Introducción de la problemática a resolver

El propósito de este proyecto fue desarrollar un programa que pudiera realizar búsqueda de patrones y diera sugerencias automáticas a partir de un texto base que se le alimente. Una aplicación que fuera sencilla de utilizar, intuitiva para el usuario, y eficiente al momento de manejar datos. Para esto utilizamos el algoritmo Z, con él encontraríamos las coincidencias entre el input del usuario y el texto, y la estructura Trie optimizado para mostrar las sugerencias con base a la frecuencia en la que se encuentran en el texto.

1. **Etapas del proyecto**

    a. **Etapa 1**

    Utilizamos textos extraídos del sitio web Project Gutenberg por lo cual son de dominio público, y por el bien del proyecto todos están en inglés para mantener constancia entre los análisis, y evitar los caracteres especiales. Las características de los textos son las siguientes:

    - Drácula de Bram Stocker
        - 839,104 caracteres aproximadamente de texto limpio
    - Frankenstein de Mary Shelley
        - 422,704 caracteres aproximadamente de texto limpio
    - Berenice de Edgar Allan Poe
        - 612,747 caracteres aproximadamente de texto limpio
    - The Castle of Otranto de Horace Walpole
        - 224,927 caracteres aproximadamente de texto limpio

    La elección de estos textos tienen varias justificaciones, empezando por sus longitudes, quisimos poder ser capaces de probar y comparar adecuadamente los algoritmos con textos muy grandes como Drácula, y textos más pequeños como

The Castle of Otranto, sumado a esto, se escogió “Frankenstein” y “Drácula” debido a que son libros clásicos, los cuales manejan un vocabulario diferente a libros más recientes, por lo cual resulta interesante comparar las sugerencias de autocompletado con la misma palabra en los diferentes libros, por último se tomaron en cuenta textos que tuvieran la misma categoría (terror), para hacer más precisa las comparaciones de sugerencias.

b. Etapa 2

Antes de empezar con los algoritmos nos enfocamos en empezar a tratar el texto, volviéndolo más sencillo de tratar al momento de con ellos.

Lo limpiamos eliminando todos los caracteres especiales, y convirtiéndolo todo a minúsculas. Utilizamos el siguiente código:

```
const textoLimpio = contenido
    .toLowerCase()
    .replace(/[^a-záéíóúüñ\s]/gi, "")
    .trim();
```

La variable *contenido* es el *string* de todo el texto, se guarda en esa variable todo el *string* usando useState de React, esto sucede cuando se abre el link que dirige a la vista del libro

c. Etapa 3

Decidimos implementar el algoritmo Z por su facilidad de uso y su eficiente y accesible costo computacional y en memoria. Esta técnica consiste en buscar un patrón dentro de un texto e ir guardando las coincidencias en un arreglo, realmente se utiliza para encontrar sufijos dentro de un texto, por lo que para hacer que funcione correctamente primero se concatena input$texto, si el total de coincidencias es igual a la

longitud del input, entonces sabemos que en esa posición del texto hay una coincidencia. A continuación un pseudocódigo que lo explica todo.

```
function ZAlgorithm(string):
    n = string.length;
    Z = [0] * n (array de 0s de longitud n)
    l, r = 0 (l y r se usa para reutilizar comparaciones anteriores y no
empezar desde el inicio de nuevo, hace el código más eficiente)
    por cada i desde 1 hasta n-1:
        si i  < r:
            Z[i] = min(r - i, Z[i - l])
        mientras i + Z[i] < n y string[Z[i]] = string[i + Z[i]]:
            Z[i] = Z[i] + 1
        si i + Z[i] > r:
            l = i
            r = i + Z[i]
    return Z
```

El código lo único que hace es evaluar los casos presentados, tal como vimos en clase y con las tablas de la función Z, donde uno de los casos era cuando i < r, entonces Z en la posición i se vuelve el mínimo entre r - i y Z en la posición i -1, después se hace una exploración a fuerza bruta, y si es necesario se reinicia el rectángulo (si i + Z[i] > r)

