**Realizado por:**

Wilder Valencia Ocampo 000375627

Emilio Martínez Rivera 000255600

Luis Esteban Santamaria 000291800

**1.** Consultar alternativas para implementar la función de comparación de similaridad (considerar al menos dos alternativas). Para cada alternativa indicar:

• Seudocódigo del algoritmo

• Tiempo necesario (usar como modelo de costo la comparación de caracteres)

Aclaración: la teoría y códigos enunciados a continuación no son de autoría propia, sino el resultados de investigación y extracción textual de fragmentos de las paginas mencionadas en la webgrafía.

**Algoritmos de búsqueda de subcadenas**

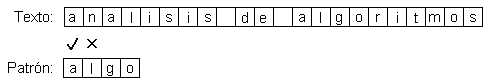
En general estos algoritmos consisten en buscar las apariciones de un cierto patrón P en un texto T.

**Algoritmo de Fuerza Bruta**

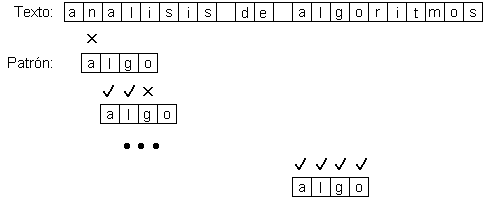
El algoritmo de Fuerza Bruta consiste en comparar carácter a carácter el patrón que se quiere buscar y la subcadena del texto que se considera en cada caso. Para ello chequea todas las subcadenas del texto de tamaño m. Si se encuentra una ocurrencia del patrón en el texto con menos discrepancias de las permitidas, se devuelve como resultado. Tras cada comparación se mueve el índice del texto una posición a la derecha.

Puede optimizarse esta versión de forma que si se supera el número de discrepancias permitidas también se mueve el índice del texto una posición a la derecha.

Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.



Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.



Este algoritmo no requiere preprocesado ni espacio extra en memoria.

En el peor caso este algoritmo realiza  comparaciones de caracteres.

Para el seudocódigo/código se ha tomado la versión que propone el libro Algorithms, 4th edition (Brute.java).

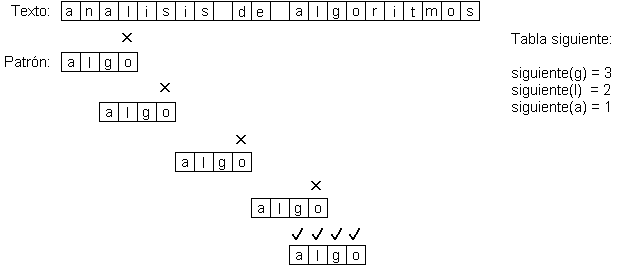
|  |
| --- |
| public static int **search1**(char[] pattern, char[] text) {  int m = pattern.length;  int n = text.length;  for (int i = 0; i <= n - m; i++) {  int j;  for (j = 0; j < m; j++) {  if (text[i+j] != pattern[j])  break;  }  if (j == m) return i; *// found at offset i*  }  return n; *// not found*  } |

**Algoritmo Boyer Moore (BM)**

El algoritmo de Boyer Moore es un algoritmo clásico de búsqueda de subcadenas. Fue desarrollado por Bob Boyer y J Strother Moore en 1977. Para conseguir una complejidad lineal respecto al tamaño del texto en la búsqueda del patrón desplaza el patrón varias posiciones después de cada comparación en vez de una sola posición como hace el algoritmo de fuerza bruta. El algoritmo produce buenos resultados cuando el tamaño del patrón a buscar es grande.

