BUSCA E ORDENAÇÃO
PROJETO 2

- **Gabriel Davi**
- **Giordani Andre**

DESAFIOS ESCOLHIDOS

- KMP
- Uso de wildscore
- Buscas de várias palavras
- Marcação de erros ortográficos

KMP

🧠 Objetivo

Encontrar todas as ocorrências de uma palavra no texto, usando o algoritmo Knuth-Morris-Pratt (KMP), que é eficiente ao evitar repetições desnecessárias de comparação.

⚙️ Implementação

- **Arquivo**: kmp.cpp

- **Modificação principal**: Adição do wildcard * que casa com qualquer caractere.

- **Lógica do LPS (Longest Prefix Suffix)**:
  - Adapta a construção da tabela LPS para aceitar * como caractere curinga.
- **Na busca**:
  - Quando o caractere do padrão for *, ele casa com qualquer letra do texto.

BUSCA DE MÚLTIPLAS PALAVRAS COM AHO-CORASICK

🧠 Objetivo

Permitir a busca de várias palavras ao mesmo tempo, de forma eficiente, mesmo em textos longos.

⚙️ Implementação

- Arquivo: aho_corasick.{h,cpp}
- Estrutura: Usa uma trie (árvore de prefixos) para armazenar as palavras e cria ponteiros de falha que agilizam a busca.
- Paralelismo: O texto é dividido em pedaços e cada pedaço é processado por uma thread separada.

🧩 Detalhes técnicos:

- Ponteiros de falha evitam reiniciar a busca do zero após falhas parciais.
- Usa std::mutex para proteger o acesso simultâneo ao mapa de resultados.

🎯 Decisão:

Aho-Corasick é ideal para busca de vários padrões, com complexidade linear em relação ao tamanho do texto + total dos padrões. Usamos paralelismo com std::thread para melhorar desempenho em textos grandes.

COMPARAÇÃO DE DESEMPENHO

🧠 Objetivo

Avaliar o tempo de execução dos algoritmos.

🛠️ Técnica:

- Usa chrono::high_resolution_clock para medir tempo.
- Tempo impresso após cada busca, com precisão em milissegundos.

🎯 Decisão:

A medição direta no terminal permite comparar efetivamente o custo dos algoritmos em diferentes condições e conjuntos de palavras.

📊 Tabela

Critério	Ingênuo	KMP	Aho-Corasick
Complexidade (1 palavra)	O(n * m)	O(n + m)	O(n + m)
Complexidade (várias palavras)	O(k * n * m)	O(k * (n + m))	O(n + z)
Pré-processamento	Nenhum	Criação de tabela LPS (O(m))	Construção da Trie + fail links (O(total caracteres das palavras))
Busca em texto	Comparações repetidas	Evita comparações desnecessárias	Uma única passagem com múltiplos matches
Melhor para	Testes simples	Busca de uma única palavra	Múltiplas palavras
Uso de memória	Baixo	Baixo	Médio (Trie ocupa mais memória)

🔷5. VERIFICAÇÃO ORTOGRÁFICA COM TRIE

🧠 **Objetivo**: Identificar palavras do texto que não estão presentes no dicionário fornecido (pt_BR.dic).

⚙️ **Implementação**

- Arquivo: trie.{h,cpp}
- Trie: Estrutura de árvore que permite busca rápida de palavras.
- Leitura do texto: Utiliza biblioteca utf8.h para correta leitura de palavras acentuadas e caracteres especiais.

🔠 **“Limitação”**

- Cada palavra é convertida para minúsculas “**tolower()**” antes de ser inserida e buscada na trie.

🎯 **Decisão**:

Trie permite busca rápida (tempo proporcional ao tamanho da palavra) e ocupa menos memória que hash tables grandes para muitas palavras. A verificação é feita em todo o texto não possuindo limitações próprias

CÓDIGO

OBRIGADO !!