Trabajo Práctico 1
Texto Predictivo

Organización del Computador II

Primer Cuatrimestre 2014

1. Introducción

El objetivo de este trabajo práctico es implementar un conjunto de funciones sobre una estructura recursiva que representa un diccionario para predecir texto en teléfonos móviles "no inteligentes". Antiguamente, las redes sociales en internet eran apenas argumentos para la ciencia ficción y por consiguiente los mensajes de texto eran furor como medio de comunicación rápido, sencillo y portátil a través de los teléfonos móviles. Lamentablemente, estos teléfonos apenas contaban con un teclado dotado de poco más de diez teclas.

Desgraciadamente a algún demente tecnólogo se le ocurrió la brillante idea de que se podían utilizar esas diez teclas para poder escribir las palabras de cualquier diccionario formadas por la combinación de más de 35 caracteres alfanuméricos. La idea fue sencilla, asignar a cada una de las diez teclas, representadas antiguamente sólo por los números *0*, *1*, *2*, ..., *9*, un subconjunto de estos caracteres. 3 En la Figura 1 puede verse esta relación en uno de estos teclado "no inteligentes".

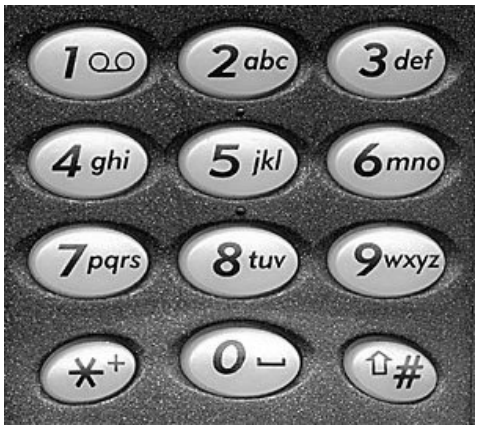

Figura 1: Ejemplo de uno de estos anticuados teclados.

La primera implementación de esta aplicación requería presionar varias veces la misma tecla hasta llegar al carácter deseado de ese subconjunto. Por ejemplo, para escribir la letra *c* había que presionar tres veces consecutivas la tecla *2*, no muy lentamente, con cuidado de no hacerlo demasiado rápidamente porque era probable pasarse de largo y ciclar en ese subconjunto de caracteres varias veces.

Pero los problemas de tendinitis en los pulgares opuestos en la población telefónicamente activa comenzaron a crecer. Entonces, al demente que estaba al lado del primero se le ocurrió que podría no hacer falta presionar varias veces cada tecla, sino, que con una vez que se presionara, alguna aplicación intentara adivinar a qué letra del alfabeto de ese subconjunto se quería referir.

De esta forma, para escribir la palabra *casa*, sería sólo necesario presionar por única vez las teclas *2*, *2*, *7* y *2*. Pero aquí aparece el problema en cuestión: con la combinación de las teclas *2*, *2*, *7* y *2* también se podrían escribir muchas otras palabras, como por ejemplo *cara*, *capa* y *abra*.

El **texto predictivo** se presentó como la solución a este problema. Esta aplicación funciona haciendo referencia a un diccionario donde las palabras ya se encuentran cargadas. Cuando el usuario presiona las teclas numéricas asociadas a cada subconjunto de caracteres, la aplicación busca en el diccionario la lista de palabras posibles de acuerdo con la combinación de teclas presionada y muestra todas las opciones, donde el usuario puede seleccionar la palabra deseada.

Posteriormente, esto se fue combinando con algoritmos de palabras más utilizadas para sugerir una de todas las combinaciones posibles, y con la predicción de palabras mediante sufijos.

La forma más común de implementar un diccionario para tal fin es utilizar una estructura de **Trie** (árbol de prefijos). En este tipo de árbol cada *nodo* está representado en realidad por una lista de nodos, donde la cantidad de elementos de esta lista es variable. Denominaremos a los elementos de esta lista como *nivel*, por comodidad. Además, cada uno de los nodos de la lista que forma un nivel posee como hijos a otra lista de nodos, otro *nivel*.

