

# Tarea 1 – Análisis de algoritmos y estructura de datos

Francisco José Sánchez Guerrero  
Universidad de Santiago de Chile,  
Santiago, Chile  
francisco.sanchez.g@usach.cl

## I. Introducción

Una empresa de juegos en línea está lanzando alternativas para acercar y motivar a los niños en el mundo de la lectura/escritura, por lo que está constantemente desarrollando juegos que combinan la lógica y la lectura. El fundamento es que los juegos lógicos lingüistas otorgan muchísimas ventajas, permiten el desarrollo del pensamiento lógico, relacionan la lectura/escritura con la diversión, desarrollan la autoestima al ir logrando los desafíos, incentivan la rapidez mental, ayudan a la flexibilidad y agilidad mental, estimulando el razonamiento inductivo (basado en patrones y tendencias) y deductivo (basado en hechos y reglas).

Se desea comercializar prontamente un juego de búsqueda de palabras para estudiantes de primer ciclo de educación básica, que consiste en buscar palabras en un tablero cuadrado, conectando vecinos diagonales, horizontales o verticales de una cierta ubicación en el tablero. En el juego, llamado Derecho Revés, las palabras se pueden leer en sentido normal o al revés (ver ejemplo de [Figura 1](#))

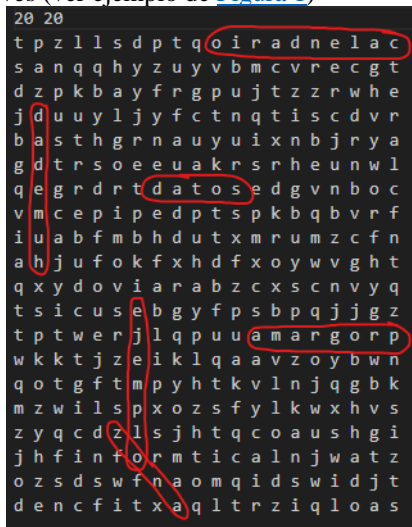


Figura 1: Ejemplo de tablero con palabras encontradas

Para apoyar la comprensión se solicita generar una simulación paso a paso de una posible solución, mostrando, por un lado, el tablero con cada palabra encontrada resaltada y la ubicación (en coordenadas) de todas las palabras encontradas al final.

Para resolver el problema presentado, se usará el lenguaje de programación C y se presentará un algoritmo que pueda recibir como entrada un archivo con la sopa de letras y la lista de palabras, y entregará la solución de esta misma.

Los archivos de entrada deben tener en la primera línea la cantidad de datos que se leerán. En el caso de la sopa de letras, debe tener la cantidad de filas y columnas correspondientes a esta, y en las palabras debe tener la cantidad de palabras que contiene.

## II. Solución propuesta

La solución que se plantea es ir buscando letra por letra a través de la sopa de letras para ver si la palabra existe dentro de alguna de las direcciones posibles y, si es así, agregaremos la solución a su archivo de salida correspondiente.

Para lograr esto, primero debemos extraer la información necesaria de los archivos de entrada correspondientes; para ello, utilizaremos las funciones `readFileMatrix` y `readFileWords`. `readFileMatrix` devolverá la sopa de letras, las filas y columnas de esta, mientras que la función `readFileWords` devolverá un arreglo con las palabras y el largo de este arreglo. Se decidió manejarlos en arreglos porque de esta manera se podrían recorrer fácilmente con ciclos `for`, y ya que las palabras son arreglos de caracteres, también facilitaba la solución del problema, como se presentará en los pseudocódigos; en este caso, quedará una matriz y un arreglo: uno de ellos almacenará la sopa de letras (matriz) y el otro

almacenará las palabras (arreglo). Luego, con la función `strtok` y `strcat`, crearemos los nombres de los archivos de salida de las palabras y la matriz. Después, abriremos y escribiremos un 0 en el archivo de salida de las palabras para iniciar el contador de las palabras que contendrá el archivo. Por último, pasaremos como argumentos la matriz, filas, columnas, el arreglo con las palabras, el tamaño del arreglo y los nombres de los archivos de salida a la función `findWords`, que es la encargada de buscar las soluciones posibles y escribirlas en sus respectivos archivos de salida.

A continuación, se presentarán los algoritmos principales en pseudocódigo que se encargarán de dar la solución al problema, aunque cabe decir que se presentará solo un ejemplo de las comparaciones que se hacen para encontrar la palabra dentro de la sopa de letras debido a que en esencia son todas las mismas; solo cambiará la suma y restas en los ejes  $x$  e  $y$ ."

