Trabalho Prático 2 - Métodos de Ordenação

Lucas Almeida Santos de Souza - 2021092563[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

luscaxalmeidass@ufmg.br

1 Introdução

Neste trabalho, será analisado o desempenho de diferentes algoritmos de ordenação sob diferentes circunstâncias. Os objetos serão ordenados com base em uma chave presente em seus atributos. Para analisar o desempenho das ordenações, serão levados em consideração três fatores: a quantidade de comparações de chaves, o número de cópias de registros realizadas e o tempo de processamento total gasto para a realização da ordenação.

Para isso, serão utilizados conceitos de algoritmos de ordenação dos tipos a seguir:

- **Quicksort:** essa ordenação utiliza o conceito de "dividir para conquistar". Ela particiona o vetor ao meio em um pivô, mantendo os valores menores que esse pivô à esquerda e os valores maiores à direita. Depois, para cada partição, o programa repete o processo, particionando-a ao meio novamente. **Este link** contém uma animação demonstrando o funcionamento do Quicksort.
- **Selection Sort:** essa ordenação percorre o vetor em busca do menor valor e substitui pelo valor da primeira posição. Em seguida, percorre novamente a partir da segunda posição e insere o segundo menor valor na segunda posição. Isso se segue até o último valor. **Este link** contém uma animação demonstrando o funcionamento do Selection Sort.
- **Merge Sort:** essa ordenação também utiliza o conceito de "dividir para conquistar"em seu funcionamento. O programa divide o vetor ao meio recursivamente, ordenando cada uma das metades e, em seguida, unindo-as em um único conjunto ordenado. **Este link** contém uma animação demonstrando o funcionamento do Merge Sort.
- **Heap Sort:** essa ordenação é um tipo de ordenação por seleção. O programa insere os elementos em um heap - estrutura que pode ser vista como uma árvore binária - e, a cada iteração, retira o maior valor, que está presente na raiz, e o adiciona ao final do vetor ordenado. **Este link** contém uma animação demonstrando o funcionamento do Heap Sort.

O trabalho consiste de duas partes principais. Na primeira, será analisado o impacto de algumas versões do Quicksort, cada uma com uma lógica de implementação distinta. Na segunda parte, a análise será feita comparando o melhor Quicksort obtido na primeira parte com a

ordenação Merge Sort e Heap Sort.

1.1 Versões do Quicksort

Na primeira parte, serão implementadas 5 versões distintas da ordenação Quicksort, vistas a seguir:

Quicksort Recursivo

Essa versão do Quicksort é a mais conhecida e, apesar de ser a mais simples de se implemetar, nem sempre é a mais eficaz por conta de seu uso de recursividade. O algoritmo dessa função recebe o vetor, particiona-o segregando os valores maiores e menores que o pivô e, em seguida, chama a própria função para cada uma das metades.

Quicksort Mediana

A segunda versão implementada contém uma alteração na escolha do pivô do particionamento. Em vez de selecionar o pivô como sendo o elemento do meio do vetor, o programa escolhe aleatoriamente k elementos do vetor - sendo k um parâmetro passado pelo usuário - calcula a mediana desses k elementos e essa mediana é escolhida como o pivô da partição. Depois disso, o programa segue similar ao Quicksort Recursivo.

Quicksort Seleção

A terceira implementação do Quicksort recebe, além do vetor e do tamanho, uma constante m, que corresponde a um limiar. Enquanto o tamanho do vetor for maior que o limiar, a função utilizará a mesma lógica do Quicksort Recursivo para ordená-lo. Quando o tamanho do vetor for menor que esse limiar, ela utilizará o Selection Sort.

Quicksort não Recursivo

A quarta implementação consiste de uma versão do Quicksort sem utilizar recursividade, utilizando uma estrutura de Pilha Encadeada para ordenar as partições do vetor. Os elementos dessa pilha contém um inteiro indicando o início da partição e um indicando o fim, ou seja, cada elemento da pilha guarda a informação de um intervalo do vetor. Assim, enquanto a pilha não estiver vazia, o programa insere os intervalos de cada partição esquerda e depois direita do vetor até que seja inserido o menor intervalo. Quando isso acontece, o programa desempilha o valor desse intervalo (que corresponde ao menor valor do vetor) e continua a execução.

Quicksort Empilha Inteligente

A última versão do Quicksort implementada é bastante similar ao Quicksort não Recursivo, porém ao invés de inserir a partição esquerda e depois a direita, o programa insere a menor das duas partições na pilha em cada iteração.