De esta forma, el primer nivel del **Trie** contiene todos las primeros caracteres de cada palabra almacenada en el **Trie**. Los segundos niveles del **Trie**, es decir, cada uno de los segundos niveles hijos de algún nodo del primer nivel, contienen todos las segundos caracteres que le siguen a cada caracter del primer nivel del cual son hijos. Y así podemos seguir, teniendo tantos niveles como la longitud de la palabra más larga que estemos almacenando.

A simple vista puede percibirse en esta estructura que aquellos nodos en un nivel que no posean otro nivel como hijo, representarán el fin de una secuencia de caracteres que formará un palabra. Pero, además, como nuestro idioma (y muchos otros) poseen palabras que son sufijos de otras, será necesario indicar en cada nodo si es el fin, o no, de una secuencia que representa una palabra válida del diccionario. En la Figura 2 podemos ver un ejemplo de esta situación.

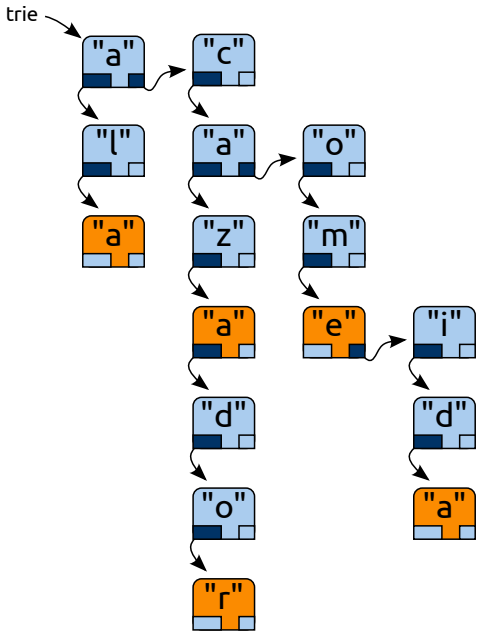

Figura 2: Ejemplo de Trie que contiene palabras que son sufijos de otras.

Las funciones a implementar permiten crear Tries, recorrerlos, imprimirlos y borrarlos. Además de estas operaciones, se deben programar funciones avanzadas para poder consultar por las palabras que contiene, realizar la predicción de texto y calcular el peso del **Trie**.

La mayor parte de los ejercicios a realizar para este trabajo práctico estarán divididos en dos secciones.

- **Primera** Completar las funciones que permiten manipular el tipo **Trie**.
- **Segunda** Realizar las funciones avanzadas sobre **Trie**.

Por último se deberá realizar un programa en `lenguaje C`, que cree determinados **Trie**s y ejecute algunas de las funciones de manipulación de los mismos.

1.1. Tipos trie y nodo

La estructura del **Trie** está compuesta por un bloque de memoria que contiene un puntero al nodo raíz del árbol. Cada nodo contiene un caracter que representa la letra dentro del árbol. Tiene un puntero al siguiente, su hermano derecho dentro del nivel (`null` si no tiene), cuya letra es estrictamente mayor a la de éste, es decir, poseen un orden lexicográfico dentro del nivel. También tiene un puntero al primero de sus hijos (siguientes posibles letras dentro del árbol). Finalmente, cuenta con un campo *booleano* que se utiliza para saber si una palabra termina allí.

```
typedef struct nodo_t {
    struct nodo_t *sig;
    struct nodo_t *hijos;
    char c;
    bool fin;
} __attribute__((packed)) nodo;
```

```
typedef struct trie_t {
    nodo *raiz;
} __attribute__((packed)) trie;
```

donde el tipo `bool` para indicar un valor de verdad, es un byte que vale 0 o 1, y está definido en `stdbool.h`

En la Figura 3 se puede ver un ejemplo de un **Trie** que contiene las palabras `{ala, alan, alas}`, utilizando estas estructuras.