## Algoritmos

`findWords` (Figura 2) hace lo siguiente: Con el primer `for` recorre el arreglo de las palabras para buscar si la palabra existe o no dentro de la matriz. Con el segundo y tercero, irá recorriendo la sopa de letras y si la primera letra de la palabra que está verificando coincide con alguna de la sopa de letras, manda a comprobar si la letra existe en alguna dirección.

```
findWords(matriz matriz, num filas, num columnas, arreglo array_words,
num size_array_words, palabra list_name, palabra matrix_name)

para k ← 1 hasta size_array_words
    para y ← 1 hasta columnas
        para x ← 1 hasta filas
            si matriz[y][x] = array_words[0][k] entonces
                comprobacion_right(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_upper_right(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_upper(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_upper_left(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_left(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_down_left(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_down(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
                comprobacion_down_right(matriz, filas, columnas, array_words[k], y, x,
list_name, matrix_name)
```

Figura 2: Algoritmo de `findWords` con complejidad  $O(n^5)$

`comprobationRight` (Figura 3) hace lo siguiente: Primero, copia la sopa de letras con la función `copyArray` (Figura 4) para no modificar la original y evitar problemas a la hora de reemplazar la palabra por un '\*' dentro de la misma. Si cambiamos la matriz original, al pasar esta matriz con otra palabra, los '\*'

reemplazados con la anterior palabra aparecerán, lo que podría provocar un conflicto a la hora de detectar las demás palabras e incluso al pasar la sopa de letras solucionadas. El ciclo `for` lo que hace es ir recorriendo la palabra para comparar cada letra dentro de la dirección correspondiente en la sopa de letras. Luego, tenemos dos verificaciones: si se termina el ciclo `for` recorriendo toda la palabra, significa que esta existe en esa dirección en la sopa de letras, y se manda a escribir tanto la palabra (`writeWord`) como la sopa de letras (`writeMatrix`) a sus archivos de salida correspondientes; ya que en la segunda verificación se comprueba que, si la palabra en la dirección correspondiente no es la misma, se regresa a `findWords`. Por último, mientras recorremos la palabra y las letras coinciden en la sopa de letras, se irá cambiando por un '\*' para tener la sopa de letras con la palabra reemplazada cuando se cumpla la primera verificación.

```
comprobation_right(matriz matriz, num filas, num columnas, string word, num y, num x, palabra list_name,
palabra matrix_name)

copy_matrix ← copiar_matriz(matriz)
len_word ← calcular_len_word(word);
y_aux ← y
x_aux ← x

para i ← 1 hasta len_word paso 1
    si i = len_word y comprobacion_word(list_name, word) entonces
        write_word(word, y_aux, x_aux, list_name)
        write_matrix(copy_matrix, filas, columnas, matrix_name)

    si x < 0 o x >= columnas o y < 0 o y >= filas o copy_matrix[y][x] != word[i] entonces
        liberar(copy_matrix)
        para i ← 1 hasta columnas
            liberar(copy_matrix[i])
        devolver

    copy_matrix[y][x] ← '*'
    x ← x + 1 ...aquí es donde varía las diferentes comparaciones en la suma o resta de los ejes

    liberar(copy_matrix)
    para i ← 1 hasta columnas
        liberar(copy_matrix[i])
```

Figura 3: Algoritmo de `comprobationRight` con complejidad  $O(n^2)$

`copyArray` (Figura 4) hace lo siguiente: Primero, reserva memoria para una matriz auxiliar de filas x columnas (filas y columnas dados como argumentos). Luego, recorre la matriz dada para copiar todos los caracteres a la matriz auxiliar y entregarla. De esta manera, al momento de cambiar los caracteres de la matriz a un '\*', no se altera la matriz original.

```

copyArray(matriz array, num filas, num columnas)
    aux = create_matrix(filas, columnas)

    para i ← 1 hasta filas
        para j ← 1 hasta columnas
            aux[i][j] = array[i][j]

    devolver(aux)

```

Figura 4: Algoritmo de copyArray con complejidad  $O(n^2)$

writeWord (Figura 5) hace lo siguiente: Primero, abre el archivo de salida de las palabras y verifica que no sea nulo y que haya un número válido para reservar memoria. Si es así, lo guarda en currentCount. Luego, aumenta en 1 currentCount para indicar que se agregó una nueva palabra. Por último, se setea el nuevo contador al inicio del archivo y se agrega la nueva palabra al final del archivo.

```

write_word(palabra word, num y, num x, palabra list_name)
    archivo ← abrirArchivo(list_name, "r")
    current_count ← 0
    si (file = nulo) entonces
        escribir("Error al abrir el archivo de la sopa de letras")
        salir(1)

    si (leerArchivo(archivo, "%d", current_count) <> 1)
        escribir("Error al leer la cantidad actual de elementos")
        cerrarArchivo(archivo)
        salir(1)

    current_count ← current_count + 1

    mirarArchivo(archivo, 0, SEEK_SET) ... aqui se mira el primero elemento del archivo
    escribirArchivo(archivo, "%d", current_count) ... aqui se setea el nuevo contador

    mirarArchivo(archivo, 0, SEEK_END) ... aqui se mira el ultimo elemento del archivo
    escribirArchivo(archivo, "%d %d (%s)", y+1, x+1, word)

    cerrarArchivo(archivo)

