Trabalho Prático 2

DCC214 - Estrutura de Dados

Igor Joaquim da Silva Costa

1. Introdução

O problema proposto foi realizar uma análise do número de ocorrências das palavras usadas em um texto baseada em uma nova ordem lexicográfica.

Para resolver o problema citado, foram usados dois algoritmos de ordenação, o primeiro sendo o Quicksort com mediana de M elementos e o segundo sendo a ordenação InsertionSort para textos pequenos, sempre considerando a nova ordem lexicográfica requerida. Além disso, o programa é capaz de corrigir eventuais problemas no arquivo de entrada, realizando a conversão de caracteres acentuados para seu respectivo par, sem acentos, a união de palavras com hífen separadas por espaços em branco e a remoção de pontuação.

Diante do exposto, a documentação presente possui como objetivo detalhar como o sistema foi implementado (Seção 2), o quão eficiente ele pode ser em termos teóricos(Seção 3) e aplicados (Seção 5), além de explicar como o programa lida com possíveis inconsistências durante sua execução (Seção 4). Por fim, o projeto é sumarizado junto com os aprendizados gerados durante a produção do trabalho(Seção 6).

2. Método

Esta seção tem como objetivo discutir as decisões que levaram à atual implementação do programa.

2.1 Fluxo principal

Antes de detalhar como o programa opera sobre um texto, vale discorrer de maneira textual e intuitiva o que o programa faz a cada passo de execução. Primeiramente, cada palavra presente no texto é lida e "**normalizada**", ou seja, todo caractere maiúsculo ou acentuado é substituído por seu respectivo par e cada pontuação presente é retirada. Munidos da nova ordem lexicográfica, o conjunto de palavras é **ordenado**, de forma que palavras repetidas ficam adjacentes e há a possibilidade de escolher quantos elementos serão

considerados na mediana. Após isso, é feita a **contagem** de quantas vezes cada palavra apareceu no texto, que fica salva em um arquivo.

2.2 Armazenamento

A fim de suportar as operações descritas acima, é necessário uma estrutura robusta o suficiente para guardar um superconjunto de palavras e ser possível de encontrar qualquer elemento de maneira rápida. Nesse sentido, a estrutura mais prática se mostrou o vetor dinâmico, visto que o mesmo possui todas as características necessárias, além de poder variar seu tamanho durante a execução. O tamanho inicial do vetor é de 10^5 elementos, número grande escolhido para não atrapalhar a execução de testes de performance, visto que são necessárias ao menos 10^5 palavras em um texto para que o programa gaste no mínimo 60 segundos. Sempre que o tamanho do texto for maior que o tamanho atual do array, ele é realocado com o dobro de elementos até chegar no limite de 10^6 palavras, medida de segurança para evitar overflow.

2.3 Normalização da Palavra

Como as palavras da saída precisam seguir uma formatação específica, se tornou válida a presença de uma função que "normalize" as palavras presentes. A normalização se trata de um processo que retira qualquer pontuação ao meio da palavra, além de transformar todas as letras maiúsculas em minúsculas. Para eliminar a pontuação, a forma mais cômoda encontrada foi armazenar os caracteres proibidos em uma string e a partir dela fazer buscas na palavra, se o i-ésimo elemento da palavra estiver entre os caracteres de pontuação, ele não é adicionado a nova string normalizada.

2.3 Normalização da Palavra [EXTRA]

Para tratar eventuais problemas no arquivo de entrada, a função de normalização é capaz de retirar os acentos da língua portuguesa presentes na palavra. Essa operação pode ser confusa ao usar o tipo string padrão de C++, isso porque caracteres especiais não são representados da mesma forma que os caracteres normais. Como uma string é uma lista de palavras, nem todos os 8 bits presentes no tipo char são usados para codificar seu conteúdo, seus bits iniciais são usados para codificar qual o próximo char presente na palavra, fazendo

com que um tipo char padrão de C++ só possa armazenar valores entre 0 e 127 da tabela ASCII, o que é um problema, já que todo caractere especial possui um valor maior que 127.

Dessa forma, um caractere especial é formado de no mínimo 2 posições de memória do tipo char, dificultando sua operação direta usando operações de strings. A forma usada para resolver o problema foi usar um tipo alternativo de string feita especialmente para lidar com esses “wide chars”. Convertendo a string do C++ em um vetor de char padrão, é possível contar quantas posições de memória cada caractere da string ocupa e usar essa informação para fazer um cast de string para wstring, um tipo onde comparações com caracteres especiais são implementadas e funcionam da forma com que deveriam. Para esse fluxo funcionar, usa-se a biblioteca locale para forçar que os caracteres advindos da entrada estejam no padrão UTF-8.

2.4 Ordenação QuickSort

Como especificado na ementa do trabalho, a ordenação é feita a partir do método QuickSort, que é um algoritmo de ordenação que usa dividir e conquistar para resolver o problema da ordenação. A premissa do QuickSort é, a partir de um pivô, dividir o conjunto em 2 sub partições menores, uma com elementos menores que o pivô e outra com elementos maiores que o pivô, fazendo isso sucessivamente para as sub partições tem como resultado um vetor ordenado. Entretanto, existem formas de tornar esse algoritmo mais eficiente, principalmente ao escolhermos com cautela qual elemento será o pivô e ao ordenar partições pequenas usando outros algoritmos mais simples que o QuickSort. Dessa forma, ambos comportamentos foram implementados na função.

2.5 InsertionSort

Para ordenar as partições menores e encontrar a mediana de uma partição, foi utilizado o algoritmo de ordenação InsetionSort. A ideia por trás dele é que o maior elemento do conjunto sempre estará na última posição, dessa forma, basta ir gradualmente comparando o primeiro elemento do vetor com todos os outros, até chegar no final, se o elemento comparado for menor ou igual com esse primeiro elemento, eles trocam de lugar, ao final da execução, todo elemento está na sua devida posição.

Sendo assim, o InsertionSort é útil para lidar com partições já ordenadas, visto que os elementos já estão em posição e nunca ocorreram trocas. Dessa forma, ele foi escolhido para ordenar as partições menores, já que elas possuem uma maior chance de já estarem ordenadas.

Da mesma maneira, ele foi escolhido para encontrar a mediana da partição. Se a partição tiver menos elementos que o necessário para calcular a mediana, é feita a mediana de 2 elementos, se a partição tiver menos que 2 elementos ela nunca chamará a função mediana, devido ao caso base do QuickSort.