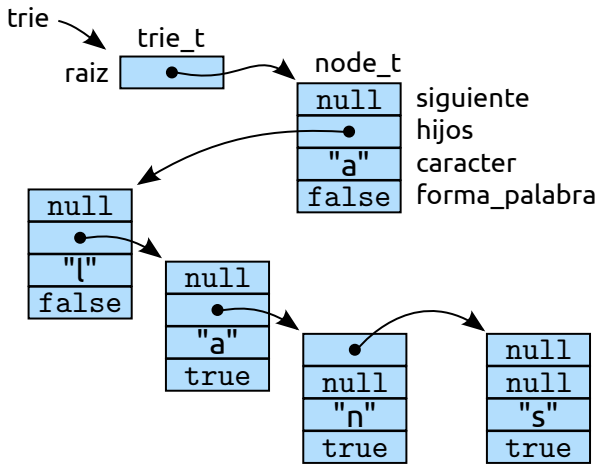

Figura 3: Estructura de datos del Trie con las palabras {`ala, alan, alas`}.

1.2. Funciones de **trie** y derivados

- `double peso_palabra(char *palabra)`
  Calcula el promedio de los caracteres de la palabra, según el valor de cada caracter en la tabla de códigos de caracteres ASCII.
  Nota: Esta función debe implementarse en `lenguaje C`.

- `trie *trie_crear(void)`
  Crea un **Trie** vacío.

- `void trie_borrar(trie *t)`
  Borra el **Trie** `self` y todos sus nodos internos.

- `nodo *nodo_crear(char c)`
  Crea un nodo nuevo cuyo contenido es caracter, sin siguiente ni hijo.

  El **Trie** no será sensible a mayúsculas y minúsculas. Tampoco a caracteres especiales por fuera de los números y las letras convencionales. Por lo tanto, al momento de definir el contenido del nodo, el caracter pasado por parámetro será convertido a minúscula si se encontraba en mayúscula.

  Además, si el caracter pasado por parámetro difiere de las letras y números permitidos, será convertido al caracter `'␣'`.

Los caracteres permitidos son:

- números = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'}
- letras = { 'a','b','c','d','e','f','g', 'h','i','j','k','l','m','n', 'o','p','q','r','s','t','u', 'v','w','x','y','z' }
- espacio = {'␣'}

- `nodo *insertar_nodo_en_nivel(nodo **nivel, char c)`
  Recibe un puntero que apunta al puntero al primer nodo de un nivel del **Trie** y verifica si el caracter pertenece a la lista de nodos del nivel.

  Si pertenece, devuelve el puntero al mismo, si no pertenece, crea un nuevo nodo con el caracter deseado y lo inserta correctamente en el lugar indicado respetando el orden lexicográfico de los caracteres en la lista de nodos, y devuelve el puntero al mismo.

- `void trie_agregar_palabra(trie *t, char *p)`
  Agrega la secuencia de caracteres (la palabra) al **Trie**, respetando el orden lexicográfico de los caracteres en cada nivel. En la Figura 4 se puede ver un ejemplo de **Trie** al cual se le van agregando varias palabras.

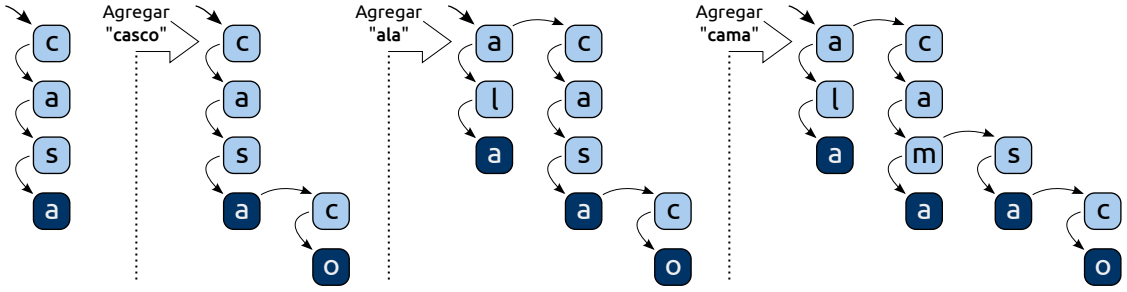

Figura 4: Ejemplo de Trie con la palabra `casa`, al que se le agregan consecutivamente las palabras `casco`, `ala` y `cama`.

- `trie *trie_construir(char *nombre_archivo)`
  Debe crear un **Trie** y agregar en él todas las palabras contenidas en el archivo pasado por parámetro. El archivo se debe abrir en modo `read`, de modo que no pueda ser modificado.

