El objetivo de este proyecto es crear una especie de ctr + f con autocompletado. La manera en la que lo abordaré es una aplicación en la que cargues un archivo txt en utf-8 y se visualice, tendrá una entrada en la que se podrá ingresar el patrón o palabra que se quiere buscar y abajo se mostrarán las sugerencias de autocompletado. Los matches se resaltarán en amarillo.

Notas Importantes:
La aplicación está hecha con C++ y python, la manera para comunicar los dos lenguajes es con pybind11 (se debe de incluir la librería de pybind11 con github y compilar las funciones para ver todo completo aqui hay un repositorio https://github.com/goliat2020/Autocompletado-y-busqueda-de-patrones), los algoritmos fueron hechos en C++ y exportados a la gui de python

Etapa 1 - Selección y lectura de textos

- Texto elegido para las pruebas: Frankenstein; Or, The Modern Prometheus by Mary Wollstonecraft Shelley
- Fuente de donde se obtuvo: Project Gutenberg
- Cantidad de palabras: 78 529
- Cantidad de caracteres: 446 542
- Justificación de la elección: Es un libro largo pero no tanto, se puede descargar gratis en project gutenberg y ya lo había usado así que ya tengo una benchmark de tiempo y podré ver que tanta diferencia hay usando la gui

Etapa 2 - Preprocesamiento del texto

Primero se lee todo el texto completo sin limpiar para la función KMP y se almacena como un string, después se recorre todo ese string para almacenar en un set las palabras únicas ya limpias en minúsculas solo letras.

Para la lectura usé la librería fstream de C++, para el guardado de las palabras primero usé la función isalpha de la librería std para solo guardar letras y para hacerlas minúsculas la función tolower igual de la librería std, usando static_cast para estas funciones

n = número de caracteres en el texto
Tiene una complejidad computacional de O(n)

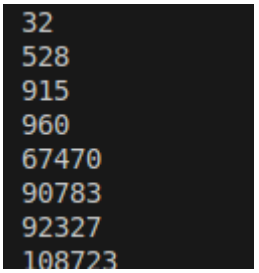
Etapa 3 - Implementación de algoritmo de búsqueda de patrones

Para este proyecto elegí el algoritmo KMP (Knuth-Morris-Pratt).
Este tiene una complejidad algorítmica de O(n + m)
n = número de caracteres en el texto
m = tamaño del patrón a buscar

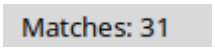
Estos son los resultados con la palabra Frankenstein:

- Tiempo de búsqueda: 

- Posición de resultados:



- Número de ocurrencias:



Etapa 4 - Implementación del autocompletado

En este caso elegí usar como estructura un Trie

La complejidad de la construcción es de O(n)
n = Caracteres totales de todas las palabras únicas

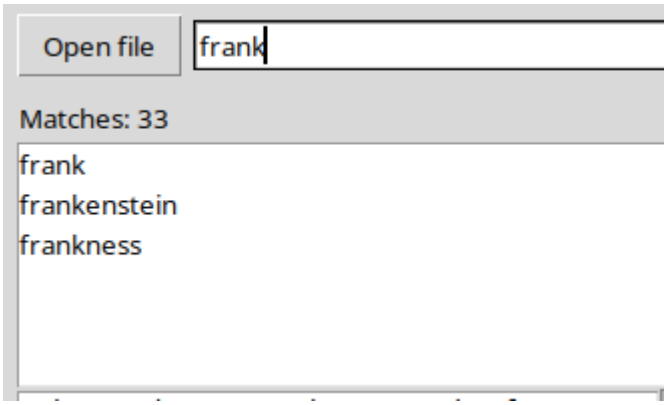
La complejidad del autocompletado es de O(N * L)
N = número de palabras en el trie
L = El tamaño de la palabra más larga en el trie

Cada nodo tiene de contiene un unordered_map que tiene como llave una letra y como info un apuntador a los nodos hijos.

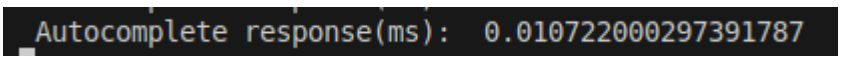
Se construye de una manera normal recorriendo todo el set de las palabras únicas

Para el autocompletado se usan dos funciones la primera recorre el el camino del prefijo y después llama a la segunda que explora todos los hijos recursivamente y regresa esa lista a la primera función que ahora sortea esa lista de palabras por frecuencia.

