

**Algoritmia**

# **Practica 4:**

## **Algoritmos de Búsqueda**

**David Piñuel Bosque**



2023

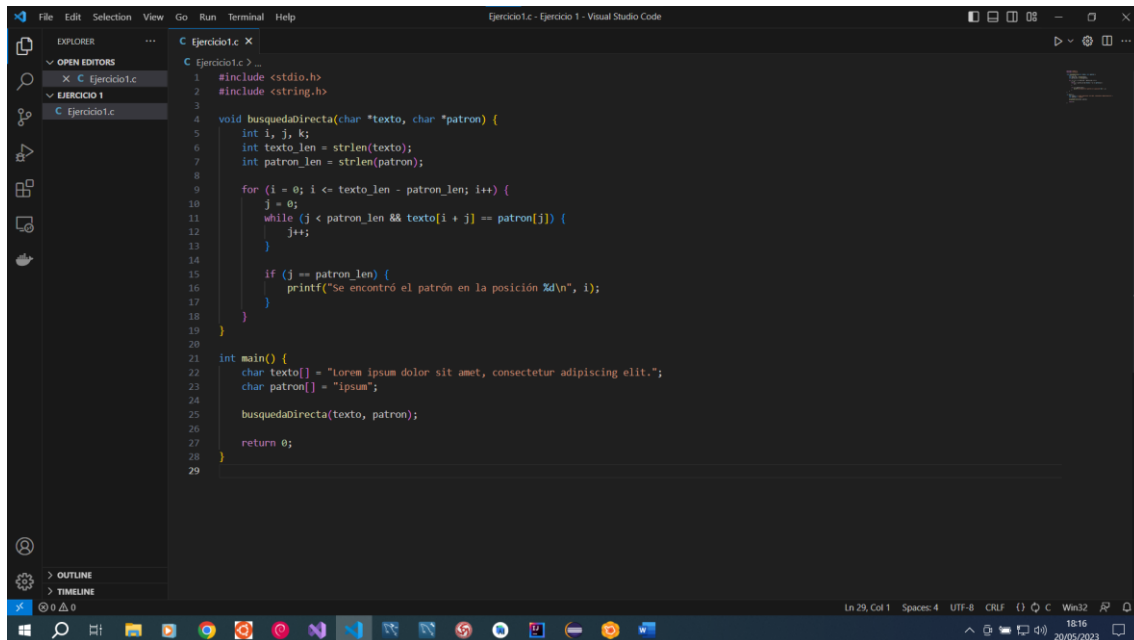
# Índice

1. Código Fuente del Programa de algoritmos de búsqueda.....	3
2. Respuesta ejercicio 2. ....	8
3. Ejercicio Opcional. ....	10
4. Aclaraciones y comentarios. ....	13

# Índice Ilustración

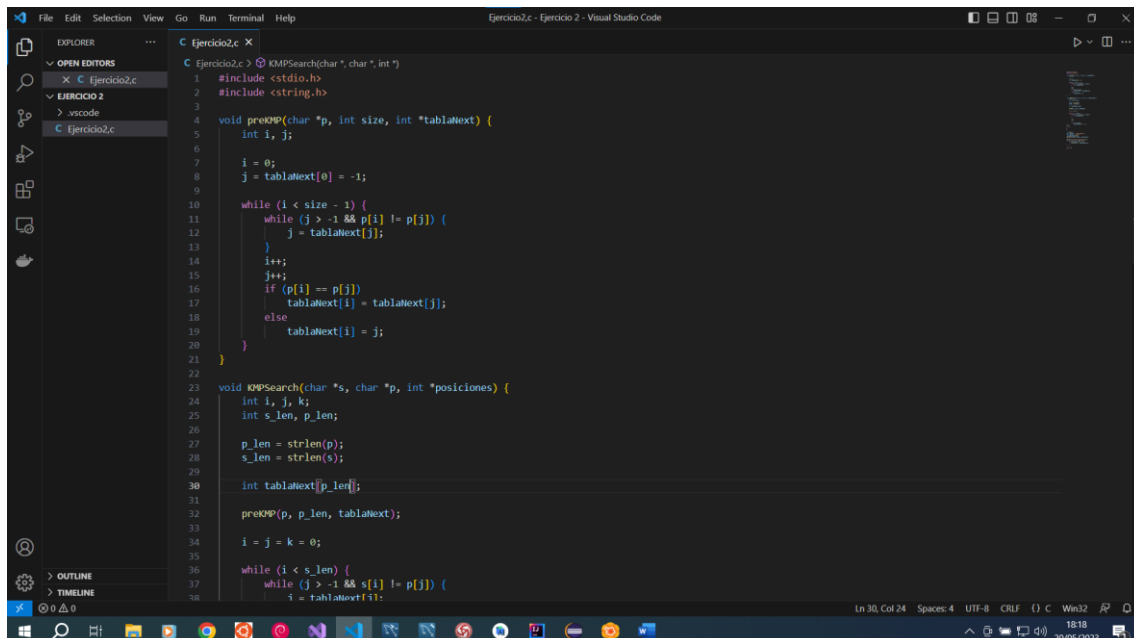
<i>Ilustración 1: Código Fuente Ejercicio 1 Parte 1 .....</i>	<i>3</i>
<i>Ilustración 2: Código Fuente Ejercicio 2 Parte 1 .....</i>	<i>3</i>
<i>Ilustración 3: Código Fuente Ejercicio 2 Parte 2 .....</i>	<i>4</i>
<i>Ilustración 4: Código Fuente Ejercicio 3 Parte 1 .....</i>	<i>4</i>
<i>Ilustración 5: Código Fuente Ejercicio 4 Parte 1 .....</i>	<i>5</i>
<i>Ilustración 6: Código Fuente Ejercicio 4 Parte 2 .....</i>	<i>5</i>
<i>Ilustración 7: Código Fuente Ejercicio 4 Parte 3 .....</i>	<i>6</i>
<i>Ilustración 8: Código Fuente Ejercicio 4 Parte 4 .....</i>	<i>6</i>
<i>Ilustración 9: Código Fuente Ejercicio 4 Parte 5 .....</i>	<i>7</i>
<i>Ilustración 10: Código Fuente Ejercicio 4 Parte 6 .....</i>	<i>7</i>