2 Método

2.1 Especificações

O código foi desenvolvido na linguagem C++, compilada pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional Linux Ubuntu 22.04
- Processador intel core i5 8ª geração
- 8GB de memória RAM

2.2 Execução

Para realizar o trabalho, foram utilizados os códigos dos algoritmos de ordenação apresentados nas aulas de Estruturas de Dados, com algumas alterações para cumprir os requisitos. Para o Quicksort não Recursivo e o Quicksort Empilha Inteligente, foi implementado também o TAD Pilha.

Além disso, foi criado uma biblioteca somente para declaração das variáveis globais utilizadas para receber os argumentos da linha de execução do código e para contabilização das estatísticas usadas na análise de desempenho.

Variáveis Globais

- **Namespace reg:** Namespace criado para agrupar as variáveis globais `tempo`, `copias` e `comparacoes`, usadas durante todo o programa para contabilizar as métricas da análise de desempenho.
- **Namespace args:** Namespace criado para agrupar as variáveis globais utilizadas para receber os argumentos na linha de execução do programa, que são usadas durante o programa em suas respectivas funções.

Classes

- **Item:** Registro inserido nos vetores para ordenação. Essa classe contém um vetor de 10 números reais, um vetor de 15 cadeias de caracteres com tamanho de 200 caracteres cada e uma chave inteira, utilizada para a ordenação.
- **ItemPilha, Node e Pilha:** Classes do TAD Pilha visto em sala, com alterações na classe ItemPilha, que tem como TipoItem um intervalo de um vetor, demarcado pelos índices inteiros de suas posições inicial e final.

Funções utilizadas para as ordenações

Além das funções de ordenação e suas auxiliares, foram implementadas outras funções para implementar as variações do Quicksort:

- **Troca( ):** Recebe o endereço de duas variáveis do tipo Item e troca seus valores.

- **Particao( ):** Realiza o particionamento de um vetor, com os valores maiores que ele à direita e os valores menores à esquerda. Pode receber um pivô personalizado, mas por padrão selciona o elemento intermediário do vetor.

Funções utilizadas para a execução do programa

- **chamaOrdenacao( ):** Chama a respectiva função de ordenação de acordo com a versão indicada pelo usuário.
- **contagem( ):** Conta o tempo total do programa (somando os tempos de usuário e de sistema) no momento da chamada da função.
- **preencheVetor( ):** Recebe um vetor de Item e seu tamanho e preenche as chaves com números aleatórios utilizando a função `lrand48()`.
- **parseArgs( ):** Recebe os argumentos da linha de execução e os armazena nas respectivas variáveis globais.
- As funções **nomeOrdenacao( )**, **inicializaSaida( )** e **registraNoArquivo( )** são utilizadas na formatação do arquivo de saída.

3 Análise de Complexidade

Essa seção se trata de analisar a complexidade de cada um dos algoritmos de ordenação citados. Vale ressaltar que a função de tempo geral do programa pode ser afetada pois, para cada vez que o código é executado, o algoritmo de ordenação escolhido é chamado n vezes, para cada um dos tamanhos descritos no arquivo de entrada.

3.1 Selection Sort

Essa função percorre o vetor de tamanho n a cada iteração, comparando o atual menor valor com todos os próximos em busca de um menor. Como cada iteração tem tempo de execução linear $O(n)$, ao percorrer n iterações, o programa tem uma complexidade de tempo $O(n^2)$ para todos os casos, ou seja, independente da forma com que o vetor esteja inicialmente ordenado.

3.2 Quicksort

Todas as versões do quicksort utilizam a função `Particao()`. Essa função define um pivô em tempo constante $\Theta(1)$ e, em seguida, percorre simultaneamente do início ao meio e do fim ao meio em tempo $\Theta(\frac{n}{2})$ cada, totalizando assim uma complexidade de $\Theta(1) + 2\Theta(\frac{n}{2}) = \Theta(n)$.

Um algoritmo de quicksort comum - como é o caso do **Quicksort Recursivo**, **Quicksort não Recursivo** e **Quicksort Empilha Inteligente** - utiliza a função `Particao()` e, em seguida, executa a função para cada uma das metades geradas. Assim, temos a função de complexidade dada por

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n), & n > 1 \end{cases} \quad (1)$$

Utilizando o teorema mestre, podemos concluir que a complexidade total de tempo dessas três funções pode ser definida por $O(nlogn)$.