Muestra de la lista del autocompletado y se le puede hacer click a las sugerencias



Tiempo de respuesta del auto completado con la palabra **frank**:



Etapa 5 - Mejoras y extensiones del autocompletado

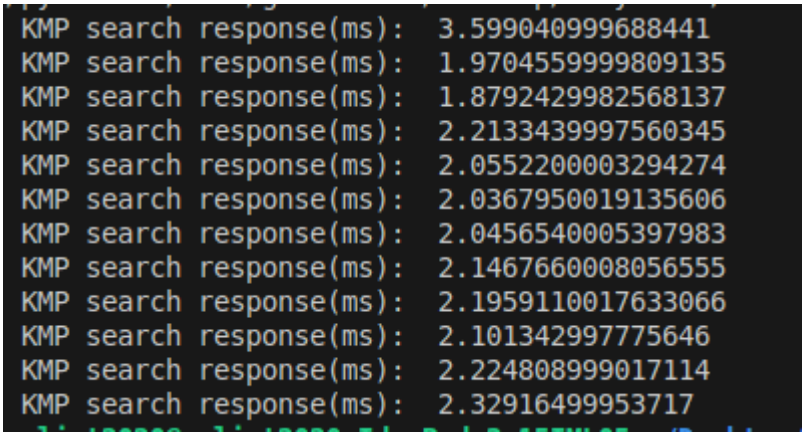
Para esta etapa elegí la la mejora que las palabras mostradas por el autocompletado sean mostradas por frecuencia. La principal razón de la elección de esta mejora es porque personalmente me parece la más importante y la que más cambiaría la experiencia del usuario. Y la que menos cambiaría la complejidad computacional.

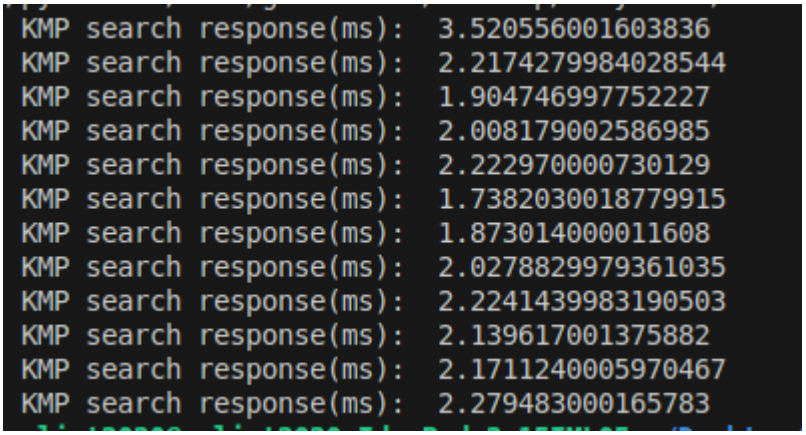
Solo tuve que agregar un unordered_map mas que guarde la frecuencia de las palabras y a antes de regresar el vector que contiene las posibles palabras autocompletadas con el prefijo, se sortea de mayor a menor en frecuencia y si tienen la misma frecuencia se ordena lexicográficamente.

frankenstein se repite 31 veces, france 5 y las demás 1 sola vez.
A pesar de que frank tiene 33 matches, 31 de esos matches son de frankenstein, 1 de frankness y 1 de frank. Es decir se ordena por matches de la palabra exacta

Etapa 6 - Comparación y análisis de resultados

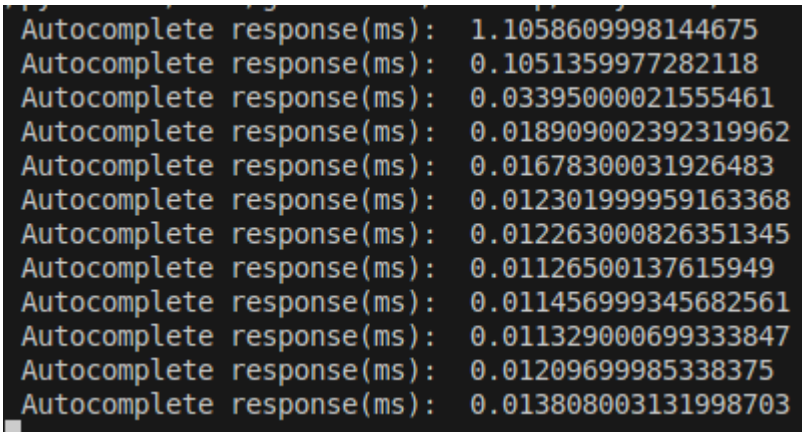
Tiempos de búsqueda de patrones:
Las búsquedas de este ejemplo fueron todos los prefijos de la palabra frankenstein, es decir escribí la palabra frankenstein y cada letra escrita hace la llamada al algoritmo KMP para búsqueda de patrones.