## 1. Código Fuente del Programa de algoritmos de búsqueda.



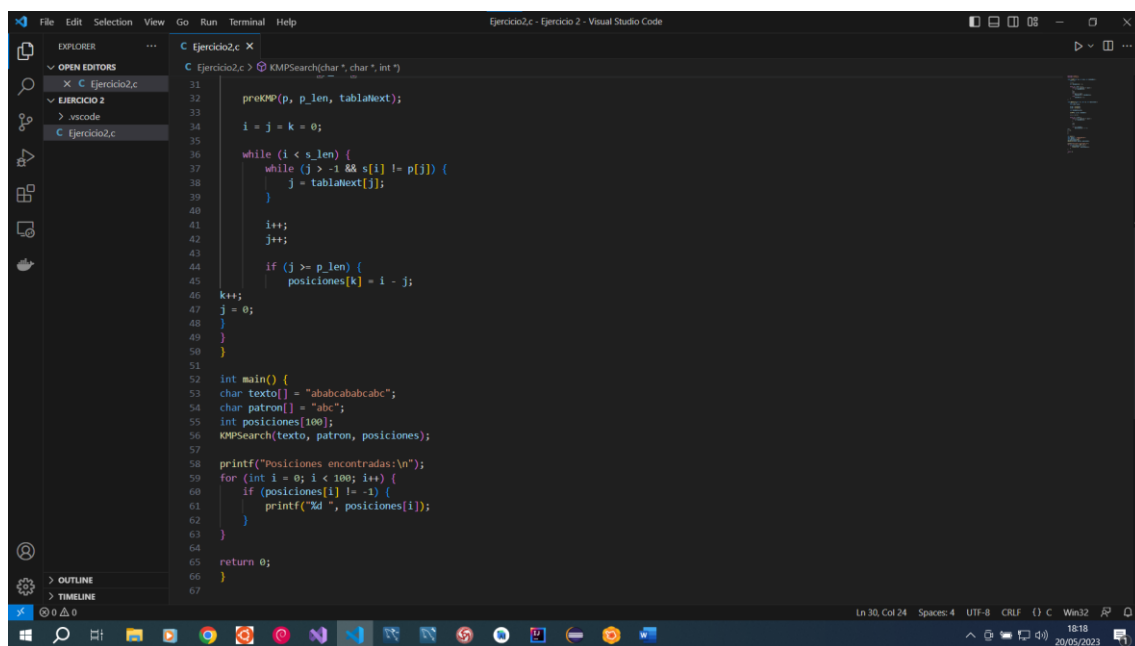
```
1 #include <stdio.h>
2 #include <string.h>
3
4 void busquedaDirecta(char *texto, char *patron) {
5     int i, j, k;
6     int texto_len = strlen(texto);
7     int patron_len = strlen(patron);
8
9     for (i = 0; i <= texto_len - patron_len; i++) {
10         j = 0;
11         while (j < patron_len && texto[i + j] == patron[j]) {
12             j++;
13         }
14
15         if (j == patron_len) {
16             printf("Se encontró el patrón en la posición %d\n", i);
17         }
18     }
19 }
20
21 int main() {
22     char texto[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
23     char patron[] = "ipsum";
24
25     busquedaDirecta(texto, patron);
26
27     return 0;
28 }
29
```

Ilustración 1: Código Fuente Ejercicio 1 Parte 1



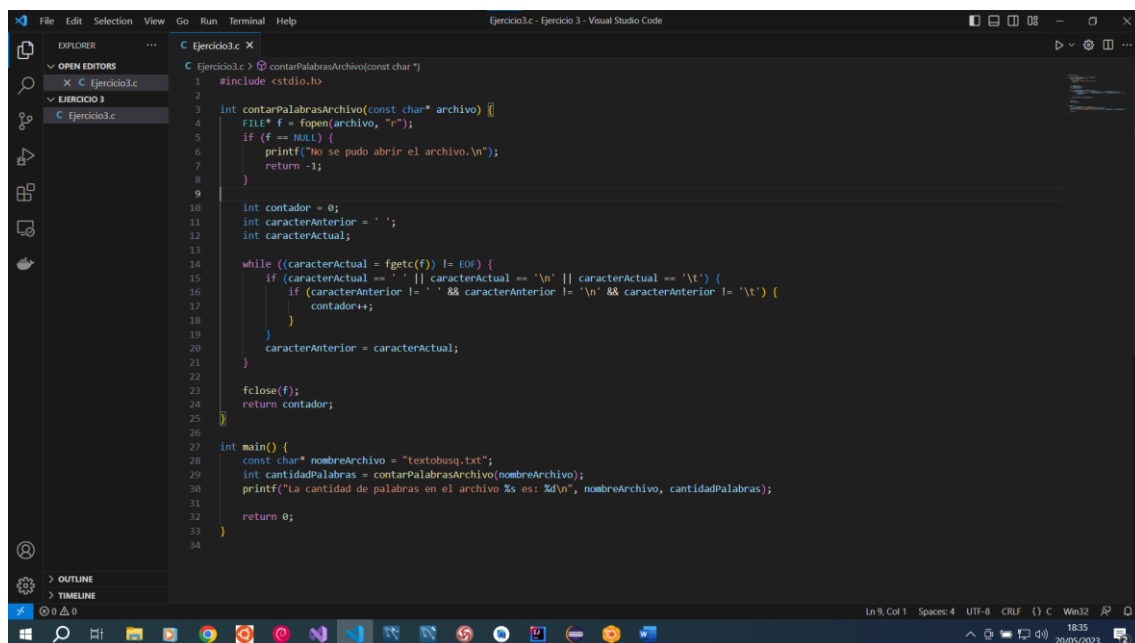
```
1 #include <stdio.h>
2 #include <string.h>
3
4 void preKMP(char *p, int size, int *tablaNext) {
5     int i, j;
6
7     i = 0;
8     j = tablaNext[0] = -1;
9
10    while (i < size - 1) {
11        while (j > -1 && p[i] != p[j]) {
12            j = tablaNext[j];
13        }
14        i++;
15        j++;
16        if (p[i] == p[j]) {
17            tablaNext[i] = tablaNext[j];
18        } else {
19            tablaNext[i] = j;
20        }
21    }
22 }
23
24 void KMPsearch(char *s, char *p, int *posiciones) {
25     int i, j, k;
26     int s_len, p_len;
27
28     p_len = strlen(p);
29     s_len = strlen(s);
30
31     int tablaNext[p_len];
32
33     preKMP(p, p_len, tablaNext);
34
35     i = j = k = 0;
36
37     while (i < s_len) {
38         while (j > -1 && s[i] != p[j]) {
39             j = tablaNext[j];
40         }
41         if (s[i] == p[j]) {
42             j++;
43         }
44         if (j == p_len) {
45             posiciones[k] = i - p_len + 1;
46             k++;
47             i = i - p_len + 1;
48             j = 0;
49         }
50         i++;
51     }
52 }
```