Já o **Quicksort Seleção** recebe um argumento m e ordena o vetor utilizando Quicksort quando o tamanho é maior que m e Selection Sort quando o tamanho é menor ou igual a m. Assim, esse método tem sua função de complexidade de tempo definida por

$$T(n) = \begin{cases} O(nlogn), & n > m \\ \Theta(n^2), & n \leq m \end{cases} \quad (2)$$

Porém, além da complexidade de tempo, é válido considerar que uma função recursiva gasta mais tempo e memória, pois cada iteração causa uma nova chamada de função pelo sistema. Por isso, mesmo que a complexidade do Selection Sort seja maior, como ele não utiliza recursividade, ele é mais eficiente para ordenar vetores menores, tornando o Quicksort Seleção mais eficiente dependendo do valor de m.

Por fim, o **Quicksort Mediana** utiliza um método que apesar de gastar mais tempo, diminui a probabilidade de acontecer o pior caso. Para isso, ela recebe uma variável k, aloca um vetor auxiliar desse tamanho para armazenar valores aleatórios do vetor original e seleciona como pivô a mediana desse vetor auxiliar. Para encontrar a mediana, foi utilizado o Selection Sort para ordenar o vetor auxiliar e um cálculo de tempo constante $\Theta(1)$ para pegar o elemento médio do vetor, que corresponde à mediana. Esse elemento é então usado no pivô da partição do Quicksort. Vale lembrar novamente que, apesar de não ter uma complexidade de tempo muito favorável, o Selection Sort foi escolhido para ordenar o vetor auxiliar pois, como não utiliza recursividade, é mais eficiente para ordenação de vetores pequenos.

Temos então que a complexidade de tempo do Quicksort Mediana é $\Theta(k^2) + \Theta(1) = \Theta(k^2)$ para ordenar o vetor auxiliar e pegar sua mediana, repetindo esse procedimento para cada subvetor particionado. No melhor caso, o pivô escolhido aleatoriamente será sempre o item mediano do vetor, dividindo-o em duas partições de tamanho $\frac{n}{2}$. Nesse caso, a complexidade de tempo total da função será $O(k^2nlogn)$. No pior caso, o pivô escolhido será sempre o i-ésimo primeiro ou último valor do vetor, dividindo o vetor em partições de tamanho 0 e n-i. Nesse caso, a complexidade de tempo total da função se torna $O(k^2n^2)$. Se considerarmos o crescimento da função de complexidade para um k constante, podemos ter as complexidades de $O(nlogn)$ para o melhor caso e $O(n^2)$ no pior caso, assim como as outras versões do Quicksort.

3.3 Merge Sort

Essa ordenação calcula o ponto médio do vetor de tamanho n e, recursivamente, resolve cada um dos dois subvetores gerados de tamanho $\frac{n}{2}$, juntando-os depois com uma função de complexidade $\Theta(n)$. Como é uma recursiva, a função de complexidade pode ser definida como

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(\frac{n}{2}) + \Theta(n), & n > 1 \end{cases} \tag{3}$$

Expandindo e resolvendo a função, temos que a complexidade de tempo é dada por $\Theta(nlogn)$.

3.4 Heap Sort

Por fim, essa ordenação entende o vetor como uma árvore, onde cada posição i tem como filhos as posições $2i+1$ e $2i+2$. Ao consruir essa árvore, ela compara os valores e realiza trocas de modo que, ao final de uma iteração, o maior valor esteja na raiz. Por fim, ela troca a raiz pela folha mais baixa (visto como a última posição do vetor) e continua a execução com todos os valores menos o último, já ordenado. Temos então que a função percorre todo o vetor em tempo $\Theta(n)$ e, a cada iteração, realiza comparações em uma árvore binária, de tempo $\Theta(logn)$, totalizando uma complexidade de tenpo geral de $\Theta(nlogn)$.

3.5 Complexidade de espaço

Quase todos os algoritmos utilizam apenas o vetor principal para realizar a ordenação, contendo assim complexidade de espaço $O(n)$. O Quicksort Mediana, aloca um vetor de tamanho k para armazenar os valores aleatórios do vetor e calcular sua mediana. Assim, inicialmente podemos calcular sua complexidade de espaço como $O(n+k)$, ou $O(max(n,k))$. Porém, se analisarmos a função em n suficientemente grande, temos sua complexidade similar à dos outros algoritmos, $O(n)$.