2.6 Comparação de palavras

A principal operação da ordenação é decidir se uma palavra é menor que outra, no caso da nova ordem lexicográfica presente no problema, se torna válida a existência de uma função booleana que retorna se duas palavras são menores ou iguais. Para isso, foi criado um vetor de 26 inteiros que representa a nova ordem lexicográfica de cada uma das 26 letras do alfabeto. A fim de comparar duas palavras, basta procurar pela posição do primeiro caractere destoante entre elas e retornar se aquele caractere na posição encontrada é menor ou igual ao caractere na string de comparação. Se essa posição não existir, basta comparar o tamanho das strings, se elas tiverem o mesmo tamanho, elas são a mesma string, se não, a menor é aquela com o menor tamanho.

2.7 Contagem de palavras

Com as palavras já ordenadas no vetor de palavras, a contagem de palavras repetidas se torna trivial, visto que toda palavra repetida ficará adjacente dentro do vetor. Sendo assim, basta a criação de lista auxiliares para armazenar as palavras e quantas vezes elas aparecem, lendo linearmente a lista de palavras.

3. Análise de complexidade

3.1 Espaço

Inicialmente, é criado um vetor de palavras de tamanho 10^5, sendo assim, para textos com menos que 10^5 palavras, o espaço gasto é constante. Entretanto, se o texto tiver mais palavras que esse limite, mais memória é alocada e a complexidade se torna $O(N)$ na quantidade de palavras.

3.2 Tempo

Para análise de tempo, considere N o número de palavras do texto.

3.2 Ordenação

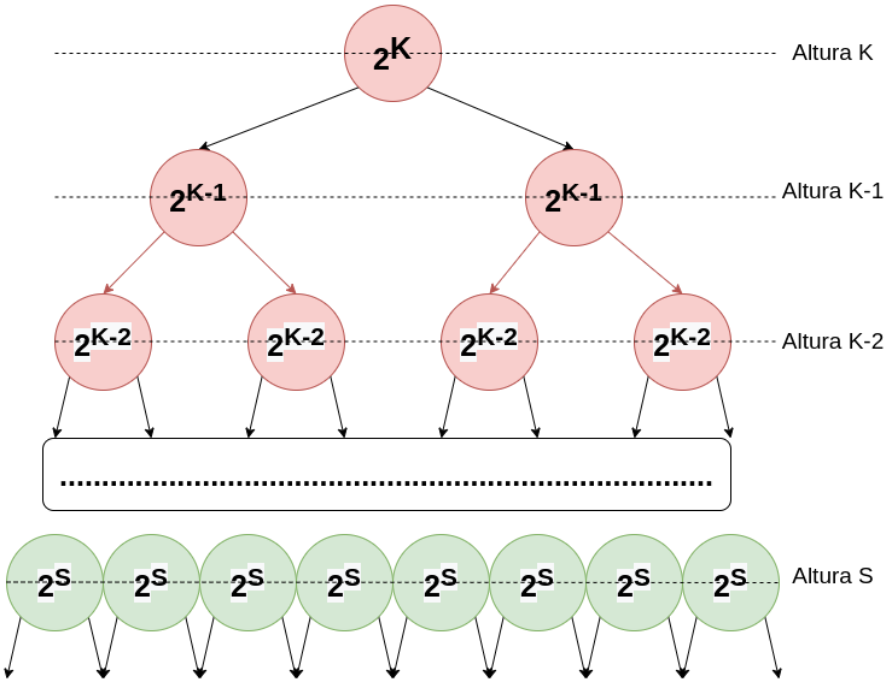
Para se realizar a ordenação, são utilizados dois algoritmos em conjunto, o QuickSort para lidar com partições muito grandes e o InsertionSort para ordenar partições menores e encontrar medianas. Considerando a comparação como operação mais significativa, é conhecido que o QuickSort efetua $O(N * log(N))$ comparações em vetores desordenados em média, já que a cada chamada do método todos os elementos são comparado uma vez e são realizados cerca de $log(N)$ chamadas. Entretanto, esse argumento é válido apenas nos casos que a escolha do pivô faz com que o conjunto seja dividido em 2 partes de tamanho próximo, se por acaso o pivô for o menor ou o maior elemento do conjunto, todos os demais elementos ficam dentro dentro da mesma sub-partição e a execução de passo não traz muitas melhoras na ordenação do conjunto. Essa propriedade é especialmente prejudicial em vetores já ordenados em ordem crescente ou decrescente, transformando o número de passos de $log(N)$ para N, gerando uma complexidade de $O(N^2)$ nesses casos, que são os piores.

Outrossim, existem formas de evitar com que esses casos aconteçam. A primeira maneira é utilizar o algoritmo de InsertionSort em partições pequenas. Tal método ajuda a mitigar o problema pois partições pequenas têm mais chances de estarem ordenadas em relação a partições grandes. Nesse sentido, se torna válida a análise do quão eficaz é essa medida:

3.2.1 InsertionSort Para partições menores

Para entender como o InsertionSort pode ser benéfico se usado em conjunto ao QuickSort, primeiro são necessárias algumas análises e premissas. O InsertionSort é $O(N)$ em vetores ordenados de forma crescente, já que o elemento usado para a comparação é sempre menor que o resto, nunca acontecendo comparações adicionais para mudar os elementos de lugar. Nos piores casos e nos casos médios, ele é $O(N^2)$, já que cada elemento vai ser comparado com todos os outros todas as vezes, em média. Além disso, podemos considerar que as entradas sempre será uma potência de 2, para facilitar a análise.

Sendo S o tamanho mínimo necessário para uma partição ser considerada pequena, podemos analisar qual o efeito que o InsertionSort terá em um QuickSort com entrada $N = 2^k$ da seguinte maneira:



Como N é uma potência de 2, a árvore de chamadas do método QuickSort corresponde a uma árvore binária completa de altura K com 2N + 1 nós, representando o total de chamadas recursivas feitas, onde a cada passo o tamanho da sub-partição corresponde a 2 elevado à altura do nó. Entretanto, cada subpartição com tamanho menor ou igual a S deve ser tratada como InsertionSort.