Ilustración 2: Código Fuente Ejercicio 2 Parte 1



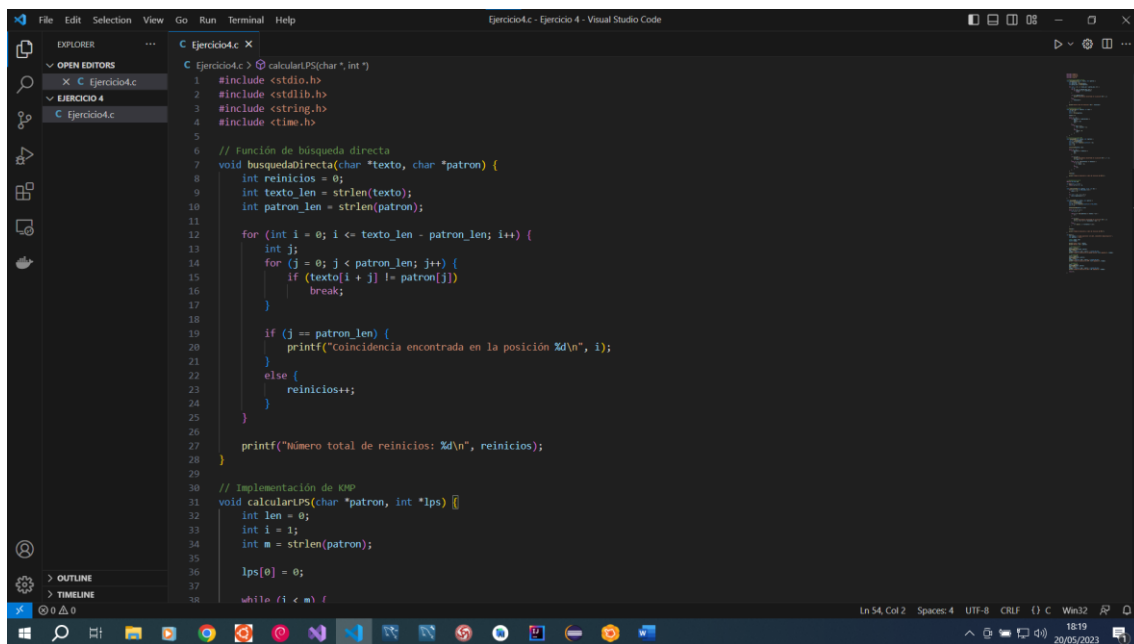
```
1  File Edit Selection View Go Run Terminal Help
2  Ejercicio2.c - Ejercicio 2 - Visual Studio Code
3
4  Explorer
5  OPEN EDITORS
6  Ejercicio2.c
7  Ejercicio 2
8  .vscode
9  Ejercicio2.c
10
11  C Ejercicio2.c
12  31 preKMP(p, p_len, tablaNext);
13  32
14  33 i = j = k = 0;
15  34
16  35 while (i < s_len) {
17  36     while (j > -1 && s[i] != p[j]) {
18  37         j = tablaNext[j];
19  38     }
20  39
21  40     i++;
22  41     j++;
23  42
24  43     if (j >= p_len) {
25  44         posiciones[k] = i - j;
26  45         k++;
27  46     }
28  47     j = 0;
29  48 }
30  49
31  50
32  51
33  52 int main() {
34  53     char texto[] = "abababababab";
35  54     char patron[] = "abc";
36  55     int posiciones[100];
37  56     KMPSearch(texto, patron, posiciones);
38  57
39  58     printf("Posiciones encontradas:\n");
40  59     for (int i = 0; i < 100; i++) {
41  60         if (posiciones[i] != -1) {
42  61             printf("%d ", posiciones[i]);
43  62         }
44  63     }
45  64
46  65     return 0;
47  66 }
48  67
```

