho y cómo se ha hecho (incluyendo referencias), y si hay algo que no funciona.

5. Valoración

Se podrá alcanzar una **máxima puntuación** en la práctica de **1 punto**. Son **2 partes independientes**, de forma que **si se hacen ambas, el máximo es 1 punto**. Si se hace sólo la parte de las letras, el máximo es **0.75 puntos** y si se hace sólo la parte de las cifras, el máximo es **0.5 puntos**. Si se usa IA vía ChatGpt o similares como soporte, la valoración máxima de la práctica (independientemente de qué partes de hagan) será **0.25 puntos**.

6. Referencias

- Rosa Rodriguez-Sánchez, J. Fdez-Valdivia, J.A. García, Estructuras de Datos en C++. Un enfoque

práctico, 2020

- Garrido, A., Fdez-Valdivia, J. “*Abstracción y estructuras de datos en C++*”. Delta publicaciones, 2006.
- Garrido, A. Estructuras de datos avanzadas: con soluciones en C++. Editorial Universidad de Granada. 2025