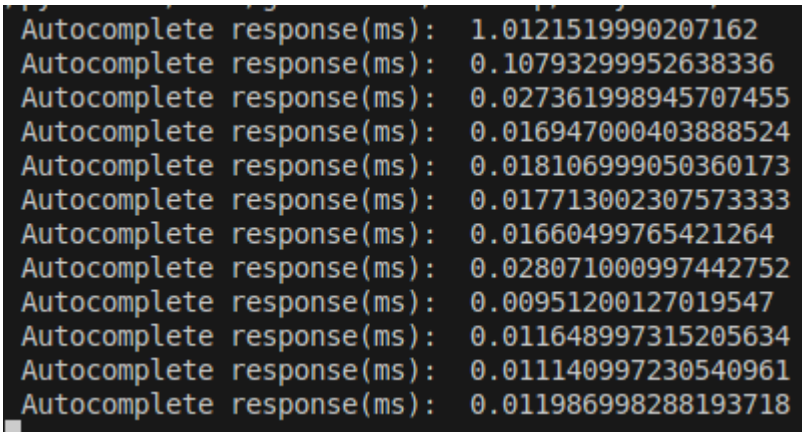


Vemos que se queda masomenos igual en unos 2ms en las dos pruebas pero constantemente la primera es la que más tarda, por todos los matches que hay ya que tiene que encontrar todas las “f” en el texto.

Tiempos de sugerencias (Autocompletado):
Las búsquedas de este ejemplo fueron todos los prefijos de la palabra frankenstein, es decir escribí la palabra frankenstein y cada letra escrita hace la llamada de las sugerencias.

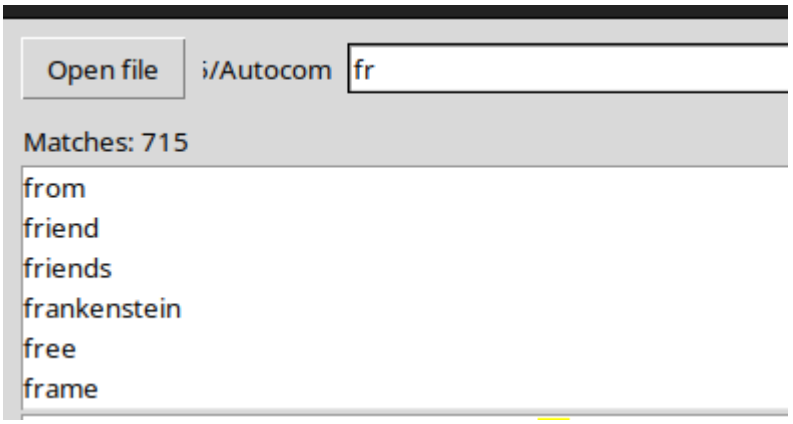


Vemos que la tendencia es que el tiempo vaya bajando entre más largo sea el prefijo que introducimos en el buscador



Aqui otra vez la misma prueba y nos damos cuenta que se mantiene muy parecido

Precisión o utilidad de las sugerencias:



Creo que este punto seria algo mas personal pero considero que las sugerencias son bastante buenas, tal vez lo único es que no es muy común que alguien en un texto vaya a buscar una palabra como “from” o “the” ya que normalmente buscamos algo mas especifico como frankenstein, pero para que primero recomiendo este tipo de palabras lo que se me ocurre seria darle algún peso a las palabras que se consideren importantes o las que más búsquedas tengan, pero haría que el desarrollo del programa sea complejo. La razón por la que decidí no eliminar estas stopwords fue porque prefiero que se muestren en el autocompletado a que no.

Análisis

Para la búsqueda lo que mejor me funcionó para esta aplicación fue el KMP, con una ventaja de tiempo considerable debido a la longitud del texto.

Si el texto fuera aún más largo me pensaría implementar algún otro algoritmo de búsqueda de patrones que sea mejor para textos largos

Conclusiones personales

Este proyecto me lo tomé más como un reto o una oportunidad de aprendizaje, esa es la razón por la que elegí el segundo proyecto. Al mismo tiempo también es la razón de mi elección de usar dos lenguajes diferentes y comunicarnos, también aprendí a hacer la gui con python y el autocompletado con un trie.