

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

Etapas 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

Etapas 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 limpiadas 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 $O(n)$

Etapas 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: **KMP TIME (ms): 218.31954600202153**

```
32
528
915
960
67470
90783
92327
108723
```

- Posición de resultados:

- Número de ocurrencias: Matches: 31

Etapla 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

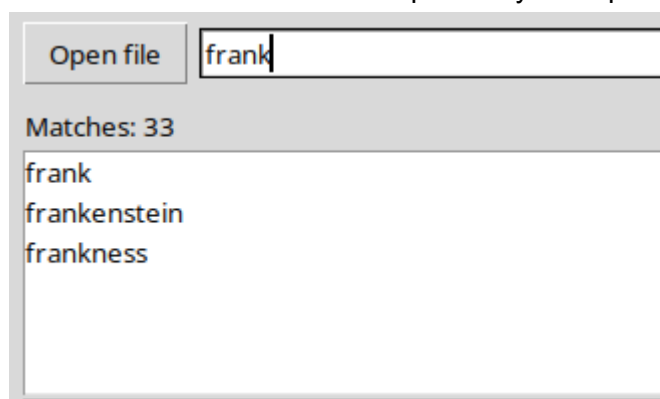
La complejidad del autocompletado es de $O(N * L)$
 N = número de palabras en el trie
 L = El tamaño de la palabra más larga en el trie

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



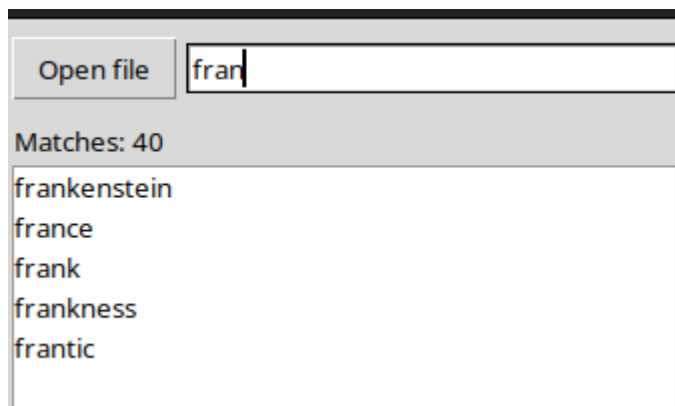
Tiempo de respuesta del auto completado con la palabra **frank**:

```
Autocomplete response(ms): 0.010722000297391787
```

Etapas 5 - Mejoras y extensiones del autocompletado

Para esta etapa elegí 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

Etapas 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.

```
KMP search response(ms): 3.599040999688441
KMP search response(ms): 1.9704559999809135
KMP search response(ms): 1.8792429982568137
KMP search response(ms): 2.2133439997560345
KMP search response(ms): 2.0552200003294274
KMP search response(ms): 2.0367950019135606
KMP search response(ms): 2.0456540005397983
KMP search response(ms): 2.1467660008056555
KMP search response(ms): 2.1959110017633066
KMP search response(ms): 2.101342997775646
KMP search response(ms): 2.224808999017114
KMP search response(ms): 2.32916499953717
```

```
KMP search response(ms): 3.520556001603836
KMP search response(ms): 2.2174279984028544
KMP search response(ms): 1.904746997752227
KMP search response(ms): 2.008179002586985
KMP search response(ms): 2.222970000730129
KMP search response(ms): 1.7382030018779915
KMP search response(ms): 1.873014000011608
KMP search response(ms): 2.0278829979361035
KMP search response(ms): 2.2241439983190503
KMP search response(ms): 2.139617001375882
KMP search response(ms): 2.1711240005970467
KMP search response(ms): 2.279483000165783
```

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.

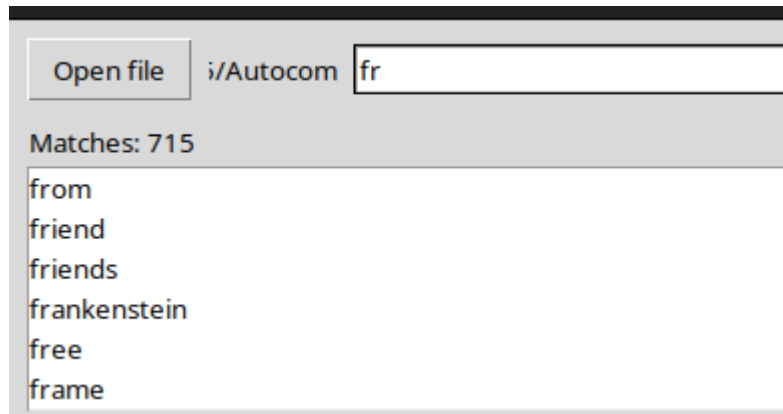
```
Autocomplete response(ms): 1.1058609998144675
Autocomplete response(ms): 0.1051359977282118
Autocomplete response(ms): 0.03395000021555461
Autocomplete response(ms): 0.018909002392319962
Autocomplete response(ms): 0.01678300031926483
Autocomplete response(ms): 0.012301999959163368
Autocomplete response(ms): 0.012263000826351345
Autocomplete response(ms): 0.01126500137615949
Autocomplete response(ms): 0.011456999345682561
Autocomplete response(ms): 0.011329000699333847
Autocomplete response(ms): 0.01209699985338375
Autocomplete response(ms): 0.013808003131998703
```

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

```
Autocomplete response(ms): 1.0121519990207162
Autocomplete response(ms): 0.10793299952638336
Autocomplete response(ms): 0.027361998945707455
Autocomplete response(ms): 0.016947000403888524
Autocomplete response(ms): 0.018106999050360173
Autocomplete response(ms): 0.017713002307573333
Autocomplete response(ms): 0.01660499765421264
Autocomplete response(ms): 0.028071000997442752
Autocomplete response(ms): 0.00951200127019547
Autocomplete response(ms): 0.011648997315205634
Autocomplete response(ms): 0.011140997230540961
Autocomplete response(ms): 0.011986998288193718
```

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 sería algo más 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 más específico como frankenstein, pero para que primero recomiende este tipo de palabras lo que se me ocurre sería 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.