Encontrar quais nós possuem essa propriedade é fácil, se $S = 2^s$, todos os nós com altura maior que "s" vão ser tratados pelo QuickSort, logo, o QuickSort vai ser executado nos níveis entre 0 e (K - s), ou seja, entre 0 e $log(N/S)$. Em relação ao InsertionSort, ele será chamado apenas no nível "s", que possui 2^{K-s} nós, de tamanho 2^L cada. Sendo assim, a função de complexidade apenas do InsertionSort para partições pequenas é

$$g(N = 2^k) = 2^{K-s} * (2^s)^2 = 2^{K+s} = N * S$$

Da mesma maneira a função de complexidade do QuickSort fica:

$$h(N) = N * (K - s) = N * log(N/S)$$

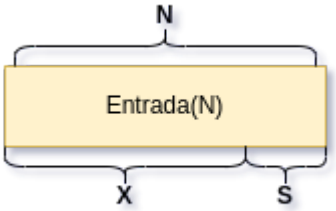
Sendo assim, a complexidade da ordenação no caso médio f(N) = g(N) + h(N) se torna:

$$f(N) = N * log(N/S) + N * S$$

Tomar S = N corresponde a executar o InsertionSort em todo conjunto, fazendo com que a complexidade se torne quadrática. Se S for um valor pequeno em relação a N, a ordem de complexidade se torna a mesma do QuickSort normal:

$$f(N) = N * log(N/O(1)) + N * O(1) = O(N * log(N))$$

De maneira análoga, se a entrada for um conjunto ordenado de maneira crescente, cada chamada do QuickSort diminui o tamanho da partição em 1, até que o tamanho do conjunto seja S:



$$X = N - S;$$

$$f(N) = \sum_{i=1}^{X} (N - i) + S$$

$$f(N) = \frac{X*(N-X+N-1)}{2} + S = \frac{(N-S)*(N+S-1)}{2} + S$$

$$f(N) = (N - S)^2 + S$$

O mesmo acontece com conjuntos ordenados de forma decrescente, ou seja, se o N for pequeno o suficiente para estar próximo de S - ou o S grande o suficiente para ser próximo de N - o melhor caso se torna linear e o caso médio fica quadrático, não havendo grande ganho assintótico entre esse método e o InsertionSort padrão. Entretanto, tal situação só acontece, porque o algoritmo ainda apresenta problemas na hora de lidar com conjuntos já ordenados, sendo válida a estratégia da mediana de M elementos:

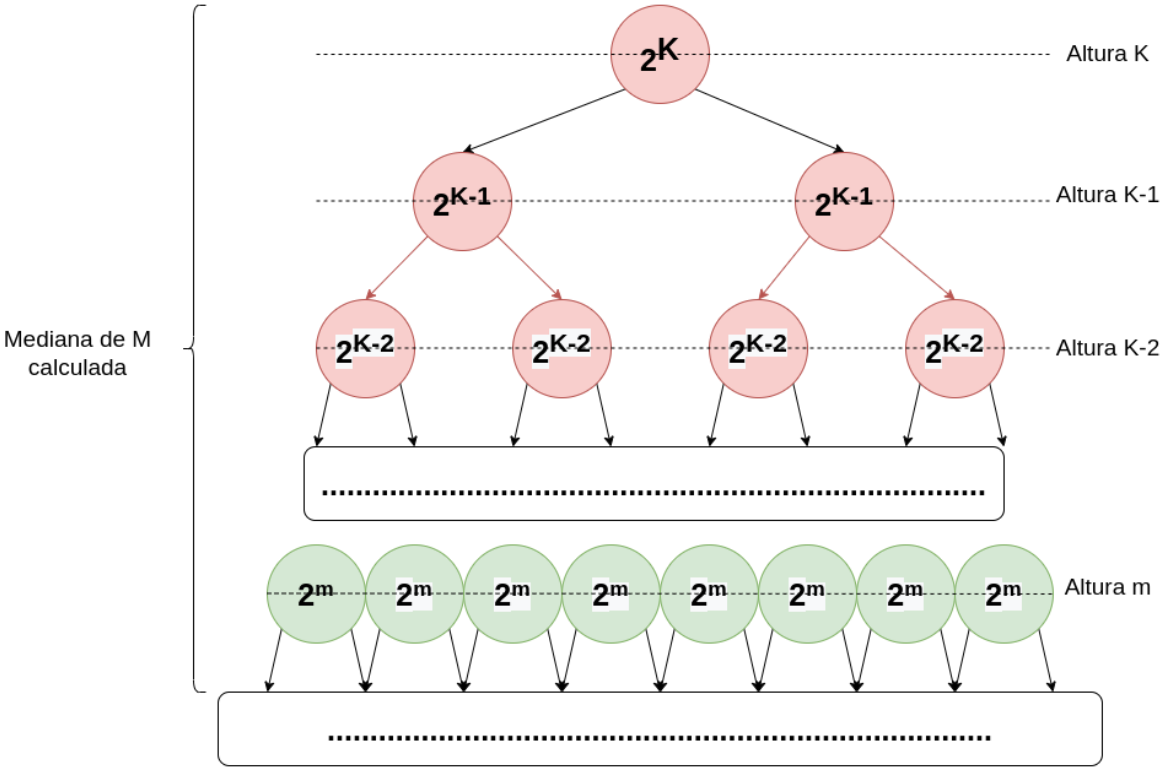
3.2.2 InsertionSort para mediana de M elementos

Entende-se como mediana de um conjunto como o valor central presente nele, sabendo a mediana de conjunto desordenado, é fácil separá-lo em 2 subconjuntos de com quase a mesma quantidade de elementos, tomando os elementos menores que a mediana para um conjunto e os maiores, para outro. No contexto do QuickSort, o pivô sempre corresponder à mediana de sua partição configura uma escolha ótima, já que a partição vai ser separada em

duas outras subpartições que também serão divididas sucessivamente, num total de log(N) vezes. Dessa forma, escolher a mediana como pivô garante que o QuickSort seja $O(\ Nlog(N))$ em qualquer caso. Entretanto, encontrar a mediana também possui um custo.

Como a mediana é o valor central da partição, a forma mais fácil de encontrá-la é ordenar a partição e escolher o elemento na posição $N/2$, só que para ordenar a partição é o problema que estamos tentando resolver. Uma outra alternativa é não pegar a mediana da partição inteira, e sim escolher a mediana dos primeiros M valores presentes como pivô e trabalhar com ela, garantindo um aumento na eficiência geral do algoritmo.

Ainda assim, essa opção também possui um custo, que pode ser estimado da mesma forma feita na seção anterior. Como discutido anteriormente, o uso da mediana faz com a função de complexidade do algoritmo fique $Nlog(N)$ + X em qualquer caso, onde X é o custo adicional para calcular as medianas. De maneira similar, a mediana de M elementos só será executada enquanto o tamanho da partição analisada for maior ou igual a M, se N = 2^K e M = 2^m, isso corresponde à todos os nós com profundidade menor que $K\ -\ m\ +\ 1$, ou seja, $2^{K-m+1}\ -\ 1$ ou $(2\frac{N}{M}\ -\ 1)$ nós.