O Merge Sort aparenta ter uma complexidade de espaço diferente, pois em cada iteração, aloca vetores com a metade do tamanho total para serem auxiliares no momento da ordenação. Seguindo esse raciocínio, a complexidade de espaço total seria $O(nlogn)$. Porém, ao terminar de processar cada menor conjunto possível de subvetores, a função desaloca o espaço antes de prosseguir. Assim, a maior quantidade de espaço que ele ocupará simultaneamente é $n+\frac{n}{2}+\frac{n}{2^2}+\frac{n}{2^3}\cdots \approx 2n$, ou seja, $O(n)$.

3.5.1 Memória

Ao ordenar um vetor de 1 milhão de registros de 3044 bytes cada, os algoritmos que têm complexidade de espaço igual a $O(n)$ ocupam aproximadamente $2,83$GB de RAM durante toda sua execução.

$$1000000 \cdot 3044 = 3,044 \cdot 10^9 \text{bytes} \approx 2,83\text{GB} \tag{4}$$

Como dito anteriormente, o Merge Sort aloca recursivamente subvetores de tamanho $\frac{n}{2}$ para ordená-los, ocupando mais memória que os outros algoritmos. Como sua função de complexidade de espaço é $2n$, a maior quantidade de memória alocada simultâneamente no Merge Sort será de aproximadamente $2,83 \cdot 2 = 5,7$GB.

4 Estratégias de Robustez

Para tratar da robustez do código, foram usadas as macros da biblioteca `msgassert.h`. Na função `parseArgs()`, o programa confere se foram passados arquivos de entrada e saída válidos. Caso não tenha sido informado um arquivo de entrada, o programa é terminado, mas caso não seja informado um arquivo de saída, o programa simplesmente cria um arquivo padrão na pasta bin e informa ao usuário o local e nome do arquivo. Também há um `avisoAssert()` para impedir que o usuário insira um número de versão de ordenação não válida. Caso ele insira um número de versão não válido ou não insira, o programa coloca por padrão o Quicksort Recursivo, pois se a execução for interrompida, pode haver *memory leak*.

Além disso, para as variáveis globais m (limiar do Quicksort Seleção), k (tamanho do vetor auxiliar do Quicksort Mediana) e a semente da função `rand48()`, caso o usuário não insira algum desses argumentos ao escolher a versão ou insira um número negativo, o programa insere nas variáveis o valor arbitrário de 10 para a variável m, 3 para a variável k e 10 para a semente, para que as funções ainda sim possam ser executadas sem problemas ou exceções. Ainda sim, é colocado um aviso para o usuário explicando o erro e informando a medida tomada.

5 Análise Experimental

5.1 Método

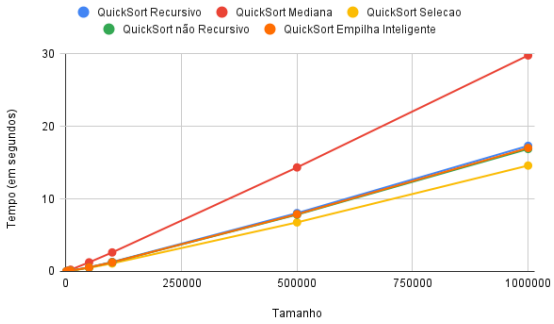
Como dito na seção Execução, para se calcular as três métricas (tempo, número de comparações de chave e número de cópias de registro) foi utilizado um namespace com variáveis globais para que se pudesse utilizá-las por todo o código. Os testes foram feitos utilizando um arquivo *shellscript* executado após o computador ser completamente reiniciado, para que a performance dos testes fosse afetada o mínimo possível.

Cada método de ordenação recebeu 5 testes com sementes aleatórias, com exceção do Quicksort Mediana que recebeu 5 testes para cada valor de $k = 3, 5, 7$ e do Quicksort Seleção que recebeu 5 testes para cada valor de $m = 10, 100$. Para manter consistência nos testes, o arquivo de entrada foi o mesmo, um arquivo `.txt` com sete tamanhos distintos para os vetores a serem ordenados: mil, 5 mil, 10 mil, 50 mil, 100 mil, 500 mil e 1 milhão. Durante a execução dos testes, cada saída foi armazenada em um arquivo distinto e, após o fim dos testes, os dados foram passados para uma planilha para calcular a média dos resultados de cada ordenação e gerar os respectivos gráficos.