Ilustración 3: Código Fuente Ejercicio 2 Parte 2



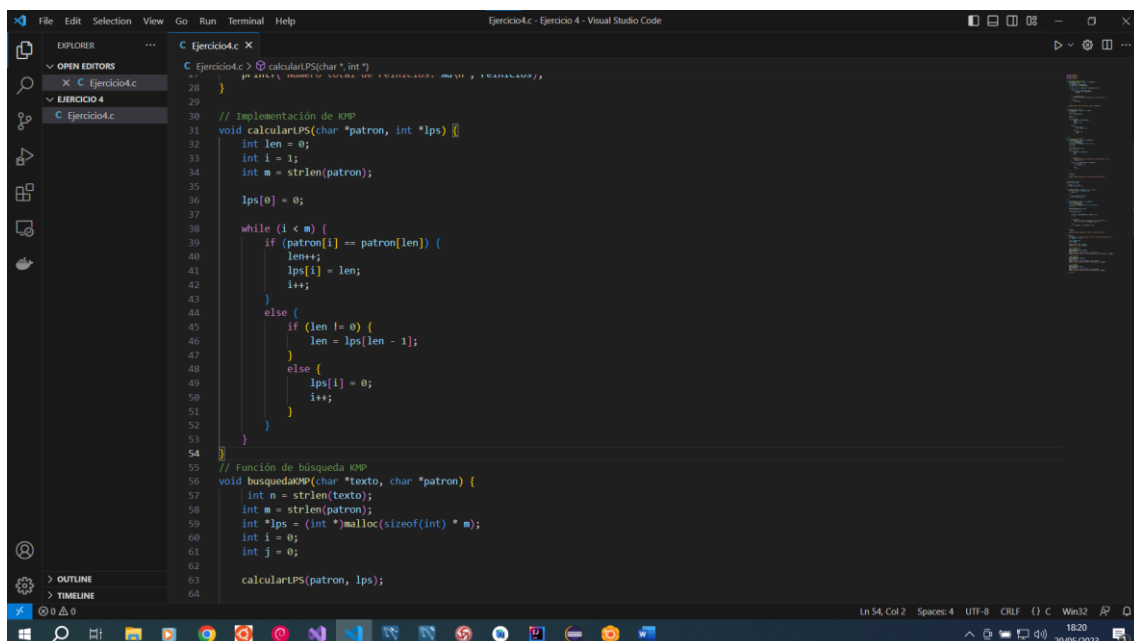
```
1  File Edit Selection View Go Run Terminal Help
2  Ejercicio3.c - Ejercicio 3 - Visual Studio Code
3
4  Explorer
5  OPEN EDITORS
6  Ejercicio3.c
7  Ejercicio 3
8  Ejercicio3.c
9
10  C Ejercicio3.c
11  1  #include <stdio.h>
12  2
13  3  int contarPalabrasArchivo(const char* archivo) {
14  4      FILE* f = fopen(archivo, "r");
15  5      if (f == NULL) {
16  6          printf("No se pudo abrir el archivo.\n");
17  7          return -1;
18  8      }
19  9
20  10     int contador = 0;
21  11     int caracterAnterior = ' ';
22  12     int caracterActual;
23  13
24  14     while ((caracterActual = fgetc(f)) != EOF) {
25  15         if (caracterActual == '\n' || caracterActual == '\t') {
26  16             if (caracterAnterior != ' ' && caracterAnterior != '\n' && caracterAnterior != '\t') {
27  17                 contador++;
28  18             }
29  19             caracterAnterior = caracterActual;
30  20         }
31  21     }
32  22
33  23     fclose(f);
34  24     return contador;
35  25 }
36  26
37  27 int main() {
38  28     const char* nombreArchivo = "textobusq.txt";
39  29     int cantidadPalabras = contarPalabrasArchivo(nombreArchivo);
40  30     printf("La cantidad de palabras en el archivo %s es: %d\n", nombreArchivo, cantidadPalabras);
41  31
42  32     return 0;
43  33 }
44  34
```