Para completar este algoritmo se hizo uso de una función llamada

```
buscar_patron_archivo(patron, texto):
        se concatena patrón y texto con "$" enmedio y se guarda en combinado
        calcula z = ZAlgorithm(combinado)
        posiciones = lista vacía
        m = longitud(patron)
        para i desde 0 hasta longitud(Z) -1:
                si Z[i] = m:
                        posReal = i - (m + 1)
                        si posReal >= 0:
                                agregar posReal a posiciones
        resultados = []
        para cada p en posiciones:
                fragmento = substring(texto, p, p + 50)
                agregar (posicion=p, fragmento=fragmento) a resultados


    devolver {
        encontrado: (longitud(resultados) > 0),
        totalCoincidencias: longitud(resultados),
        resultados: resultados,
        tiempoEjecucionSeg: (fin - inicio) / 1000
    }
```

Se regresa un breve contexto antes y después de la palabra encontrada (funcionalidad que aporta a la interfaz y ya implementada en actividad de búsqueda de patrones)

d. Etapa 4

En la siguiente etapa fue donde creamos la parte del autocompletado para el cual utilizamos la estructura Trie. Con esto pudimos mostrar la lista de palabras sugeridas, se podía seleccionar una sugerencia para autocompletarla y tiempo de búsqueda.

Clase NodoTrie

Atributos:

- children, un diccionario vacío
- _end_of_word = False

Esta es la función que es clave para el autocompletado debido a que con esta se exploran los nodos que siguen al prefijo ingresado por el usuario

Funcion DFS (nodo_inicial, prefijo):

palabras = una lista vacia

pila = una lista vacía

agregar (nodo_incial, prefijo) a pila

Mientras la pila no este vacia:

(nodo, palabra_actual), elimina último(pila)

Si nodo.fin_de_palabra es True:

se agrega (palabra_actual) a palabras

Para cada caracter en .hijos:

hijo <- nodo.hijos [caracter]

se agrega (hijo, palabra_actual + caracter) a la pila

return palabras

Clase Trie:

Atributos:

- raiz, nuevo nodo

Constructor()

raiz es nuevo nodoTrie

En la siguiente función es donde irán insertando las palabras, junto con la función insertar_texto fue con la que procesamos el texto ya limpio para poder manejarlo.

Funcion insert(word):

nodo_actual es la raíz

Para cada caracter en palabra:

Si el caracter no esta en los hijos del nodo actual:

nodo_actual.hijos[c] se vuelve un nuevo nodo

nodo_actual <- nodo_actual.hijos[c]

nodo_actual._end_of_word = True

Esta función lo que hace es que va mandando a llamar la funcion insertar en cada palabra para agregarla a la estructura, de manera que se pueda manejar

Funcion insert_text(text):

se divide el texto por espacios, esto queda en una variable words

Para cada palabra en palabras:

insert(word)

Busca una palabra en especifico

Funcion search(word)

nodo_actual <- raíz

Para cada caracter en palabra

Si el carácter no está en children de ese nodo:

return False

nodo_actual <- nodo_actual.hijos[caracter]

return nodo_actual._end_of_word

Verifica si existe un prefijo

Funcion starts_width(prefijo)

nodo_actual <- raíz

Para cada caracter en el prefijo:

Si el caracter no esta en Children del nodo actual:

return false

nodo_actual <- nodo_actual.hijos[caracter]

Aqui esta localizada la lógica necesaria para el autocompletar

Funcion autocomplete(prefijo):

iniciar_reloj() para medir el tiempo que tarda en dar las sugerencias

nodo_actual <- raíz

Para cada caracter en el prefijo:

Si el caracter no esta en children del nodo actual:

return {resultados: [], tiempo: 0}

nodo_actual <- nodo_actual._children[caracter]

results = DFS(nodo_actual, prefijo)

detener_reloj()

tiempo = tiempo transcurrido

return {resultados, tiempo }

Con esta implementación del Trie, con la función del autocompletado logramos optimizar la búsqueda de palabras dentro del texto. Recorre de forma eficiente los prefijos que podrían ser ingresados por el usuario, permitiendo tener una experiencia más agradable. Al medir el tiempo de ejecución, fue posible observar el comportamiento del código, incluso antes de agregarlo por completo a la interfaz.