El algoritmo exacto intenta encajar el patrón de derecha a izquierda en el trozo del texto considerado y usa dos heurísticas, en el caso de que se produzca una discrepancia entre el texto y el patrón, para el desplazamiento del patrón:

* Regla de desplazamiento de carácter malo: se desplaza el patrón a la derecha para hacer coincidir el carácter no coincidente del texto con una posición más a la izquierda del patrón en donde ese carácter coincide. Supongamos que la discrepancia se produce al comparar el carácter de la posición k del texto con el carácter en la posición j del patrón. Si la posición más a la derecha del carácter T(k) se encuentra en el patrón a la izquierda de la posición j desplazamos el patrón hasta hacer coincidir la aparición más a la derecha del carácter T(k) en el patrón con la posición k del texto. Si el carácter T(k) se encuentra a la derecha de la posición j del patrón desplazamos el patrón en una posición. Por último, si el carácter T(k) no se encuentra en el patrón se desplaza el patrón hasta que la posición 0 del patrón esté alineada con la posición k + 1 del texto.
* Regla de sufijo bueno: se desplaza el patrón a la derecha para hacer coincidir el sufijo correcto del texto considerado con la siguiente ocurrencia de ese sufijo en el patrón. Además, el carácter incorrecto anterior al sufijo debe ser diferente al de la siguiente ocurrencia de ese sufijo en el patrón.



Este algoritmo requiere preprocesado.

Se puede demostrar que el tiempo promedio que toma el algoritmo BM es:



donde c es el tamaño del alfabeto (c<<n). Para un alfabeto razonablemente grande, el algoritmo es .

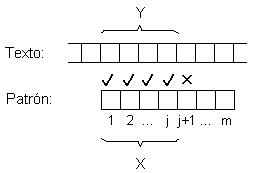
En el peor caso, BM tiene el mismo tiempo de ejecución que el algoritmo de fuerza bruta.

Para el seudocódigo/código se ha tomado la versión que propone el libro Algorithms, 4th edition (BoyerMoore.java).

|  |
| --- |
| public class BoyerMoore {  private final int R; *// the radix*  private int[] right; *// the bad-character skip array*  private char[] pattern; *// store the pattern as a character array*  private String pat; *// or as a string*  public **BoyerMoore**(String pat) {  this.R = 256;  this.pat = pat;  *// position of rightmost occurrence of c in the pattern*  right = new int[R];  for (int c = 0; c < R; c++)  right[c] = -1;  for (int j = 0; j < pat.**length**(); j++)  right[pat.**charAt**(j)] = j;  }  .  .  .  public int **search**(String txt) {  int m = pat.**length**();  int n = txt.**length**();  int skip;  for (int i = 0; i <= n - m; i += skip) {  skip = 0;  for (int j = m-1; j >= 0; j--) {  if (pat.**charAt**(j) != txt.**charAt**(i+j)) {  skip = Math.**max**(1, j - right[txt.**charAt**(i+j)]);  break;  }  }  if (skip == 0) return i; *// found*  }  return n; *// not found*  }  } |

**Algoritmo Knuth-Morris-Pratt (KMP)**

Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.

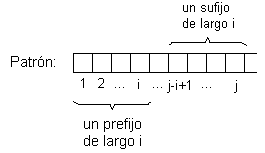


Sea X la parte del patrón que calza con el texto, e Y la correspondiente parte del texto, y suponga que el largo de X es j. El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros j-1 caracteres de X pueden o no pueden calzar los últimos j-1 caracteres de Y.

La observación clave que realiza el algoritmo Knuth-Morris-Pratt (en adelante KMP) es que X es igual a Y, por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite precalcular la respuesta y almacenarla en una tabla.

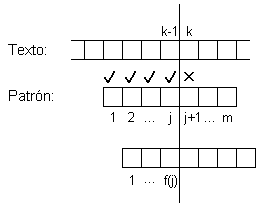
Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en 2, 3, ..., hasta j posiciones.

Se define la función de fracaso (failure function) del patrón como:



Intuitivamente, f(j) es el largo del mayor prefijo de X que además es sufijo de X. Note que j = 1 es un caso especial, puesto que si hay una discrepancia en b1 el patrón se desliza en una posición.

Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar bj+1, se desliza el patrón de manera que bf(j) se encuentre donde bj se encontraba, y se intenta calzar nuevamente.



Este algoritmo requiere preprocesado.