Para cada um desses nós, é feita uma ordenação de M elementos usando o InsertionSort para se descobrir a mediana, essa operação é quadrática em M, logo, sendo d(N) a função para calcular as medianas de uma entrada de tamanho N:

$$d(N) = (2\frac{N}{M} - 1) * M^2 = 2NM - M^2$$

A função de complexidade para M > 2, que garante que nunca será escolhido ou menor ou o maior elemento da partição fica:

$$f(N) = Nlog(N) + d(N) = Nlog(N) + 2NM - M^2$$

Se M for menor que $log(N)$, a função f(N) é O(Nlog(N)) em todos os casos, garantindo uma execução próxima de ótima para o algoritmo independente do tipo de entrada, isto porque as partições que não usarem a técnica da mediana serão pequenas e ordenadas pelo InsertionSort, como dito na seção 3.2.1. Além disso, a combinação da mediana de M elementos e a ordenação simples para partições pequenas melhora ainda mais a constante envolvida na complexidade do algoritmo, visto que o uso da mediana proporciona uma maior incidência de subpartições já ordenadas dentre as chamadas recursivas do QuickSort, havendo casos onde a função de complexidade se torna

$$f(N,S,M) = Nlog(\frac{N}{S}) + S + 2NM - M^2 = O(Nlog(N))$$

se S e M forem suficientemente pequenos em comparação a N.

Por fim, conclui-se que, a depender das escolhas de S e M em relação a N, realizar mais operações em conjunto ao QuickSort apresenta melhoras em sua complexidade assintótica para alguns casos.

3.3 Busca

Considerando a comparação como a operação mais significativa, que toda busca pelas palavras acontece após o vetor ser ordenado e toda palavra repetida fica adjacente após a ordenação, para-se buscar as palavras repetidas, só é necessário percorrer o vetor uma vez, checando se o elemento atual é igual ao próximo. Dessa forma, contar a quantidade de vezes que cada palavra acontece no texto é $O(N)$.

3.4 Comparar Palavras.

Outro processo que é significativo na análise de complexidade do programa é o quão bem ele compara se uma palavra é menor que outra. A estratégia usada segue uma lógica sequencial, varre-se a menor palavra até encontrar o primeiro caractere que seja diferente entre as duas, se esse caractere na primeira palavra for menor ou igual à ordem lexicográfica

do caractere na mesma posição da segunda palavra, retorna verdadeiro. Se não existirem caracteres diferentes, a menor palavra é aquela que possui o menor tamanho. Dessa forma, a comparação é $O(L)$, onde L é o tamanho médio de uma palavra do texto.

3.5 Complexidade Geral.

Dito isso, a complexidade geral da ordenação corresponde ao produto entre o custo de comparar as palavras e a quantidade de comparações feitas, ou seja:

$$O(Nlog(N) * L)$$

4. Estratégias de Robustez

Ao lidar com listas de strings, a quantidade de memória disponível para o programa pode facilmente ser insuficiente para sua operação, dessa forma, são permitidos textos com no máximo 10^6 palavras.

5. Análise Experimental

A análise experimental a seguir tem como objetivo medir o quão eficiente é o sistema implementado usando duas métricas, o desempenho computacional - quão rápido o programa é executado com entradas grandes - e análises de acesso em memória.

5.1 Desempenho computacional

5.1.1 Perfil de execução

Para testar o desempenho computacional, primeiramente, o programa foi compilado em estado de "profiling", a fim de analisar quais funções consomem relativamente mais tempo durante a execução do programa. Diante disso, foram feitas baterias de testes a partir de textos retirados de Ebooks gratuitos, onde cada livro possui cerca de 10^5 palavras. Com o profile das execuções pronto, cada execução é processada pelo programa “gprof”, uma ferramenta que auxilia na análise do desempenho computacional. Dito isso, segue a análise das chamadas de funções.

Tempo de execução (%)	Calls	
46.24	49822106	isLessEqual(std::string,std::string)
16.18	81907	partition(std::string, int, int, int)
12.43	*48317830*	*void std::swap<string,string>*
10.40		*_init*
6.36	*49822106*	*unsigned long const& std::min<unsigned long>*
4.34	*97118*	*normalize(string)*
1.16	*84939*	*insertionSort(std::string, int, int, int)*
1.16	*1*	*leitura_arquivo(std::basic_ifstream<char)*
0.58	*81907*	*mediana_simples(std::string, int, int, int)*
0.58	*1*	*start_order(int*)*

Para evitar redundância, os outros perfis de execução apresentam o mesmo comportamento do teste apresentado acima. No que tange a análise dos experimentos, fica evidente que a função que mais demanda tempo, em porcentagem, é a função para comparar se uma palavra é menor que outra, já que essa é a operação principal da ordenação. Além disso, a função apresenta complexidade linear no tamanho das palavras e é chamada cerca de $3 * Nlog(N)$ vezes durante o programa, usando a fórmula desenvolvida na seção 3.2.2. Dessa forma, a função de comparação estar em primeiro lugar não é de grande surpresa, embora existam formas de diminuir esse tempo de execução usando de técnicas como hashing.

A função que ocupa a segunda posição é a função que particiona o vetor durante o QuickSort.

O terceiro lugar se trata da função que troca dois elementos do vetor de posição. Como o texto utilizado no teste foi retirado de um livro, existem muitas palavras repetidas, que faz com que muitas trocas de posições ocorram, visto que a operação necessária para executar uma troca é a operação menor ou igual. Nesse sentido, utilizar um comparador apenas de igualdade teria diminuído o número de trocas e deixaria o programa mais eficiente.

A função _init que ocupa o quarto lugar serve para carregar e compartilhar as funções de diferentes arquivos durante a execução do programa. Implementar a função de comparação no mesmo arquivo das funções de ordenação resolveria o problema.

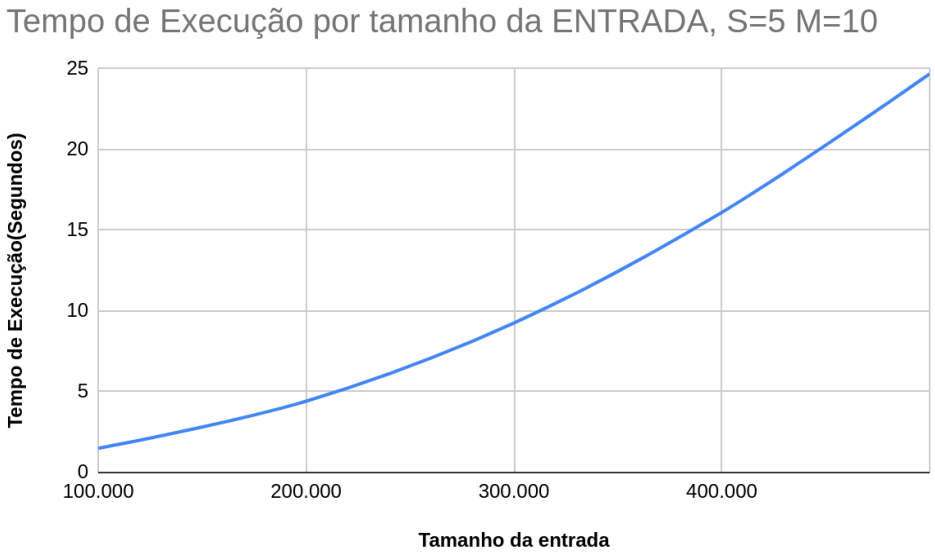
Por fim, a última função válida para se comentar é a normalize, que representou 4% do tempo de execução do programa. Esta é a função usada para retirar letras maiúsculas, acentos