  Las palabras en el archivo se encuentran en la primera línea del mismo, separadas por un espacio. La longitud de cada palabra del archivo estará acotada por 1024 caracteres (incluyendo el caracter de finalización de string).

- `void trie_imprimir(trie *t, char *nombre_archivo)`
  Debe agregar al archivo pasado por parámetro una línea con el contenido de t. El archivo se debe abrir en modo `append`, de modo que las nuevas líneas sean adicionadas al archivo original. Como en `trie_construir`, cada palabrá se supondrá acotada por la longitud de 1024 caracteres (incluyendo el caracter de finalización de string).

  Debe imprimir cada palabra contenida en el **Trie**. Debe dejarse un espacio entre cada palabra. Debe haber un espacio luego de la última palabra. Antes de la primer palabra no debe haber un espacio. Al finalizar, luego de la última palabra, el espacio deberá estar seguido del caracter especial de fin de línea (`LF = 10`).

  Las palabras deben imprimirse según el mismo orden lexicográfico utilizado para almacenar los caracteres en los niveles. Si el **Trie** está vacío se deberá imprimir `<vacio>` seguido del caracter especial de fin de línea.

  Ejemplo: `trie_imprimir(t, "trie.txt")` para el **Trie** de la Figura 2 imprime en `trie.txt`:

  `ala`*[espacio]*`caza`*[espacio]*`cazador`*[espacio]*`come`*[espacio]*`comida`*[espacio]**[LF]*

  Ayuda: Si le sirve, piense cómo podría utilizar un caso especial de la función avanzada `palabras_con_prefijo` para implementar esta función.

1.3. Tipo listaP y funciones asociadas

Para poder seguir adelante con las funciones avanzadas de **Trie**, necesitaremos definir un tipo de datos auxiliar para manipular de manera sencilla las listas de palabras que usaremos. Una `listaP` será básicamente una secuencia de secuencia de `char`s. Sin embargo, como hicimos con el **Trie**, la definiremos como una secuencia de nodos, donde cada nodo contiene una palabra. La estructura de la **listaP** está compuesta de la siguiente manera:

```
typedef struct lsnodo_t {
    char *valor;
    struct lsnodo_t *sig;
} __attribute__((packed)) lsnodo;

typedef struct listaP_t {
    lsnodo *prim;
    lsnodo *ult;
} __attribute__((packed)) listaP;
```

Para manipular fácilmente una lista de palabras se utilizarán las siguientes funciones:

- `listaP *lista_crear()`
  Crea una nueva lista vacía.
- `void lista_agregar(listaP *ls, const char *palabra)`
  Agrega una copia de la palabra al final de la lista.
- `void lista_borrar(listaP *ls)`
  Borra la lista.
- `void lista_concatenar(listaP *xs, listaP *ys)`
  Concatena las listas `xs` e `ys` en `xs`, poniendo todos los elementos de `ys` al final de `xs`. Finalmente, borra sólo la estructura principal de la lista `ys`, no sus nodos.

Nota: No se han definido funciones para recorrer las palabras contenidas en una `listaP` pero, sin embargo, es posible recorrer la estructura interna de la lista de palabras.

1.4. Funciones avanzadas sobre Tries

- `bool buscar_palabra(trie *t, char *p)`
  Busca la palabra en el **Trie** y dice si pertenece.
- `double trie_pesar(trie *t, double (*funcion_pesaje)(char*))`
  Calcula el promedio de los valores de aplicar la función de pesaje, pasada por parámetro, a cada palabra contenida en el **Trie**.
- `listaP *palabras_con_prefijo(trie *t, char *prefijo)`
  Devuelve todas aquellas palabras contenidas en el **Trie** que posean como prefijo al pasado como parámetro. La lista de palabras devuelta también respeta el orden lexicográfico ya mencionado.
- `listaP *predecir_palabras(trie *t, char *teclas)`
  Predice (devuelve) todas aquellas palabras contenidas en el **Trie** que posean como prefijo todas las secuencias posibles que se pueden formar a partir de las teclas. Las secuencias de teclas pasada como parámetro es una lista formada por dígitos (*0*, *1*, *2*, ..., *9*) en algún orden.

  Es importante tener en cuenta todas las combinaciones posibles derivadas del subconjunto de caracteres que representa cada tecla.