e. Etapa 5

Esta etapa en la que se decidió agregar una mejora al código. La que nosotros elegimos fue poner ordenamiento frecuencia, lo cual significa que las sugerencias que el usuario vería primero son las más frecuentes dentro del texto. A continuación se explicarán los cambios que se le hicieron al código, con la explicación del pseudocódigo y las mejoras marcadas en rojo :

Clase NodoTrie

Atributos:

- children, un diccionario vacío
- _end_of_word = False

- _frequency = 0

Esta es la función que es clave para el autocompletado debido a que con esta se exploran los nodos que siguen al prefijo ingresado por el usuario

```
Funcion DFS (nodo_inicial, prefijo):
    palabras = una lista vacia
    pila = una lista vacía
    agregar (nodo_incial, prefijo) a pila
        Mientras la pila no este vacia:
            (nodo, palabra_actual), elimina último(pila)
            Si nodo.fin_de_palabra es True:
                se agrega (palabra_actual, nodo._frequency) a
                palabras
            Para cada caracter en children:
                hijo <- nodo._children [caracter]
                se agrega (hijo, palabra_actual + caracter) a la
               pila
            return palabras
```

Clase Trie:

Atributos:

- raiz, nuevo nodo

Constructor()

raiz es nuevo nodoTrie

En la siguiente función es donde irán insertando las palabras, junto con la función insertar_texto fue con la que procesamos el texto ya limpio para poder manejarlo.

```
Funcion insert(word):
        nodo_actual es la raíz
        Para cada caracter en palabra:
                Si el caracter no esta en los hijos del nodo actual:
                        nodo_actual.hijos[c] se vuelve un nuevo nodo
                nodo_actual <- nodo_actual.hijos[c]
        nodo_actual._end_of_word = True
        nodo_actual._frequency = nodo_actual._frequency + 1
```

Esta función lo que hace es que va mandando a llamar la funcion insertar en cada palabra para agregarla a la estructura, de manera que se pueda manejar

```
Funcion insert_text(text):
        se divide el texto por espacios, esto queda en una variable words
        Para cada palabra en palabras:
                insert(word)
```

Busca una palabra en especifico

```
Funcion search(word)
nodo_actual <- raíz
Para cada caracter en palabra
        Si el carácter no está en children de ese nodo:
                return False
        nodo_actual  <- nodo_actual.hijos[caracter]
return nodo_actual._end_of_word
```

Verifica si existe un prefijo

Funcion starts_width(prefijo)

nodo_actual <- raíz

Para cada caracter en el prefijo:

Si el caracter no esta en Children del nodo actual:

return false

nodo_actual <- nodo_actual.hijos[caracter]

Aqui esta localizada la lógica necesaria para el autocompletar

Funcion autocomplete(prefijo):

iniciar_reloj() para medir el tiempo que tarda en dar las sugerencias

nodo_actual <- raíz

Para cada caracter en el prefijo:

Si el caracter no esta en children del nodo actual:

return {resultados: [], tiempo: 0}

nodo_actual <- nodo_actual._children[caracter]

results = DFS(nodo_actual, prefijo)

se ordenan los resultados de mayor a menos según la frecuencia

top5 = los primeros 5 resultados

detener_reloj()

tiempo = tiempo transcurrido

return {resultados: top 5, tiempo: tiempo }

Con esto fuimos contando la frecuencia de las palabras en el texto, y pensando en la interfaz y por extensión en el usuario, redujimos los resultados mostrados a los primeros 5 para volverlo más fácil al momento de buscar. El usuario al momento de seleccionar una de estas primeras 5 sugerencias será llevado a la búsqueda dentro del texto que está manejada con el algoritmo z explicado anteriormente.

f. Etapa 6

En esta última etapa deberás evaluar y comparar el desempeño de tu herramienta. Aplica diferentes búsquedas y mide:

palabras a probar: “titulo (frankesntein, dracula, etc..)”, “sadness”, “blood”, “monster”, “dark”

Drácula, tiempo promedio

Trie: 0.0002 segundos

ZFunction: 0.016500 segundos

Frankenstein, tiempo promedio:

Trie: 0.0001 segundos

ZFunction: 0.007200 segundos

Berenice, tiempo promedio:

Trie: 0.0002 segundos

ZFunction: 0.008300 segundos

The castle of Otranto, tiempo promedio;

Trie: 0.0001 segundos

ZFunction: 0.003700 segundos

Precisión de palabras: se agregó un contador de palabras como manera de eficientar el proceso y mostrar los resultados más probables, en promedio se necesitan de 2-3 letras para que el texto pueda predecir con eficiencia lo que el usuario quiere buscar, en este caso, hablando específicamente de las palabras puestas aprueba

Cuáles fueron sus ventajas y limitaciones.