e pontuações da palavra antes dela ser inserida. Para cada palavra no arquivo de entrada, a função normalize é chamada, fazendo com que ela tenha complexidade linear no tamanho da entrada, sendo justo ela ser significativa no tempo de execução para a ordenação de textos com muitos acentos.

Nesse sentido, conclui-se que ainda existem margens para a melhora no desempenho do programa, a nível de chamadas de funções, visto que muitas delas poderiam ser feitas de outras formas, a fim de melhor utilizar tempo de execução do programa.

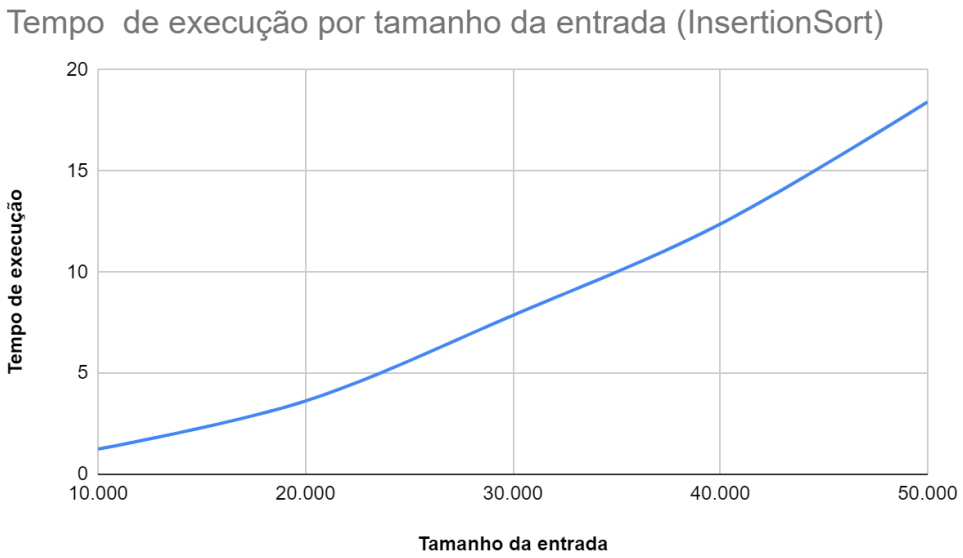
5.1.2 Desempenho em termos da entrada

Além do tempo gasto pelas chamadas de funções, deve-se medir a relação entre o aumento do volume da entrada do programa com o seu tempo de execução. Para realizar tal objetivo, foi utilizada a ferramenta Memlog do professor Wagner Meira e ferramentas para a geração de gráficos. A seguir, o gráfico com os resultados.

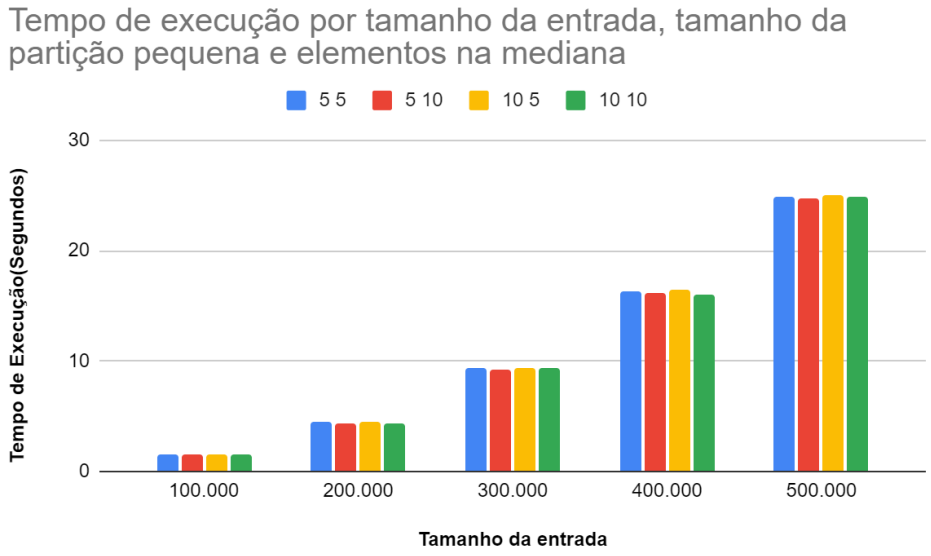


Como visto acima, o tempo de execução cresce muito próximo ao nlog(n), assim como foi discutido na seção de análise de complexidade.

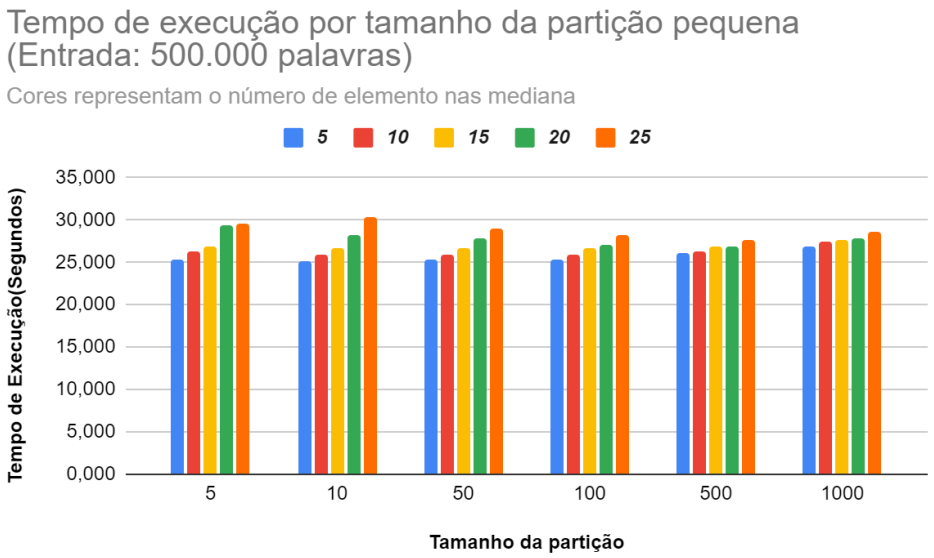
Para confirmar que o programa não performava de maneira quadrática, foram realizados testes usando apenas o InsertionSort para ordenação:



Embora os dois gráficos sejam parecidos, o InsertionSort recebeu entradas da ordem de 10^4 para ficar com um tempo de execução similar ao QuickSort com entradas de 10^5.



