Entregado				-----	
Calificación					

- **Condición mínima de aprobación: Tener BIEN o BIEN- dos de los tres ejercicios.**
- **Se tendrá en cuenta en la calificación el ESTILO y la EFICIENCIA de los algoritmos.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **Puede entregarse en lápiz.**
- **No usar variables globales ni static.**
- **En caso de necesitar usar malloc o similar, asumir que nunca retornan NULL.**
- **No es necesario escribir los #include**
- **Escribir en cada hoja Nombre, Apellido, Legajo, Número de hoja, Total de hojas entregadas**

Ejercicio 1:

Dada una lista de enteros cuya definición de tipos es la siguiente:

```
typedef struct node * TList;

typedef struct node {
    int elem;
    struct node * tail;
} TNode;
```

Una lista vacía se representa con el valor NULL.

Escribir una función recursiva **deleteAll** que reciba únicamente dos listas de enteros ordenados en forma ascendente (sin repetidos) y elimine de la primera lista los elementos que pertenezcan a la segunda lista.

Ejemplo de invocación:

```
TList l;
TList m;

/* se insertan elementos en l y m */
...

l = deleteAll(l, m);
```

Ejemplo: si la primera lista tiene los elementos {1, 2, 3, 4, 5, 6, 7} y la segunda {0, 1, 3, 5, 10, 11, 14}, entonces en la primera lista deben quedar los elementos {2, 4, 6, 7}.

No definir funciones ni macros auxiliares.

Ejercicio 2:

Se desea generar un TAD que almacene un conjunto de elementos genéricos no repetidos. Para ello se propone el siguiente contrato:

setADT.h

```
typedef struct setCDT * setADT;

/* Retorna un nuevo conjunto de elementos genéricos. Al inicio está vacío */
setADT newSet(...);

/* Inserta una copia superficial (shallow copy) de elem siempre y cuando el elemento no esté
** en el conjunto. Retorna la cantidad total de elementos luego de agregar el elemento nuevo
*/
int add(setADT set, void * elem);

/* Retorna cuántos elementos hay en el conjunto */
int size(const setADT set);

/* Retorna una copia del mayor elemento del conjunto, NULL si no hay elementos */
void * max(const setADT set);

/* Retorna una copia del menor elemento del conjunto, NULL si no hay elementos */
void * min(const setADT set);

/* Retorna una copia del último elemento agregado en el conjunto, NULL si está vacío */
void * last(const setADT set);

/* Retorna un vector con todos los elementos del conjunto, ordenados en forma ascendente */
void * setToArray(const setADT set);
```

donde **...** en una lista de parámetros indica que usted (programador) debe definir cuáles son los parámetros necesarios para esa función, en base a las características del TAD.

Implementar el TAD completo

Para la correcta implementación, es fundamental que las funciones size, max, min y last sean lo más eficientes posibles.

Ejercicio 3:

Se desea escribir un TAD que de soporte a un juego del ahorcado. Para ello se cuenta con el siguiente contrato, que permite almacenar y seleccionar palabras, donde cada palabra tiene asociado un nivel de dificultad, que es un valor entero entre 1 y un nivel máximo determinado por el usuario.
ASUMIR QUE TODAS LAS PALABRAS CONTIENEN SOLO LETRAS EN MINUSCULAS.

hangmanADT.h

```
typedef struct hangmanCDT * hangmanADT;

/* Crea la estructura que dará soporte al almacenamiento y selección de palabras
** maxLevel: la cantidad máxima de niveles de dificultad que soportará (como mínimo 1)
** Los niveles válidos serán de 1 a maxLevel inclusive
 */
hangmanADT newHangman(unsigned int maxLevel);

/* Agrega un conjunto de palabras asociadas a un nivel de dificultad.
** El arreglo words[] está finalizado en NULL
** Si alguna de las palabras de words[] ya existe en el hangmanADT para ese nivel de dificultad
** se ignora.
** No se realiza una copia local de cada palabra sino únicamente los punteros recibidos
** Si el nivel supera la cantidad máxima definida en newHangman, se ignora y retorna -1
** Retorna cuántas palabras se agregaron al nivel
*/
int addWords(hangmanADT h, char * words[], unsigned int level);

/* Retorna cuántas palabras hay en un nivel, -1 si el nivel es inválido */
int size(const hangmanADT h, unsigned int level);

/* Retorna una palabra al azar de un nivel determinado, NULL si no hay palabras de ese nivel
** o si el nivel es inválido.
*/
char * word(const hangmanADT h, unsigned int level);

/* Retorna todas las palabras de un nivel, o NULL si el nivel es inválido
** El último elemento del vector es el puntero NULL
*/
char ** words(const hangmanADT h, unsigned int level);
```

Ejemplo de invocación:

```
hangmanADT h = newHangman(3); // Los niveles válidos serán 1, 2 y 3
size(h, 1); // -> Retorna 0
size(h, 4); // -> Retorna -1

char * firstWords[] = {"ingenieria", "informatica", NULL};
addWords(h, firstWords, 1); // -> Retorna 2
size(h, 1); // -> Retorna 2
word(h, 1); // -> Retorna "ingenieria" o "informatica"
words(h, 1); // -> Retorna {"ingenieria", "informatica", NULL}
             // También podria retornar {"informatica", "ingenieria",  NULL}

addWords(h, firstWords, 1); // -> Retorna 0

addWords(h, firstWords, 5); // -> Retorna -1
size(h, 5); // -> Retorna -1
word(h, 5); // -> Retorna NULL
words(h, 5); // -> Retorna NULL;
words(h, 2); // -> Retorna {NULL}

char * secondWords[] = {"programacion", NULL};
addWords(h, secondWords, 3); // -> Retorna 1
size(h, 3); // -> Retorna 1
word(h, 3); // -> Retorna "programacion"
words(h, 3); // -> Retorna {"programacion", NULL}


// Ejemplo que muestra cómo se copian las palabras en el TAD
char v[20] = "cazador";
char * thirdWords[] = { v, NULL};
addWords(h, thirdWords, 2); // -> Retorna 1

printf("%s\n", word(h, 2); // -> Imprime "cazador"

strcpy(v, "venado");  // En la dirección v ahora hay otro string

printf("%s\n", word(h, 2); // -> Imprime "venado"

char * lastWords[] = { "cazador", "colador", NULL};
addWords(h, lastWords, 2); // -> Retorna 2
```