Ilustración 4: Código Fuente Ejercicio 3 Parte 1



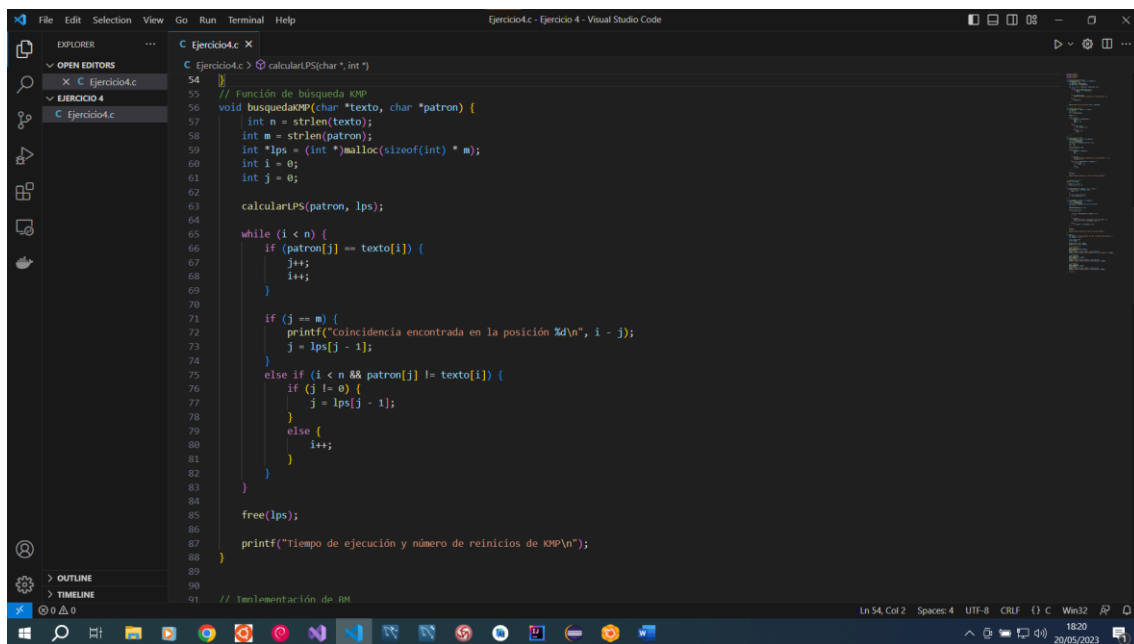
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <time.h>
5
6 // Función de búsqueda directa
7 void busquedaDirecta(char *texto, char *patron) {
8     int reinicios = 0;
9     int texto_len = strlen(texto);
10    int patron_len = strlen(patron);
11
12    for (int i = 0; i <= texto_len - patron_len; i++) {
13        int j;
14        for (j = 0; j < patron_len; j++) {
15            if (texto[i + j] != patron[j])
16                break;
17        }
18
19        if (j == patron_len) {
20            printf("Coincidencia encontrada en la posición %d\n", i);
21        }
22        else {
23            reinicios++;
24        }
25    }
26
27    printf("Número total de reinicios: %d\n", reinicios);
28 }
29
30 // Implementación de KMP
31 void calcularLPS(char *patron, int *lps) {
32     int len = 0;
33     int i = 1;
34     int m = strlen(patron);
35     lps[0] = 0;
36     while (i < m) {
```

Ilustración 5: Código Fuente Ejercicio 4 Parte 1



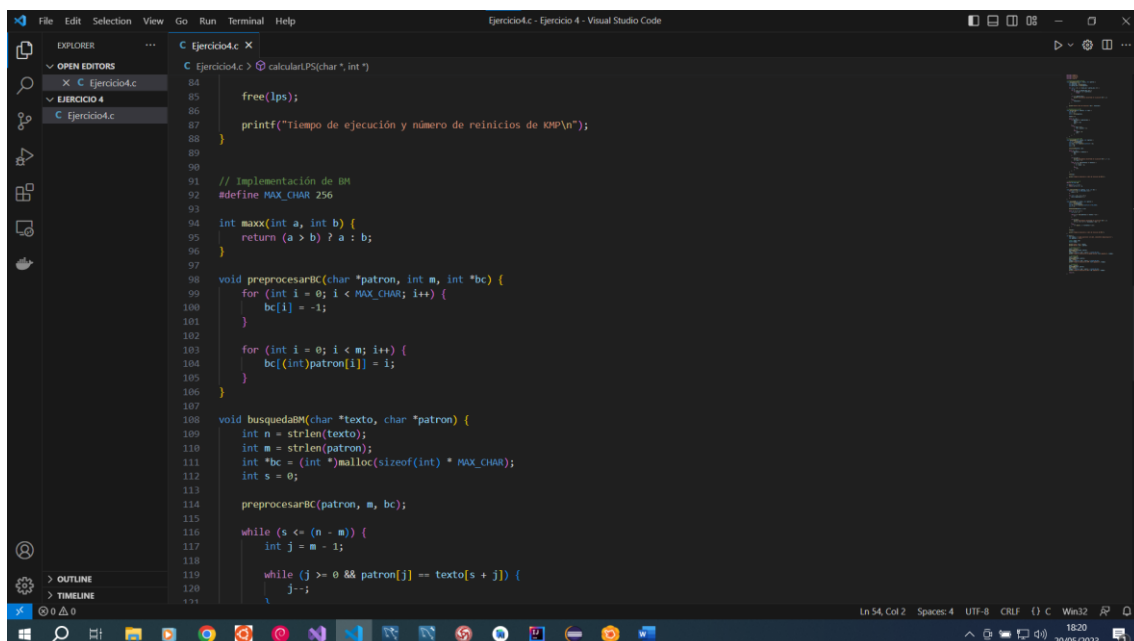
```
37     if (patron[i] == patron[len]) {
38         len++;
39         lps[i] = len;
40         i++;
41     }
42     else {
43         if (len != 0) {
44             len = lps[len - 1];
45         }
46         else {
47             lps[i] = 0;
48             i++;
49         }
50     }
51 }
52
53 // Función de búsqueda KMP
54 void busquedaKMP(char *texto, char *patron) {
55     int n = strlen(texto);
56     int m = strlen(patron);
57     int *lps = (int *)malloc(sizeof(int) * m);
58     int i = 0;
59     int j = 0;
60
61     calcularLPS(patron, lps);
62 }
```

Ilustración 6: Código Fuente Ejercicio 4 Parte 2



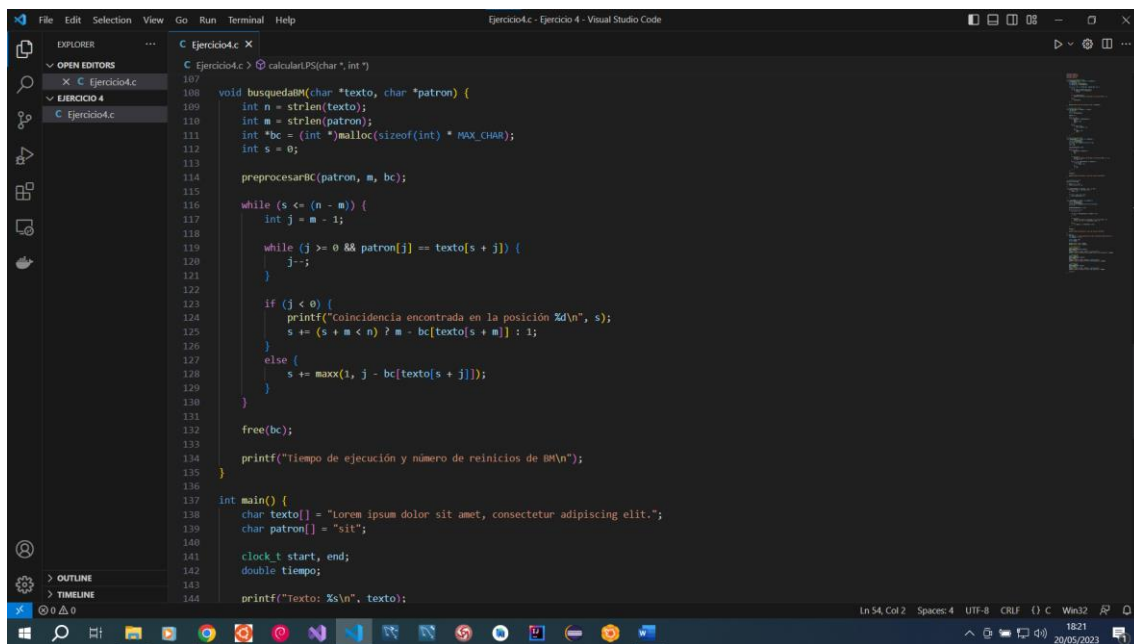
```
54 }
55 // Función de búsqueda KMP
56 void busquedaKMP(char *texto, char *patron) {