5.2 Resultados

Tempo

Para a análise de tempo das funções, foi utilizada a função `getrusage()` da biblioteca padrão do C. Ela é chamada logo antes e logo após a execução do algoritmo de ordenação e é retirada a diferença dos tempos para obter o tempo exato da execução da ordenação, desconsiderando o tempo de funções anteriores a isso.



(a) Tempo de execução das versões do Quicksort

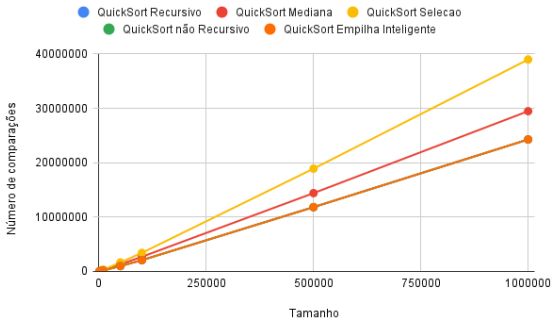
(b) Tempo de execução do Quicksort X Heap Sort X Merge Sort

Podemos observar que as ordenações Quicksort Recursivo, Quicksort não Recursivo e Quicksort Empilha Inteligente têm tempos bem similares, pois a lógica implementada é a mesma. O Quicksort Mediana tem um tempo bem maior, pois há o custo de ordenar o vetor de medianas e o Quicksort Seleção é o mais rápido, provando a afirmação que o Selection Sort é o mais eficiente para ordenar vetores pequenos.

Por ser o mais rápido, o Quicksort Seleção foi escolhido para ser comparado com o Merge Sort e Heap Sort e podemos ver que as duas ordenações têm tempos similares e maiores que o Quicksort escolhido.

Comparações

Nas análises de números de comparações de chaves e cópias de registros, as variáveis globais são inicializadas como 0 antes da chamada das funções e, quando ocorre uma comparação ou cópia, é incrementado o valor da variável.



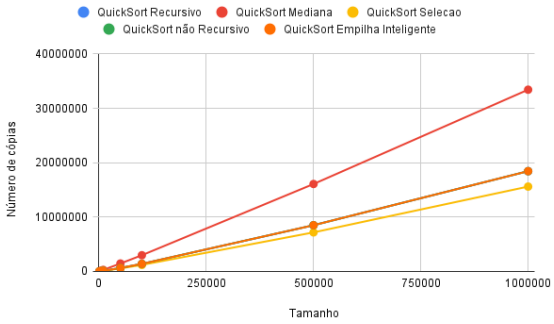
(a) Número de comparações de chaves nas versões do Quicksort

(b) Número de comparações de chaves no Quicksort X Heap Sort X Merge Sort

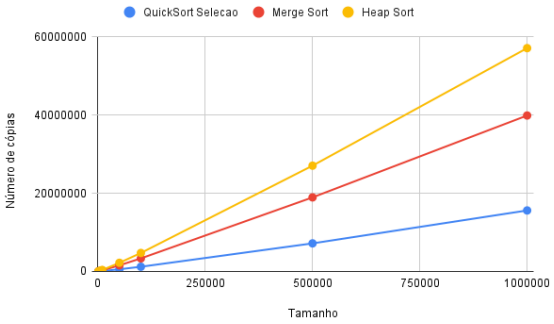
Nos gráficos acima podemos ver que, apesar de ser o mais rápido, o Quicksort Seleção é o algoritmo que mais utiliza comparações de chaves, seguido do Quicksort Mediana. As ordenações Quicksort Recursivo, Quicksort não Recursivo e Quicksort Empilha Inteligente têm exatamente os mesmos resultados, pois as úncias comparações realizadas nesses algoritmos são na chamada da função `Particao()`.

Por isso, foi escolhido o Quicksort Recursivo para ser comparado com o Merge Sort e Heap Sort. Como observado, o Merge Sort contém o menor número de comparações de chave, pois ele realiza essa operação somente no momento de juntar os subvetores particionados.