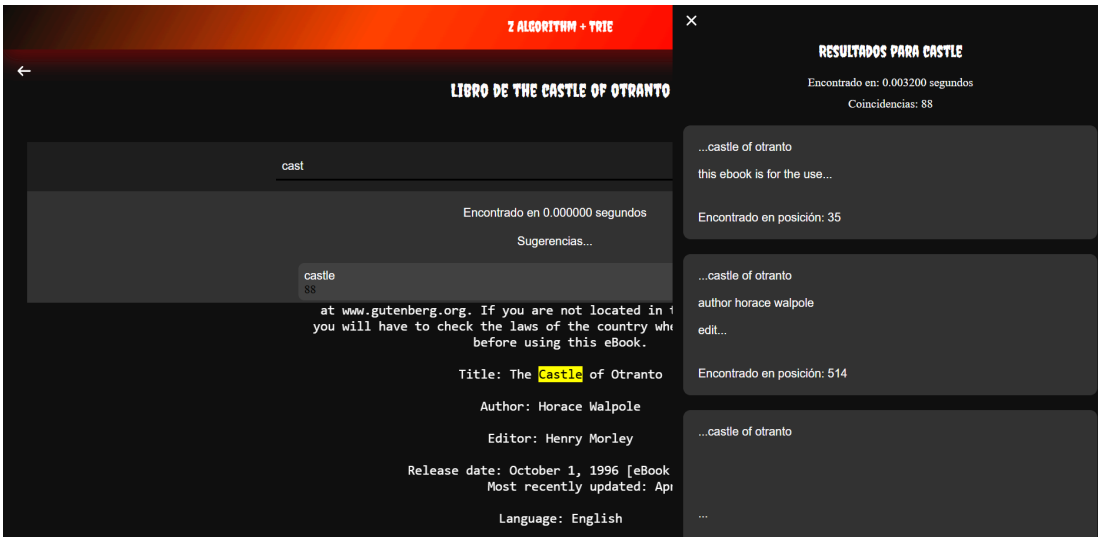
Las ventajas del Trie fue que teníamos control total sobre el vocabulario, ya que se separaba el texto por espacios y se llevaba una cuenta exacta de todas las palabras dentro del texto, la única desventaja al implementarlo fue que aún consideraba algunas palabras conectoras como “the”, “a”,”and”, etc.. Aun así, funcionaba con eficiencia y siempre mostraba correctamente los resultados más probables

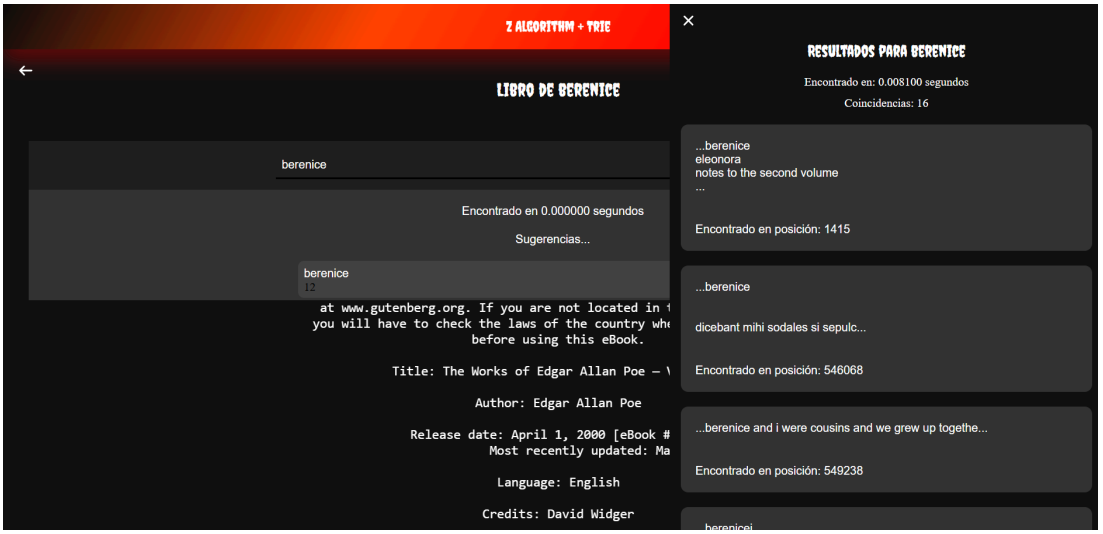
Las ventajas de aplicar el algoritmo Z fueron que las coincidencias son exactas y gracias a eso pudo saber en qué posición se encuentran las coincidencias con exactitud y poder resaltar las mismas, así como mostrar la posición y poder dar un breve contexto donde se encuentra, alumnas limitaciones fueron que mientras más crecía el texto y mientras más fueran las coincidencias era más tardado y evidentemente más pesado para el navegador renderizar todo el contenido.