57     int n = strlen(texto);
58     int m = strlen(patron);
59     int *lps = (int *)malloc(sizeof(int) * m);
60     int i = 0;
61     int j = 0;
62
63     calcularPS(patron, lps);
64
65     while (i < n) {
66         if (patron[j] == texto[i]) {
67             j++;
68             i++;
69         }
70
71         if (j == m) {
72             printf("Coincidencia encontrada en la posición %d\n", i - j);
73             j = lps[j - 1];
74         }
75         else if (i < n && patron[j] != texto[i]) {
76             if (j != 0) {
77                 j = lps[j - 1];
78             }
79             else {
80                 i++;
81             }
82         }
83     }
84
85     free(lps);
86
87     printf("Tiempo de ejecución y número de reinicios de KMP\n");
88 }
89
90 // Implementación de RM
```

Ilustración 7: Código Fuente Ejercicio 4 Parte 3



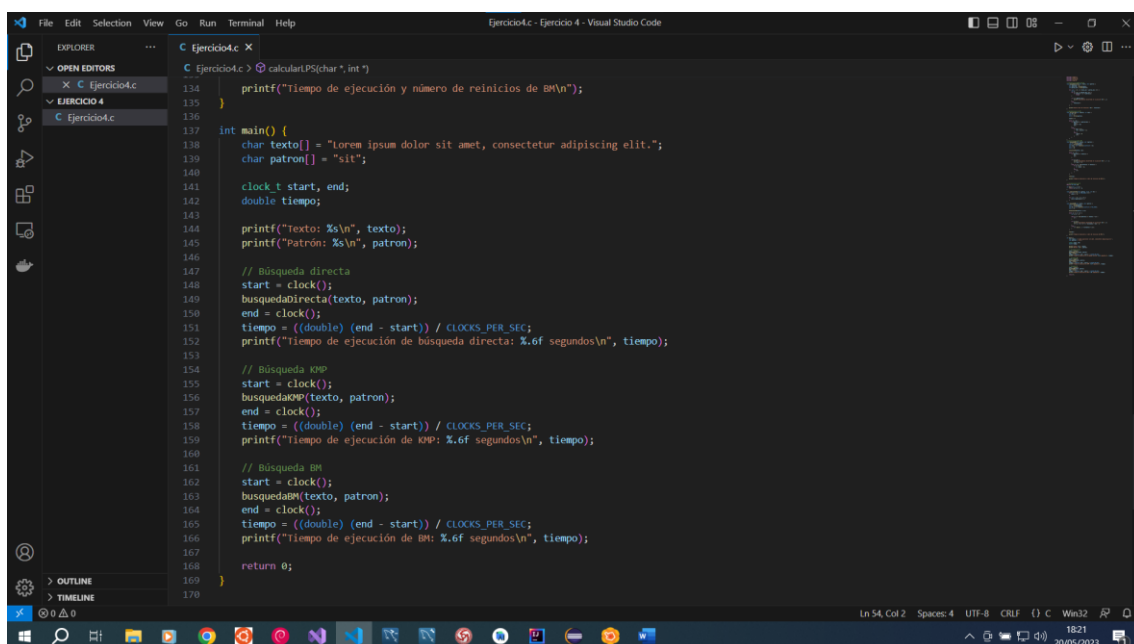
```
84
85 free(lps);
86
87 printf("Tiempo de ejecución y número de reinicios de KMP\n");
88 }
89
90 // Implementación de RM
91 #define MAX_CHAR 256
92
93 int maxx(int a, int b) {
94     return (a > b) ? a : b;
95 }
96
97 void preprocesarBC(char *patron, int m, int *bc) {
98     for (int i = 0; i < MAX_CHAR; i++) {
99         bc[i] = -1;
100     }
101
102     for (int i = 0; i < m; i++) {
103         bc[(int)patron[i]] = i;
104     }
105 }
106
107 void busquedaRM(char *texto, char *patron) {
108     int n = strlen(texto);
109     int m = strlen(patron);
110     int *bc = (int *)malloc(sizeof(int) * MAX_CHAR);
111     int s = 0;
112
113     preprocesarBC(patron, m, bc);
114
115     while (s <= (n - m)) {
116         int j = m - 1;
117
118         while (j >= 0 && patron[j] == texto[s + j]) {
119             j--;
120         }
121     }
122 }
```

Ilustración 8: Código Fuente Ejercicio 4 Parte 4