```

Figura 5: Algoritmo de writeWord con complejidad  $O(1)$

writeMatrix (Figura 6) hace lo siguiente: Primero, abre y crea al mismo tiempo el archivo de salida de la sopa de letras y luego escribe en el archivo toda la matriz dada en el argumento.

```

write_matrix(matriz matrix, num filas, num columnas, palabra matrix_name)
    archivo ← abrirArchivo(matrix_name, "a")

    para i ← 1 hasta columnas
        para j ← 1 hasta filas
            escribirArchivo(archivo, "%c", matrix[i][j])
            escribirArchivo(archivo, " ")

    escribirArchivo(archivo, "\n")
    cerrarArchivo(archivo)

```

Figura 6: Algoritmo de copyArray con complejidad  $O(n^2)$

### III. Resultados y análisis

Para poder observar el comportamiento de la solución propuesta, se evaluarán los diferentes

tiempos que presentará el computador al ejecutar el algoritmo frente a diversas entradas. Dado que tenemos dos entradas (sopa de letras y las palabras a buscar), lo que evaluaremos será cuánto se demora el algoritmo en encontrar todas las palabras en la sopa de letras de 50x50 (tablero2). Las entradas consistirán en la cantidad de palabras a resolver en este mismo; por ejemplo, cuánto se demora en encontrar 4, 6, 8, 10 palabras y así sucesivamente hasta llegar a las 20 palabras solicitadas oficialmente.

Entradas (palabras)	Tiempos
2	0,013
4	0,024
6	0,036
8	0,048
10	0,058
12	0,070
14	0,086
16	0,098
18	0,11
20	0,125

Tabla 1: Tabla de tiempos de ejecución.

Como se puede apreciar en la tabla anterior, a medida que la cantidad de palabras buscadas aumenta, también aumenta el tiempo de ejecución exponencialmente. Por ende, es concordante con la complejidad del algoritmo. A continuación, se mostrará un gráfico para ver esto de mejor manera.



Si bien puede parecer, en este caso, que es lineal, ya que falta muestra de datos; por ejemplo, se probó con una matriz de 100x100 y con 50 palabras a encontrar, y se demoró 3,562 segundos en encontrar todas las soluciones, como se puede ver a continuación:

```
Se pudo leer el archivo de la sopa de letras correctamente
Se pudo leer correctamente el archivo de las palabras
El tiempo que se demora en encontrar todas las palabras es: 3.562000
Se pudo completar la búsqueda exitosamente
numero de palabras buscadas: 50
numero de palabras encontradas: 7
Los archivos de salida son: sopa.out y palabras.out
```

Por ende, cada vez que se ingresan datos mayores, se va incrementando la cantidad de tiempo requerida para la solución y se va acercando a la complejidad de  $O(n^5)$  planteada en el algoritmo.

#### IV. Conclusiones

Se utilizaron estructuras de datos como los arreglos, debido a que por su linealidad se facilitaba el hecho de ir recorriéndolo para ir consiguiendo o buscando la información que se necesitaba.

La solución, con los suficientes datos de entrada, puede llegar a generar tiempos de ejecución muy elevados, debido a que depende totalmente del tamaño de la sopa de letras y la cantidad de palabras a encontrar. Sin embargo, mientras los datos en los archivos sean correctos, siempre se llegará a la solución mientras los recursos de la máquina lo permitan. Una posible solución a esto es combinar las comprobaciones en una sola función, así ocupando menos memoria a la hora de reservar memoria para la matriz solución o encontrar una mejor manera de recorrer y encontrar las palabras sin tantos ciclos for.

Por último, los archivos de salida son correspondientes con lo pedido, dando dos archivos de salida, uno que contenga las palabras encontradas y otro que contenga las matrices soluciones asociadas a las palabras.

#### V. Anexos

##### Manual de usuario

El programa crea las soluciones a partir de los archivos de entrada; por eso, primero debemos verificar que los archivos estén dentro de la misma carpeta y que respeten el formato apropiado (la primera línea debe contener la cantidad de datos).

```
1  20 20      1  10
2  t p z l    2  programa
```

Tablero1.ini

Lista1.lst

Para ejecutar el programa, se deben seguir los siguientes pasos:

1. Compilar:
  - gcc tarea01.c -o salida
2. Ejecutar el archivo de salida con las entradas correspondientes
  - a. Windows: salida tablero1.ini lista1.lst
  - b. Linux: ./salida tablero1.ini lista1.lst

Donde tablero1.ini es el archivo donde estará la sopa de letras a solucionar, y lista1.lst es el archivo que contiene las palabras a buscar.

Ejemplo:

1.-

```
gcc tarea01.c -o salida
```

2.-

Windows

```
salida tablero1.ini lista1.lst
```

Linux


```
./salida tablero1.ini lista1.lst
```


Hay que recordar que tablero1.ini en la primera línea debe tener la cantidad de columnas y filas separadas por un espacio y lista1.lst debe tener la cantidad de palabras a buscar.

Salida:

```
Se pudo leer el archivo de la sopa de letras correctamente
Se pudo leer correctamente el archivo de las palabras
El tiempo que se demora en encontrar todas las palabras es: 0.019000
Se pudo completar la búsqueda exitosamente
numero de palabras buscadas: 10
numero de palabras encontradas: 6
Los archivos de salida son: tablero1.out y lista1.out
```

Archivos de salida:

 lista1.out

 tablero1.out