Cópias



(a) Número de cópias de registro nas versões do Quicksort

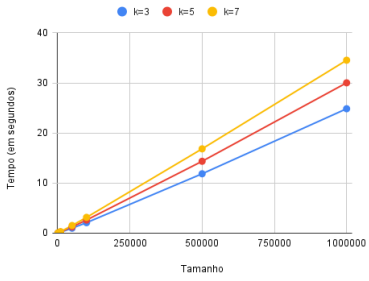


(b) Número de cópias de registro no Quicksort X Heap Sort X Merge Sort

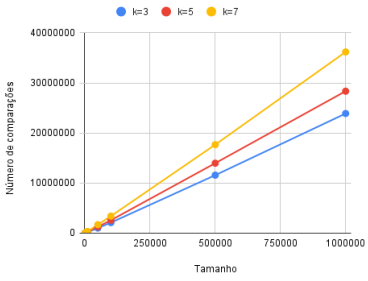
Na métrica de número de cópias, o Quicksort Seleção novamente é o mais eficaz, pois a ordenação Selection Sort realiza menos cópias de registros que as partições recursivas. Essa característica faz com que ele também seja mais eficiente que o Merge Sort e o Heap Sort nesse quesito.

QuickSort Mediana

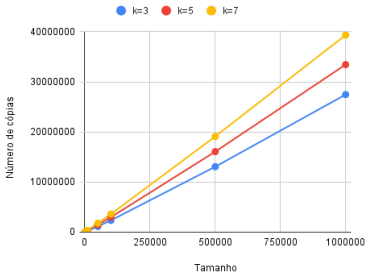
Para as estatísticas do Quicksort Mediana vistas anteriormente, foi feita a média dos 3 conjuntos de testes para $k = 3, 5, 7$. Nos gráficos abaixo podemos ver os impactos dessa variável na ordenação:



(a) Tempo total em segundos da execução



(b) Número de comparações de chave

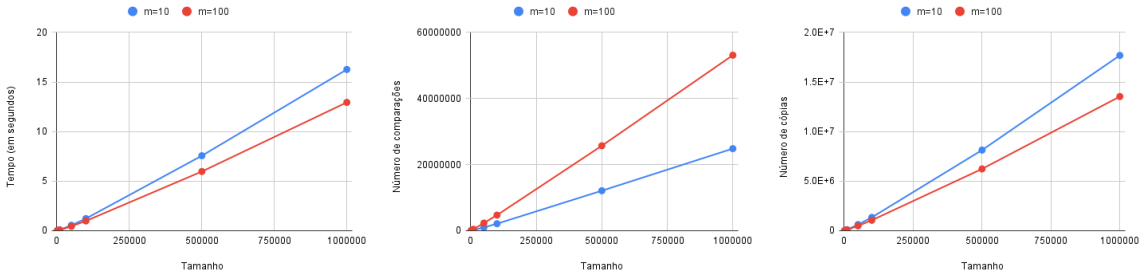


(c) Número de cópias de registro

Podemos perceber que, quanto maior o vetor de medianas, maior o tempo, número de comparações e cópias de registro realizadas. A causa disso provavelmente se dá por causa do Selection Sort utilizado para odenar o vetor auxiliar e obter sua mediana.

QuickSort Seleção

Igualmente, nas estatísticas do Quicksort Seleção foi feita a média dos dois conjuntos de testes. Os testes individuais podem ser vistos abaixo:



(a) Tempo total em segundos (b) Número de comparações (c) Número de cópias de registro

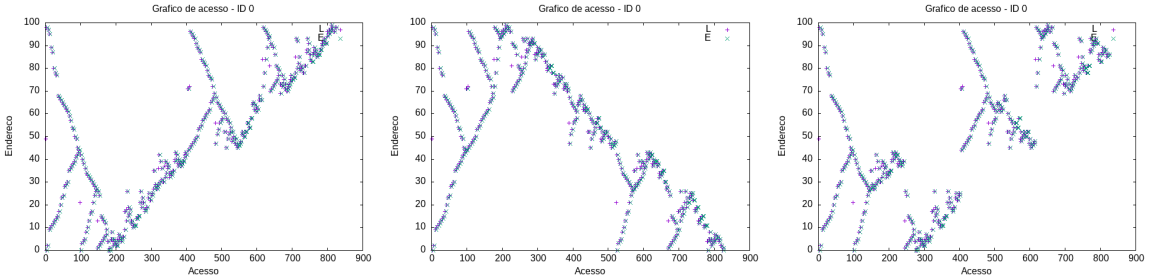
Podemos perceber que, ao aumentar o tamanho do vetor ordenado pelo Selection Sort, o tempo e número de cópias de registro diminui. Isso ocorre pois, para subvetores menores, essa ordenação é mais eficiente que o QuickSort pois um Quicksort regular realizaria aproximadamente $nlogn$ chamadas de função para os subvetores menores - ou seja, 33 chamadas para os subvetores de tamanho 10 e 664 chamadas para os subvetores de tamanho 100.