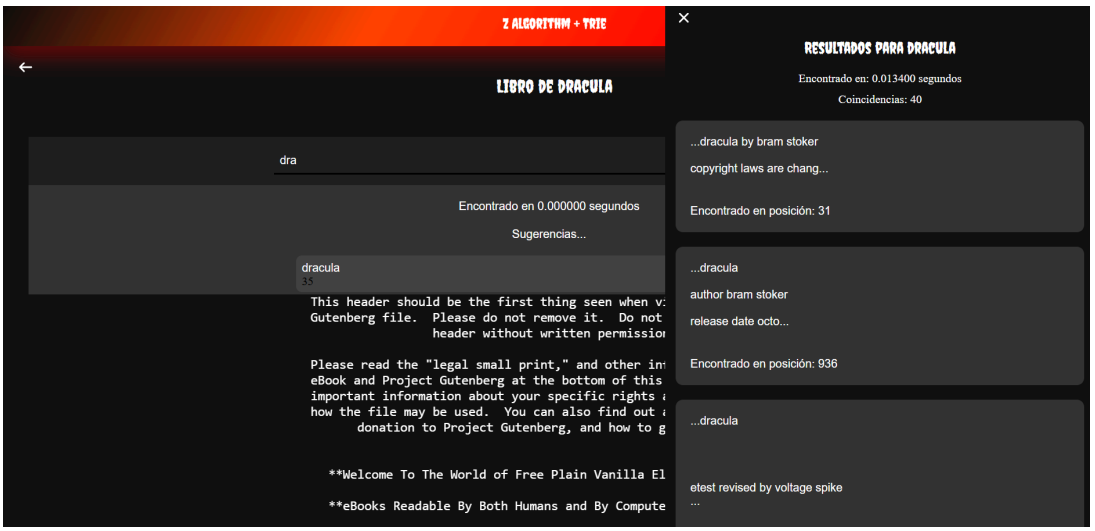
Cómo afecta la longitud del texto o el tamaño del vocabulario al rendimiento.