El tiempo de ejecución para calcular la función de fracaso puede ser acotado por los incrementos y decrementos de la variable i, que es .

Por lo tanto, el tiempo total de ejecución del algoritmo, incluyendo el preprocesamiento del patrón, es .

Para el seudocódigo/código se ha tomado la versión que propone el libro Algorithms, 4th edition (KMP.java).

|  |
| --- |
| public class KMP {  private final int R; *// the radix*  private int[][] dfa; *// the KMP automoton*  private char[] pattern; *// either the character array for the pattern*  private String pat; *// or the pattern string*  public **KMP**(String pat) {  this.R = 256;  this.pat = pat;  *// build DFA from pattern*  int m = pat.**length**();  dfa = new int[R][m];  dfa[pat.**charAt**(0)][0] = 1;  for (int x = 0, j = 1; j < m; j++) {  for (int c = 0; c < R; c++)  dfa[c][j] = dfa[c][x]; *// Copy mismatch cases.*  dfa[pat.**charAt**(j)][j] = j+1; *// Set match case.*  x = dfa[pat.**charAt**(j)][x]; *// Update restart state.*  }  }  .  .  .  public int **search**(String txt) {  *// simulate operation of DFA on text*  int m = pat.**length**();  int n = txt.**length**();  int i, j;  for (i = 0, j = 0; i < n && j < m; i++) {  j = dfa[txt.**charAt**(i)][j];  }  if (j == m) return i - m; *// found*  return n; *// not found*  }  } |

**Webgrafía**

Algoritmo Boyer-More. (2019). Retrieved 20 October 2019, from <https://prezi.com/2avslcnt1slx/algoritmo-boyer-more/>

(2019). Retrieved 20 October 2019, from <http://dspace.ucbscz.edu.bo/dspace/bitstream/123456789/975/1/1644.pdf>

(2019). Retrieved 20 October 2019, from <https://eprints.ucm.es/50249/1/1001321966-299767_BEATRIZ_CORONADO_SANZ_TFG_Beatriz_Coronado_Sanz_3506267_16039712.pdf>

BoyerMoore.java. (2019). Retrieved 20 October 2019, from <https://algs4.cs.princeton.edu/53substring/BoyerMoore.java.html>

Brute.java. (2019). Retrieved 20 October 2019, from <https://algs4.cs.princeton.edu/53substring/Brute.java.html>

CC30A Algoritmos y Estructuras de Datos: B�squeda en texto. (2019). Retrieved 20 October 2019, from <https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/>

Eliminar espacios de un String en Java - ChuWiki. (2019). Retrieved 20 October 2019, from <http://chuwiki.chuidiang.org/index.php?title=Eliminar_espacios_de_un_String_en_Java>

KMP.java. (2019). Retrieved 20 October 2019, from <https://algs4.cs.princeton.edu/53substring/KMP.java.html>

Substring Search. (2019). Retrieved 20 October 2019, from <https://algs4.cs.princeton.edu/53substring/>

GeeksforGeeks. (2019). *Boyer Moore Algorithm for Pattern Searching - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/> [Accessed 24 Oct. 2019].

Www-igm.univ-mlv.fr. (2019). *Boyer-Moore algorithm*. [online] Available at: <http://www-igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140> [Accessed 24 Oct. 2019].

Www-igm.univ-mlv.fr. (2019). *Brute force algorithm*. [online] Available at: <http://www-igm.univ-mlv.fr/~lecroq/string/node3.html#SECTION0030> [Accessed 24 Oct. 2019].

Www-igm.univ-mlv.fr. (2019). *ESMAJ*. [online] Available at: <http://www-igm.univ-mlv.fr/~lecroq/string/> [Accessed 24 Oct. 2019].

Www-igm.univ-mlv.fr. (2019). *KMP Skip Search algorithm*. [online] Available at: <http://www-igm.univ-mlv.fr/~lecroq/string/node32.html#SECTION00320> [Accessed 24 Oct. 2019].