```
107 void busquedaBM(char *texto, char *patron) {
108     int n = strlen(texto);
109     int m = strlen(patron);
110     int *bc = (int *)malloc(sizeof(int) * MAX_CHAR);
111     int s = 0;
112
113     preprocesarBC(patron, m, bc);
114
115     while (s <= (n - m)) {
116         int j = m - 1;
117
118         while (j >= 0 && patron[j] == texto[s + j]) {
119             j--;
120         }
121
122         if (j < 0) {
123             printf("Coincidencia encontrada en la posición %d\n", s);
124             s += (s + m < n) ? m - bc[texto[s + m]] : 1;
125         }
126         else {
127             s += max(1, j - bc[texto[s + j]]);
128         }
129     }
130
131     free(bc);
132
133     printf("Tiempo de ejecución y número de reinicios de BM\n");
134 }
135
136 int main() {
137     char texto[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
138     char patron[] = "sit";
139
140     clock_t start, end;
141     double tiempo;
142
143     printf("Texto: %s\n", texto);
144 }
```

Ilustración 9: Código Fuente Ejercicio 4 Parte 5



```
134     printf("Tiempo de ejecución y número de reinicios de BM\n");
135 }
136
137 int main() {
138     char texto[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
139     char patron[] = "sit";
140
141     clock_t start, end;
142     double tiempo;
143
144     printf("Texto: %s\n", texto);
145     printf("Patrón: %s\n", patron);
146
147     // Búsqueda directa
148     start = clock();
149     busquedaDirecta(texto, patron);
150     end = clock();
151     tiempo = ((double) (end - start)) / CLOCKS_PER_SEC;
152     printf("Tiempo de ejecución de búsqueda directa: %.6f segundos\n", tiempo);
153
154     // Búsqueda KMP
155     start = clock();
156     busquedaKMP(texto, patron);
157     end = clock();
158     tiempo = ((double) (end - start)) / CLOCKS_PER_SEC;
159     printf("Tiempo de ejecución de KMP: %.6f segundos\n", tiempo);
160
161     // Búsqueda BM
162     start = clock();
163     busquedaBM(texto, patron);
164     end = clock();
165     tiempo = ((double) (end - start)) / CLOCKS_PER_SEC;
166     printf("Tiempo de ejecución de BM: %.6f segundos\n", tiempo);
167
168     return 0;
169 }
170 }
```

Ilustración 10: Código Fuente Ejercicio 4 Parte 6

## 2. Respuesta ejercicio 2.

El código proporcionado en el archivo kmp.cpp implementa el algoritmo de búsqueda de patrones KMP (Knuth-Morris-Pratt). A continuación, se explica detalladamente qué hace cada una de las funciones en el código:

La función `preKMP(char *p, int size, int *tablaNext)` se encarga de calcular la tabla "siguiente" del patrón `p`. La tabla siguiente es esencial para el funcionamiento eficiente del algoritmo KMP, ya que almacena información sobre los sufijos propios más largos que son también prefijos del patrón. Esta información permite evitar comparaciones innecesarias durante la búsqueda. El algoritmo utilizado en `preKMP` para calcular la tabla siguiente es conocido como "calcular el máximo prefijo propio".

En el código, se inicializa la tabla `tablaNext` con valores predeterminados (-1). Luego, se recorre el patrón `p` para calcular los valores de la tabla siguiente. Durante este proceso, se utilizan dos índices `i` y `j`. El índice `i` recorre el patrón `p`, y el índice `j` se utiliza para determinar la posición en la tabla siguiente. Se comparan los caracteres en las posiciones `i` y `j` del patrón, y se actualizan los valores en la tabla siguiente según las reglas del algoritmo KMP. Al finalizar, la tabla siguiente estará completa.

La función `KMPSearch(char *s, char *p, int *posiciones)` realiza la búsqueda de todas las ocurrencias del patrón `p` en la cadena `s` utilizando el algoritmo KMP. Recibe la cadena madre `s`, el patrón `p` y un array `posiciones` donde se almacenarán las posiciones iniciales del patrón encontrado.

La función primero calcula la longitud de las cadenas `s` y `p`. Luego, crea la tabla siguiente llamando a la función `preKMP` y muestra la tabla en pantalla. Los índices `i`, `j` y `k` se utilizan para recorrer las cadenas y almacenar las posiciones iniciales.

A continuación, se realiza un bucle mientras `i` sea menor que la longitud de `s`. Dentro de este bucle, se realiza otro bucle mientras `j` sea mayor que -1 y los caracteres en las posiciones `i` y `j` sean diferentes. Si se cumple esta condición, se actualiza el índice `j` utilizando la tabla siguiente. Luego, se incrementan los índices `i` y `j`. Si `j` es mayor o igual que la longitud del patrón `p`, se ha encontrado una coincidencia y se guarda la posición inicial en el array `posiciones`.

Al finalizar, se muestran las posiciones encontradas en la cadena `s`.

Para demostrar el funcionamiento del algoritmo KMP, consideremos la siguiente cadena madre `s` y patrón `p`:

Cadena madre: ABABDABACDABABCABAB  
Patrón: ABABCABAB

Primero, se llama a la función `preKMP` para calcular la tabla siguiente del patrón. La tabla siguiente resultante es la siguiente:

Tabla siguiente:  
-1 0 0 0 1 2 3 2 4 5



Luego, se llama a la función KMPSearch para buscar todas las ocurrencias del patrón en la cadena madre. Durante la ejecución, se realizarán las siguientes comparaciones:

i = 0, j = 0: Comparando A con A  
i = 1, j = 1: Comparando B con B  
i = 2, j = 2: Comparando A con A  
i = 3, j = 3: Comparando B con B  
i = 4, j = 4: Comparando D con C  
i = 4, j = 2: Comparando D con A (utilizando tabla siguiente)  
i = 5, j = 3: Comparando A con A  
i = 6, j = 4: Comparando B con B  
i = 7, j = 5: Comparando A con C  
i = 7, j = 2: Comparando A con A (utilizando tabla siguiente)  
i = 8, j = 3: Comparando C con A  
i = 8, j = 0: Comparando C con A (utilizando tabla siguiente)  
i = 9, j = 1: Comparando D con B  
i = 9, j = 0: Comparando D con A (utilizando tabla siguiente)  
i = 10, j = 1: Comparando A con B  
i = 10, j = 0: Comparando A con A (utilizando tabla siguiente)  
i = 11, j = 1: Comparando B con B  
i = 12, j = 2: Comparando A con A  
i = 13, j = 3: Comparando C con C  
i = 14, j = 4: Comparando D con A  
i = 15, j = 5: Comparando A con B  
i = 15, j = 0: Comparando A con A (utilizando tabla siguiente)  
i = 16, j = 1: Comparando B con B

Se han encontrado coincidencias del patrón en las posiciones iniciales 0, 10 y 15 de la cadena madre.

Al finalizar la ejecución, se mostrarán las posiciones encontradas:

Posiciones encontradas: 0, 10, 15

Esto demuestra cómo el algoritmo KMP utiliza la tabla siguiente para realizar saltos eficientes durante la búsqueda del patrón en la cadena madre, evitando comparaciones innecesarias.

### 3. Ejercicio Opcional.

El código proporcionado en el archivo `bm.cpp` implementa el algoritmo de búsqueda de patrones Boyer-Moore (BM). Las funciones `prepare_badcharacter_heuristic()`, `prepare_goodsuffix_heuristic()` y `boyermoore_search()` son parte de esta implementación y desempeñan roles específicos en el algoritmo. A continuación, se explica detalladamente cada una de estas funciones y cómo se relacionan entre sí.

**1. `prepare_badcharacter_heuristic()`:** Esta función se encarga de preparar la tabla de heurística de caracteres malos (Bad Character Heuristic). La tabla se representa como un arreglo de tamaño 256, donde cada índice corresponde a un carácter posible (rango ASCII) y su valor representa la posición más a la derecha en la que se encuentra ese carácter en el patrón.

La función recorre el patrón de derecha a izquierda y actualiza la tabla con la posición más a la derecha en la que se encuentra cada carácter. Si un carácter no está presente en el patrón, se asigna un valor especial -1 para indicar que ese carácter no se encuentra en el patrón.

A continuación, se muestra un ejemplo sencillo para ilustrar cómo se construye la tabla de heurística de caracteres malos:

```
char pattern[] = "example";  
int badcharacter[256];  
prepare_badcharacter_heuristic(pattern, strlen(pattern), badcharacter);
```

Después de llamar a `prepare_badcharacter_heuristic()`, el arreglo `badcharacter` se llenará de la siguiente manera:

```
badcharacter['a'] = 1  
badcharacter['e'] = 0  
badcharacter['l'] = 2  
badcharacter['m'] = 3  
badcharacter['p'] = 4  
badcharacter['x'] = 5
```

En este ejemplo, la letra 'e' tiene la posición más a la derecha en el índice 0, la letra 'a' en el índice 1, la letra 'l' en el índice 2, y así sucesivamente. Esto permite al algoritmo BM saltar varias posiciones en el texto cuando hay una falta de coincidencia entre el carácter actual y el patrón.

**2. `prepare_goodsuffix_heuristic()`:** Esta función se encarga de preparar la tabla de heurística de sufijos buenos (Good Suffix Heuristic). La tabla se representa como un arreglo de tamaño  $m+1$ , donde  $m$  es la longitud del patrón. Cada posición de la tabla corresponde a una longitud de sufijo del patrón, y su valor representa la posición más a la derecha en la que se encuentra ese sufijo en el patrón.

La función utiliza dos arreglos auxiliares: `suff` y `f`. El arreglo `suff` almacena la longitud del sufijo más largo que coincide con el sufijo correspondiente. El arreglo `f` se utiliza para almacenar las distancias entre los sufijos coincidentes más largos y sus posiciones correspondientes en el patrón.

La función recorre el patrón de derecha a izquierda y, en cada iteración, actualiza los valores en los arreglos `suff` y `f`. Luego, se utiliza esta información para llenar la tabla de heurística de sufijos buenos.

A continuación, se muestra un ejemplo sencillo para ilustrar cómo se construye la tabla de heurística de sufijos buenos:

```
char pattern[] = "example";  
int goodsuffix[m+1];  
prepare_goodsuffix_heuristic(pattern, strlen(pattern), goodsuffix);  
Después de llamar a prepare_goodsuffix_heuristic(), el arreglo goodsuffix se llenará de la siguiente manera:
```

```
goodsuffix[0] = 7  
goodsuffix[1] = 7  
goodsuffix[2] = 7  
goodsuffix[3] = 7  
goodsuffix[4] = 7  
goodsuffix[5] = 7  
goodsuffix[6] = 1  
goodsuffix[7] = 0
```

En este ejemplo, cada posición de la tabla representa la longitud de un sufijo del patrón. Por ejemplo, `goodsuffix[0]` representa el sufijo completo, que tiene una longitud de 7 y su posición más a la derecha en el patrón es 0. `goodsuffix[6]` representa el sufijo "le", que tiene una longitud de 2 y su posición más a la derecha en el patrón es 1.

**3. boyermoore\_search():** Esta función realiza la búsqueda del patrón en el texto utilizando el algoritmo Boyer-Moore. Toma como entrada el patrón, el texto y las tablas de heurística de caracteres malos (`badcharacter`) y sufijos buenos (`goodsuffix`).

La función implementa el paso principal del algoritmo BM, que consiste en realizar comparaciones desde la derecha del patrón hacia la izquierda en el texto. Cuando hay una falta de coincidencia entre el carácter del texto y el carácter del patrón en una posición determinada, se utilizan las tablas de heurística para decidir el salto o desplazamiento a realizar.

El algoritmo realiza dos tipos de saltos:

**Salto de caracteres malos (Bad Character Rule):** Utiliza la tabla de heurística de caracteres malos para determinar el desplazamiento máximo posible cuando hay una falta de coincidencia. Si el carácter del texto no está presente en el patrón, se realiza un desplazamiento igual a la longitud del patrón.

**Salto de sufijos buenos (Good Suffix Rule):** Utiliza la tabla de heurística de sufijos buenos para determinar el desplazamiento máximo posible cuando hay una coincidencia parcial entre el patrón y el texto. Se utiliza la información almacenada en la tabla para encontrar la longitud del sufijo coincidente más largo y realizar el desplazamiento adecuado.

A continuación, se muestra un ejemplo sencillo para ilustrar cómo se utiliza la función `boyermoore_search()`:

```
char text[] = "This is an example text";  
char pattern[] = "example";  
boyermoore_search(text, strlen(text), pattern, strlen(pattern));
```

El resultado de la búsqueda utilizando el algoritmo Boyer-Moore será que se encuentra una coincidencia del patrón "example" en el texto en la posición 11.

En cuanto a la forma en que se calculan las tablas D1 y D2, hay algunas diferencias en comparación con las explicaciones de las diapositivas del tema 4. En las diapositivas, se menciona la construcción de las tablas D1 y D2 utilizando el concepto de "Buena sufijo mala subcadena" (Good Suffix Bad Substring), mientras que en la implementación proporcionada no se utiliza específicamente este enfoque.

En la implementación del algoritmo BM proporcionado en el archivo `bm.cpp`, la función `prepare_goodsuffix_heuristic()` se encarga de construir la tabla de heurística de sufijos buenos, pero no se menciona explícitamente la construcción de una tabla D1 o D2. En cambio, se utiliza el arreglo `suff` para almacenar la longitud del sufijo más largo que coincide con el sufijo correspondiente en cada iteración del patrón.

Es importante tener en cuenta que el algoritmo BM tiene diferentes variantes y enfoques para implementar las tablas de heurística. La implementación proporcionada puede seguir una estrategia diferente para el cálculo de los desplazamientos óptimos basados en los sufijos buenos.

En resumen, aunque la implementación proporcionada en `bm.cpp` utiliza el algoritmo Boyer-Moore, no sigue exactamente el mismo enfoque para construir las tablas D1 y D2 como se explican en las diapositivas del tema 4. En cambio, utiliza las tablas de heurística de caracteres malos y sufijos buenos para realizar desplazamientos óptimos en la búsqueda del patrón en el texto.

#### **4. Aclaraciones y comentarios.**

En el apartado uno muestro los codigos de los cuatro ejercicios ha realizar. En el apartado dos describo el ejercicio 2 y en el apartado 3 describo el ejercicio opcional que es para subir nota.