O número de comparações aumenta, pois o Selection Sort utiliza $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ comparações para ordenar um vetor de tamanho n, totalizando 55 comparações para cada subvetor de tamanho 10, e 5050 comparações para os subvetores de tamanho 100.

5.3 Analisamem

Para gerar os gráficos de acesso à memória e histogramas de pilha, foram utilizadas as funções e macros das bibliotecas `memlog` e `analisamem`. Foi feito um teste com um vetor de tamanho 100 - com exceção do Merge Sort, que rodou somente com vetores de tamanho menor ou igual a 40 -, pois o objetivo nesses testes não é testar a performance dos algoritmos, mas registrar os acessos à memória realizados.

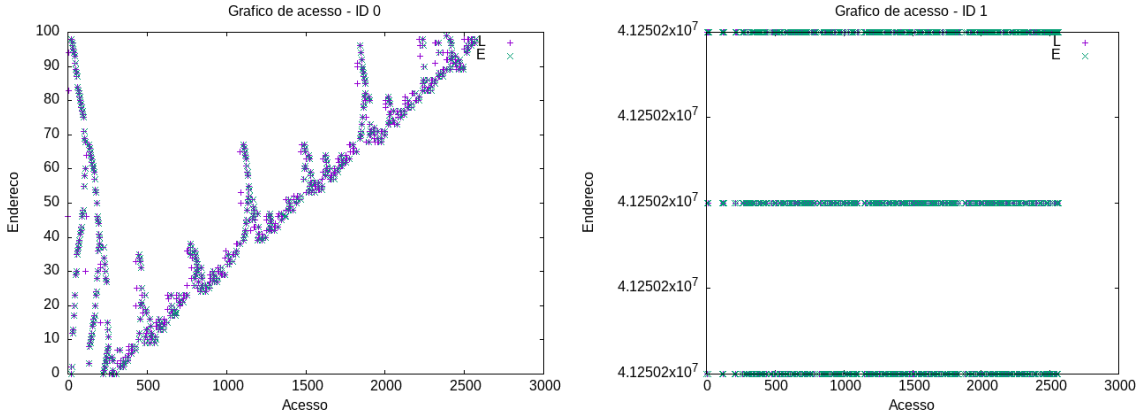
Acessos à memória



(a) Quicksort Recursivo (b) Quicksort não Recursivo (c) Quicksort Empilha Inteligente

Figura 6: Acessos à memória nas variações regulares do Quicksort

A Figura 6 mostra os acessos à memória nas três varições do Quicksort que têm uma lógica similar. Com esses gráficos, podemos perceber as diferenças no comportamento de cada uma. A principal diferença é na escolha da subpartição que será ordenada. No Quicksort Recursivo, é escolhida a partição da esquerda, no Quicksort não Recursivo, a direita e no Quicksort Empilha Inteligente apesar de visualmente não parecer haver um padrão, sabemos que o programa escolhe a menor partição para ordenar.



(a) vetor principal (b) vetor auxiliar

Figura 7: Acessos à memória no Quicksort Mediana

Na Figura 7, podemos perceber que o Quicksort Mediana realiza uma quantidade bem maior de acessos pois, como ele escolhe um elemento aleatório do vetor, a função `Particao()` realiza mais operações para dividi-lo. O gráfico da direita mostra somente o acesso às posições do vetor auxiliar de tamanho $k = 3$, alocado estaticamente.

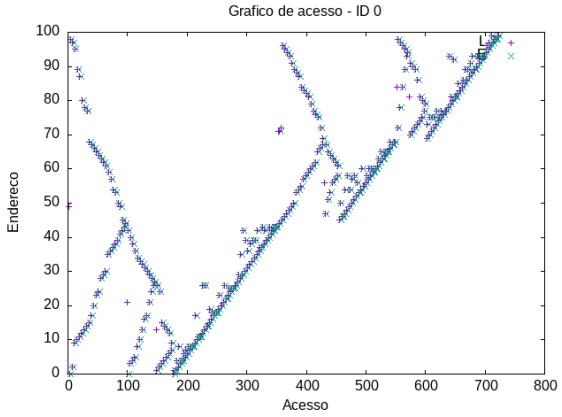
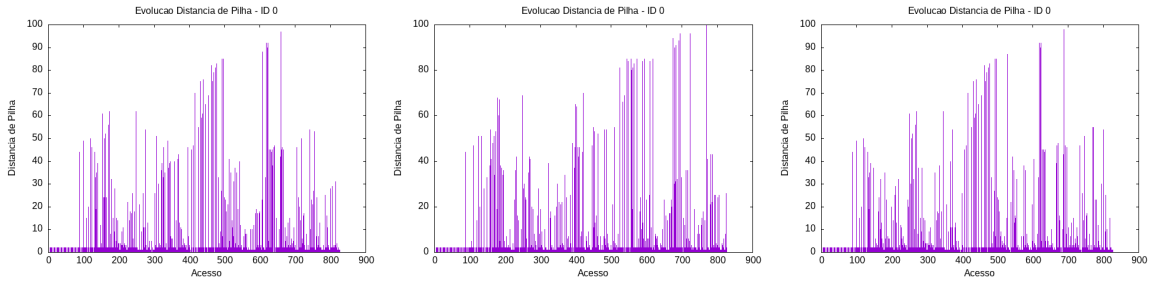