O gráfico acima mostra o que acontece ao variar a escolha da mediana e do tamanho da partição pequena com base no tamanho da entrada. Com uma análise mais detalhada, é possível observar que a escolha desses parâmetros não afeta muito o tempo de execução para textos com menos que meio milhão de palavras, embora textos grandes se beneficiem dessa técnica. Para elucidar tal fato, foram feitas baterias de testes com o texto de 500.000 palavras:



A partir do gráfico, é possível notar escolher mediana de M elementos com M desnecessariamente grande pode piorar o tempo de execução do programa, tal fato é visto ao avaliar que cada coluna colorida é maior do que a coluna anterior, para qualquer tamanho de partição. Além disso, os menores tempos de execução ficaram nas escolhas de partição próximas a Nlog(N), que nesse caso é 18.

Para além disso, foram feitos testes com valores de mediana e de partição extremos, o primeiro sendo S = 50 e M = 500, que executou em 605 segundos, ou 10 minutos, e o caso S = 500.000 M = 1, que executou em 1717 segundos, ou 28 minutos, que representa o valor N^2 para essa entrada. Por fim, vale ressaltar que o tempo de execução médio para o QuickSort nessa entrada foi 25 segundos.

5.2 Análise de localidade de referência e acesso de memória

Como explicado em aula, programas eficientes em uso de memória são aqueles que tendem a acessar regiões de memória contíguas, já que essa prática é otimizada pelos sistemas operacionais onde o programa é executado. Tal propriedade é chamada de localidade de referência e é uma ótima métrica de como o programa acessa a memória por ele utilizada.

Nesse sentido, munido das ferramentas Analisamem e Memlog, disponibilizadas pelo professor Wagner Meira, é possível gerar gráficos que demonstram o acesso de memória do programa em suas diversas fases, e, assim, analisar se o acesso de memória do programa segue as boas práticas desejadas. Dito isso, segue a análise de acesso de memória e localidade de referência, onde o id 0 representa a inserção dos elementos no vetor, o id 1 a ordenação das partições pequenas pelo InsertionSort e o id 2 como os acessos do QuickSort.

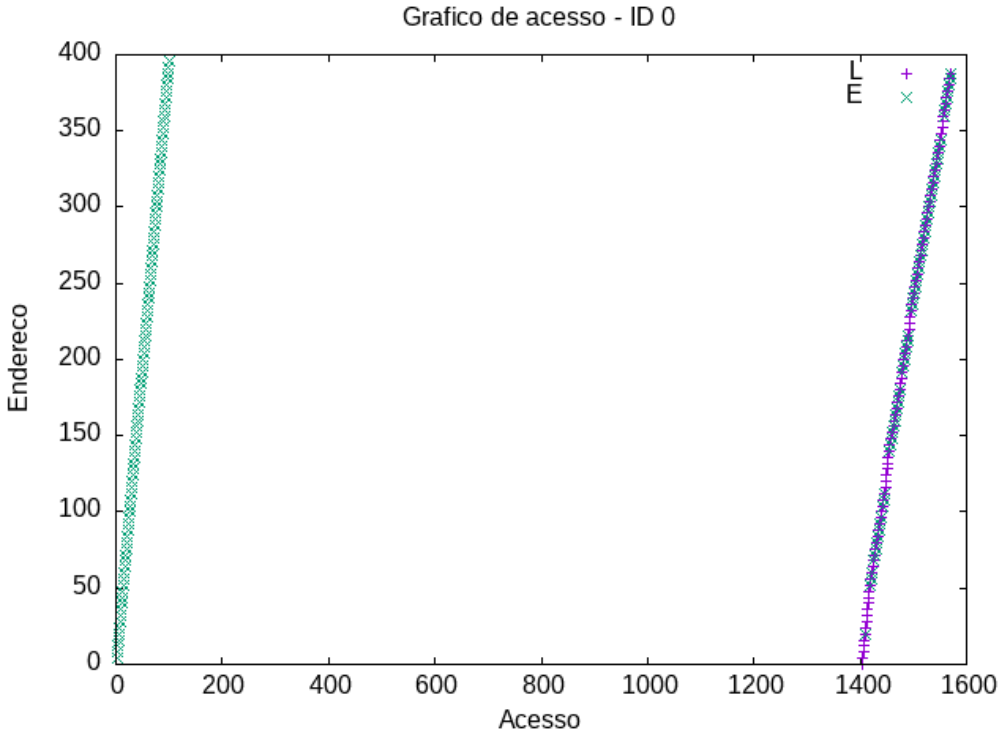


Grafico de acesso - ID 0
Endereco
Acesso
L
E
0
50
100
150
200
250
300
350
400
0
200
400
600
800
1000
1200
1400
1600