  La lista de palabras devuelta también respeta el orden lexicográfico ya mencionado.

  Nota: Esta función debe implementarse en `lenguaje C`.
  Sugerencia: Usar la función auxiliar `caracteres_de_tecla` detallada más adelante.

1.5. Funciones Auxiliares Sugeridas

A continuación se presentan funciones auxiliares sugeridas para facilitar el desarrollo del trabajo práctico.

- `void nodo_borrar(nodo *n)`
  Borra el nodo pasado por parámetro, liberando la memoria.

- `nodo *nodo_buscar(nodo *n, char c)`
  Busca el nodo dentro del nivel cuyo contenido es c, a partir del nodo del nivel pasado como parámetro.

- `nodo *nodo_prefijo(nodo *n, char *p)`
  A partir del nodo del nivel pasado como parámetro, busca a través de los hijos al nodo donde queda representado el prefijo p.

- `const char *caracteres_de_tecla(char tecla)`
  Recibe un caracter representando un dígito (*0*, *1*, *2*, ..., *9*), y devuelve una lista con el subconjunto de caracteres que representa para el texto predictivo. Esta lista también respeta el orden lexicográfico ya mencionado. Según esta descripción, el resultado correspondiente a cada tecla es:

  ```
  caracteres_de_tecla('1')} ⟶ "1"
  caracteres_de_tecla('2')} ⟶ "2abc"
  caracteres_de_tecla('3')} ⟶ "3def"
  caracteres_de_tecla('4')} ⟶ "4ghi"
  caracteres_de_tecla('5')} ⟶ "5jkl"
  caracteres_de_tecla('6')} ⟶ "6mno"
  caracteres_de_tecla('7')} ⟶ "7pqrs"
  caracteres_de_tecla('8')} ⟶ "8tuv"
  caracteres_de_tecla('9')} ⟶ "9wxyz"
  caracteres_de_tecla('0')} ⟶ "0␣"
  ```

  donde ␣ representa el caracter de espacio.

- `void trie_agregar_palabras(trie *t, listaP *palabras, int n)`
  Agrega al **Trie** las n palabras contenidas en la lista de palabras, respetando el orden lexicográfico de los caracteres en cada nivel.

2. Enunciado

A lo largo del trabajo se deberán implementar un conjunto de funciones en `lenguaje ensamblador` y otras en `lenguaje C`.

2.1. Las funciones a implementar en lenguaje ensamblador son:

- `trie *trie_crear(void)` *[10 líneas]*
- `void trie_borrar(trie *t)` *[15 líneas]*
- `nodo *nodo_crear(char c)` *[15 líneas]*
- `nodo *insertar_nodo_en_nivel(nodo **nivel, char c)` *[35 líneas]*
- `void trie_agregar_palabra(trie *t, char *p)` *[20 líneas]*
- `trie *trie_construir(char *nombre_archivo)` *[45 líneas]*
- `void trie_imprimir(trie *t, char *nombre_archivo)` *[45 líneas]*
- `bool buscar_palabra(trie *t, char *p)` *[10 líneas]*
- `double trie_pesar(trie *t, double (*funcion_pesaje)(char*))` *[40 líneas]*
- `listaP *palabras_con_prefijo(trie *t, char *prefijo)` *[40 líneas]*

2.2. Las funciones a implementar en lenguaje C son:

- `double peso_palabra(char *palabra)` *[15 líneas]*
- `listaP *predecir_palabras(trie *t, char *teclas)` *[25 líneas]*

Las funciones auxiliares sugeridas

Para implementar todas las funciones anteriores de una manera concisa y fácil de entender se pueden implementar y utilizar también las funciones auxiliares sugeridas. Las mismas son opcionales, y queda a criterio de cada uno cómo utilizarlas para implementar las funciones anteriores.

- `void nodo_borrar(nodo *n)` *[15 líneas]*
- `nodo *nodo_buscar(nodo *n, char c)` *[15 líneas]*
- `nodo *nodo_prefijo(nodo *n, char *p)` *[30 líneas]*
- `const char *caracteres_de_tecla(char tecla)` *[15 líneas]*
- `void trie_agregar_palabras(trie *t, listaP *palabras, int n)` *[25 líneas]*

Nota:

En cada función a implementar se indica, con la referencia *[N líneas]*, la cantidad aproximada de líneas de código que fueron necesarias para resolver la misma según la solución de la cátedra.