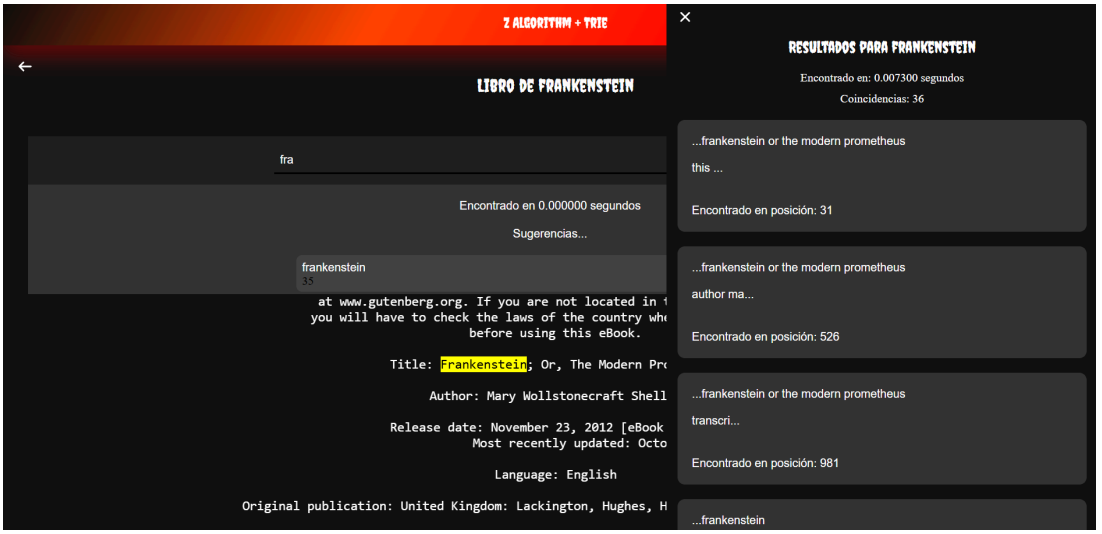
Afecta negativamente al rendimiento, y debido a sus complejidades algorítmicas lo dicen por sí mismas, aunque la diferencia sea casi imperceptible, como se mencionó anteriormente, era mucho más pesado mostrar todo ese contenido mientras fuera creciendo el texto (tanto sugerencias como las coincidencias dentro del texto)

Evidencias:









2. Explicación de cada algoritmo implementado, indicando las técnicas de diseño y un análisis de su complejidad algorítmica

Nuestro proyecto usó dos algoritmos principales:

1. Algoritmo z, para la búsqueda exacta de patrones dentro del texto
2. Trie, para el autocompletado con mejora de frecuencia

Algoritmo Z

Es un método eficaz para realizar búsquedas exactas de patrones dentro del texto. Su concepto principal viene hacer un arreglo Z, donde cada posición del arreglo

indica la longitud del prefijo más largo el texto que coincide con un *substring* en esa posición.

Para buscar un patrón dentro del texto, se le concatena un patrón con el símbolo $ para ser usado como separador al texto original. Después de esto, el algoritmo calculará el arreglo Z para el *string* que resulta, indicando las posiciones donde coincide el valor Z con la longitud del patrón, lo cual indica concurrencia exacta

Tiene una complejidad algorítmica de O(n + m), y utiliza la técnica de ventana deslizante para comparar sin reiniciar las comparaciones.

Estructura Trie con ordenamiento por frecuencia

El Trie, es un árbol de prefijos, una estructura de datos diseñada para almacenar *strings* de una manera jerarquizada. Cada nodo del Trie representa un carácter, y sus respectivas rutas representan palabra completas. En el caso de nuestro proyecto, el Trie se utilizó para implementar la función de autocompletado, porque se pueden encontrar posibles terminaciones, según el prefijo escrito por el usuario al recorrer cada rama que nace de ese nodo.

El *insert* al árbol tiene una complejidad algorítmica de O(L) donde L depende de la palabra que se va a insertar, y el autocompletado es de O(P+K) donde se suma el número de resultados con la longitud del prefijo. Su técnica de diseño es división y conquista

3. Conclusiones personales

a. Emiliano Durán Fuentes

Este proyecto me ayudó a implementar correctamente lo aprendido en clase y darle un uso a estos algoritmos, aprendí además que también importa mucho

considerar otros factores como toda la carga al navegador en cuanto al contenido y aprendí a adaptarlo a estas mismas necesidades

b. Ivana Banderas Elliot

Con este proyecto tuve una poco más comprensión sobre lo que sucede cada que busco en cualquier página. Tuve la oportunidad de implementar todo lo que hemos ido aprendiendo durante la clase, además de tomar en cuenta cuanto es que una mala implementación puede alentar drásticamente los resultados que se desean obtener. Subrayando la importancia de un buen diseño de algoritmos, de la planeación antes de la implementación, para volver más eficiente el código, tomando en cuenta no solo los algoritmos como tal, sino también el hecho que se tienen que cargar para poder ser visualizados en pantalla.

4. **Video explicación**

https://drive.google.com/file/d/19kMxK_F5iSLRlyD9CtSsxGBw4JhnFJUT/view?usp=sharing