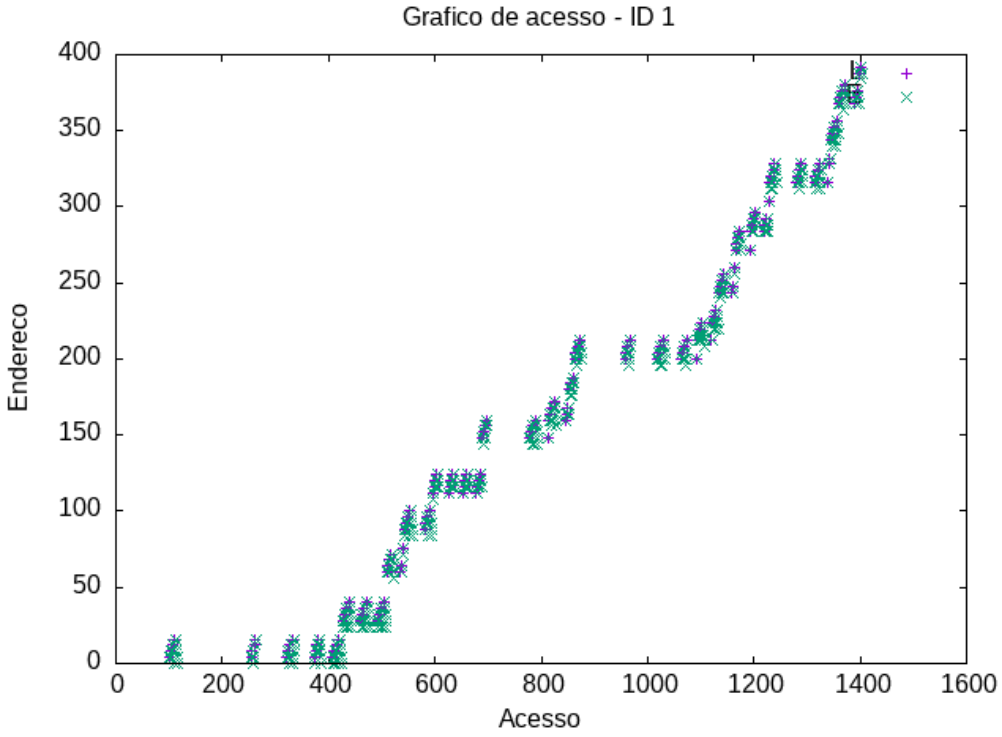
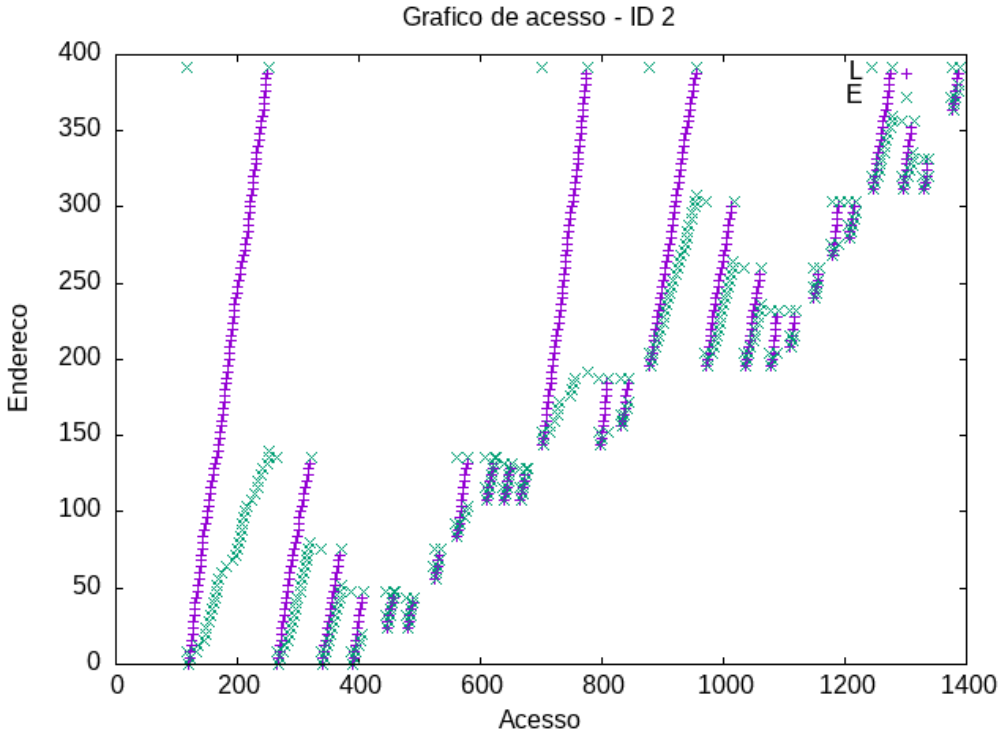
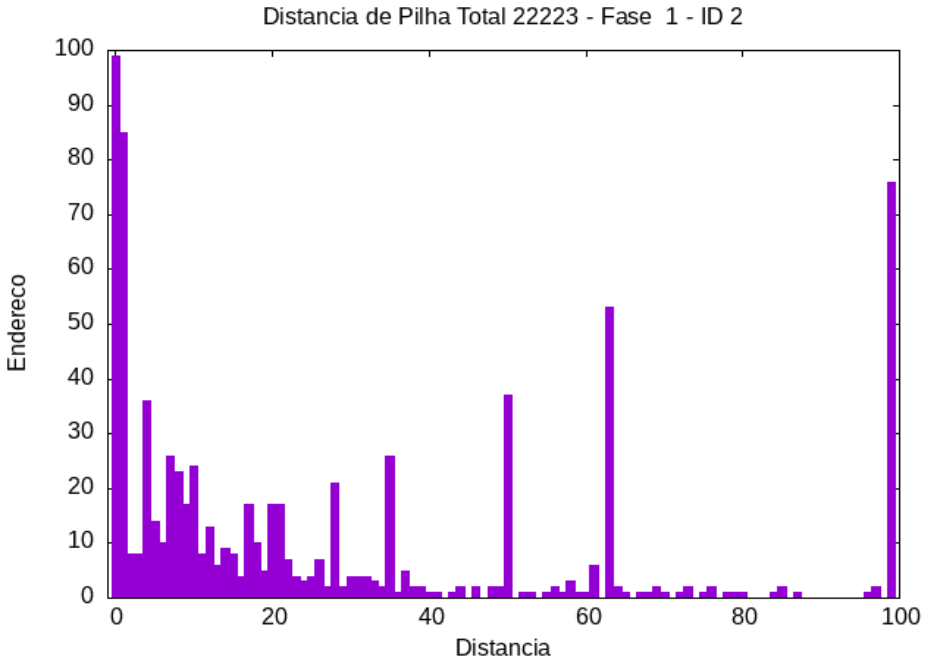
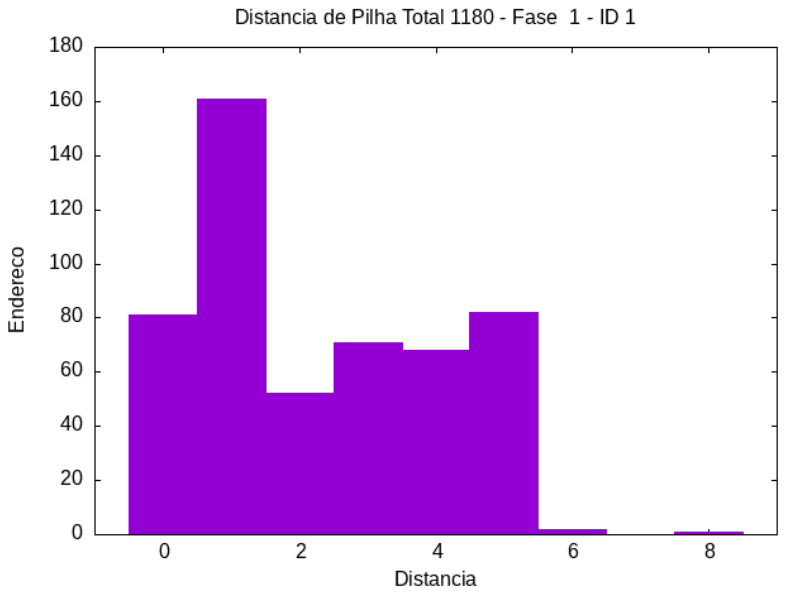