2.3. Compilación y testeo

Para compilar el código, deberá ejecutar `make main` y `make tester` según corresponda.

2.3.1. Prueba corta

Deberá construir un programa de prueba (`main.c`) que realice las acciones detalladas a continuación. La idea del ejercicio es verificar manualmente que las funciones que vaya implementando funcionen correctamente. Corra el programa con

```
$ ./pruebacorta.sh
```

para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma. Recordar siempre borrar los tries y las listas luego de usarlos.

1- Crear un Trie nuevo vacío, imprimirlo y borrarlo. Para esto deberá implementar las funciones `trie_crear`, `trie_imprimir` y `trie_borrar` en ensamblador. No es necesario que implemente completamente imprimir y borrar, ya que no utiliza nodos, puede dejar eso para el siguiente punto.

2- Crear una Trie nuevo, agregarle una palabra, imprimirlo y borrarlo. Deberá implementar `nodo_crear`, `insertar_nodo_en_nivel` y `trie_agregar_palabra`, además ahora sí debe completar `trie_imprimir` y `trie_borrar`.

3- Crear una Trie nuevo, agregarle las palabras `casa`, `casco`, `ala` y `cama`, en ese orden, imprimirlo y borrarlo. Compararlo con el de la Figura 4. Deberá implementar `trie_construir`.

Por último, implemente las funciones avanzadas de Trie:

- En lenguaje ensamblador: `buscar_palabra`, `palabras_con_prefijo` y `trie_pesar`.
- En lenguaje C: `predecir_palabras` y `peso_palabra`.

2.3.2. Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código de manera automática.

Luego de compilar, puede ejecutar `./tests.sh` y eso correrá todos los tests de la cátedra con su código. Un test consiste en la creación, inserción, eliminación e impresión en archivo de una gran cantidad de tries. Luego de cada test, el script comparará los archivos generados por su TP con las soluciones correctas provistas por la cátedra.

También será probada la correcta administración de la memoria dinámica.

2.4. Archivos

Se entregan los siguientes archivos:

- `main.c`: Es el archivo principal donde escribir los ejercicios para la prueba corta.
- `tester.cpp`, `test-o-matic.cpp/h`: Son los archivos del tester. No debe modificarlos.
- `trie.h`: Contiene la definición de la estructura del Trie, los elementos y las funciones que debe completar.
- `trie_asm.asm`: Archivo a completar con su código en lenguaje ensamblador.
- `trie_c.c`: Archivo a completar con su código en lenguaje C.

- `listaP.h`: Contiene la definición de la estructura de la lista de palabras, los elementos y las funciones.
- `listaP.c`: Contiene la implementación de la lista de palabras.
- `Makefile`: Contiene las instrucciones para compilar el código.
- `tests.sh`: Es el script que corre todos los test con distintas entradas.
- `pruebacorta.sh`: Es el script que corre el test simple.

Notas:

a) Todas las funciones deben estar implementadas en `lenguaje ensamblador`, salvo las explícitamente indicadas a implementar en `lenguaje C`. Cualquier función extra, incluyendo las auxiliares sugeridas, deben estar implementadas en `lenguaje ensamblador`.

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para una correcta lectura de las palabras de los archivos utilice la función `fscanf` y el parámetro de formato `"%s"`. Tenga en cuenta los inconvenientes que podrían surgir dependiendo el uso que haga de esta función.

e) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

f) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests.

3. Resolución, informe y forma de entrega

Este trabajo práctico es de carácter **individual**. No deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado sólo los archivos `trie_asm.asm` y `trie_c.c`.

Este trabajo deberá entregarse como máximo a las 16:59:59 hs del día Martes 15 de Abril de 2014. ***La reentrega del trabajo, en caso de ser necesaria, deberá realizarse como máximo a las 16:59:59 hs del día Martes 13 de Mayo de 2014.***

Ambas entregas se realizarán a través de la página web. El sistema sólo aceptará entregas de trabajos hasta el horario de entrega, sin excepciones.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.