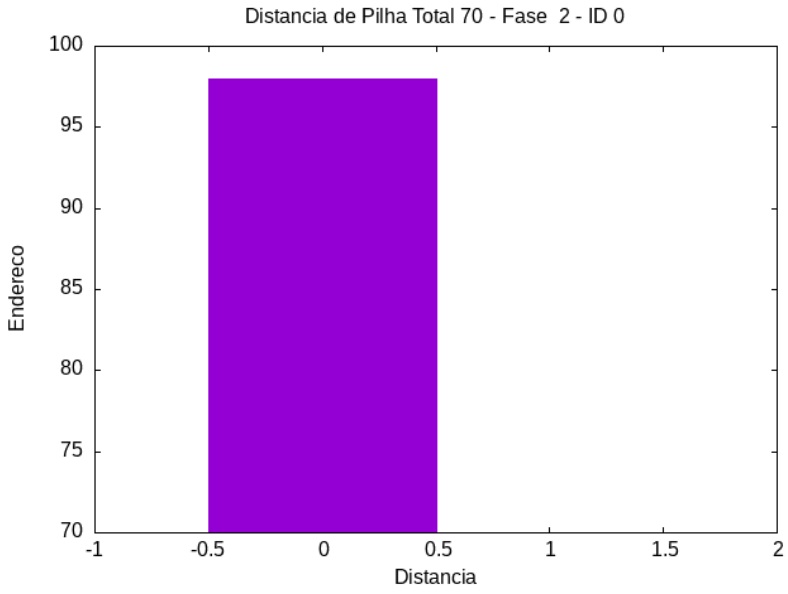
Grafico de acesso - ID 1
Endereco
Acesso
L
E
0
50
100
150
200
250
300
350
400
0
200
400
600
800
1000
1200
1400
1600



Como visto acima, o programa acessa diversas vezes posições de memória adjacentes, isso acontece porque os elementos estão dispostos por todo o vetor. Dessa forma, é esperado que a distância de pilha seja alta, visto que elementos distantes na memória são acessados e trocados de posição durante toda a execução do programa.







Dessa forma, conclui-se que o sistema nem sempre mantém uma baixa distância de pilha durante a execução, desfavorecendo a hipótese do acesso em posições contíguas de memória e sendo uma flocalidade de referência.

6. Conclusões

Com o intuito de contar quantas palavras existem em um texto, e apresentá-las em uma nova ordem lexicográfica, foi implementado um programa que utiliza algoritmos de ordenação de forma não trivial para resolver o problema.

Durante o projeto do sistema foram levadas em consideração não só aspectos práticos do processamento de texto, mas também como a linguagem de programação escolhida poderia ser uma ferramenta útil para chegar no objetivo esperado. Toda a questão de mapear um mini-mundo de interesse em um modelo computacional robusto se mostrou bastante produtiva, levando o aluno a pensar em formas criativas de se resolver e entender o problema, tendo como resultado uma extensa discussão sobre como os métodos da mediana e da ordenação simples para partições pequenas afetam o desenvolvimento do programa. Por fim, o tempo extra usado para projetar o sistema trouxe várias recompensas no sentido da implementação, sendo um aspecto a ser levado para trabalhos futuros.

Além disso, os testes e a análise de complexidade se mostraram muito úteis para entender como as entradas influenciam a execução do programa e como alguns detalhes de implementação podem ser significativos em termos de desempenho computacional e localidade de referência.

Nesse sentido, todo o fluxo de trabalho foi essencial para a consolidação de conteúdos aprendidos em sala, além de apresentar, de forma prática, como softwares maiores, mais consistentes e robustos são projetados e implementados.

7. Bibliografia:

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas . Editora Cengage.

Paulo Feofiloff, Notas de Aula da disciplina Projeto de Algoritmos (2018). DCC-IME-USP. disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/

8. Instruções para compilação e execução:

8.1 Compilação

Existem partes do programa que são compatíveis apenas às versões mais recentes da linguagem c++, dito isso, deve-se seguir as seguintes configurações para a compilação:

Linguagem: C++
Compilador: Gnu g++
Flags de compilação: -std=c++11 -g
Versão da linguagem: standard C++11
Sistema operacional (preferência): distribuições baseadas no kernel Linux 5.15.

O comando para compilar o programa automaticamente está presente no arquivo **"Makefile"** e sua execução é chamada pelo comando **"make all"**. Ainda assim, seguem as instruções para compilar manualmente:

Para gerar o executável do programa, é necessário, primeiro, gerar o objeto para cada arquivo presente na pasta "/src". Tal objetivo pode ser alcançado seguindo os seguintes comandos em ordem:

```
g++ -g -std=c++11 -Wall -c src/main.cpp -o obj/main.o -I./include/
g++ -g -std=c++11 -Wall -c src/memlog.cpp -o obj/memlog.o -I./include/
g++ -g -std=c++11 -Wall -c src/ordenacao.cpp -o obj/ordenacao.o -I./include/
g++ -g -std=c++11 -Wall -c src/str_funcs.cpp -o obj/str_funcs.o -I./include/
```

Após esse passo, deve-se juntar todos os objetos em um único arquivo executável, seguindo o comando:

```
g++ -g -std=c++11 -Wall -o ./bin/tp2.out ./obj/main.o ./obj/memlog.o ./obj/ordenacao.o ./obj/str_funcs.o
```

Deste modo, o executável "/bin/tp2.out" estará compilado e pronto para ser utilizado.

8.2 Execução

Seguem as instruções para a execução manual:

1. Certifique-se que o compilável foi gerado de maneira correta, se algum problema ocorrer, execute o comando “make all” presente no “Makefile”.
2. Dado que o compilável foi gerado de maneira correta, certifique-se que o arquivo de entrada existe. Se ele não existir, crie-o.
3. Certifique-se que o arquivo de entrada está na formatação esperada, onde existe um marcador #ORDEM para marcar a nova ordem lexicográfica e o marcador #TEXTO para marcar o texto a ser lido
4. Uma vez que os passos anteriores foram cumpridos, execute o programa com o comando: ./bin/tp2.out -i (entrada) -o (saida) -s (tamanho particao) - m (valor da mediana de m)
5. A saída estará guardada no arquivo de saida.