Figura 8: Acessos à memória no Quicksort Seleção

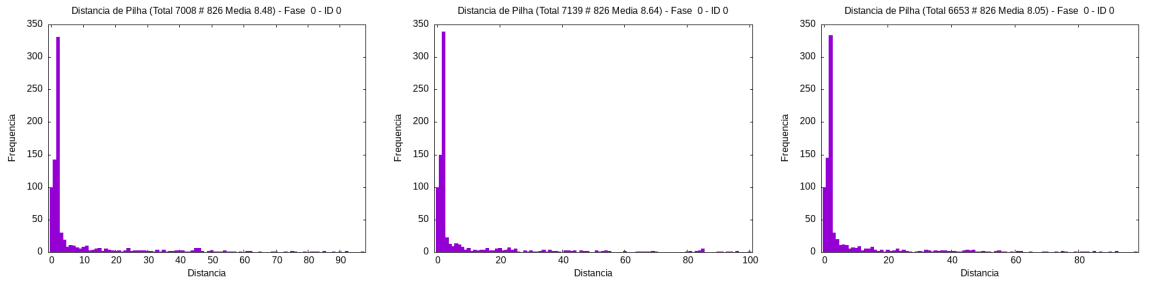
No gráfico da Figura 8, o limiar da função foi colocado como 20 e podemos ver claramente um padrão similar ao observado no Quicksort Recursivo, porém com alguns trechos de acesso sequencial que representam os subvetores de tamanho menor ou igual a 20, ordenados pelo Selection Sort.

Histogramas de pilha



(a) Quicksort Recursivo (b) Quicksort não Recursivo (c) Quicksort Empilha Inteligente

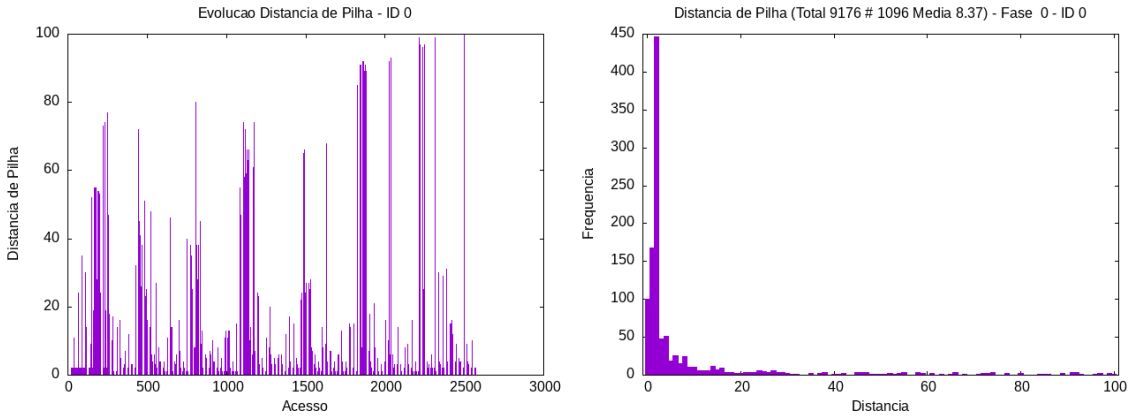
Figura 9: Evolução da distância de pilha das variações regulares do Quicksort



(a) Quicksort Recursivo (b) Quicksort não Recursivo (c) Quicksort Empilha Inteligente

Figura 10: Distância de pilha total das variações regulares do Quicksort

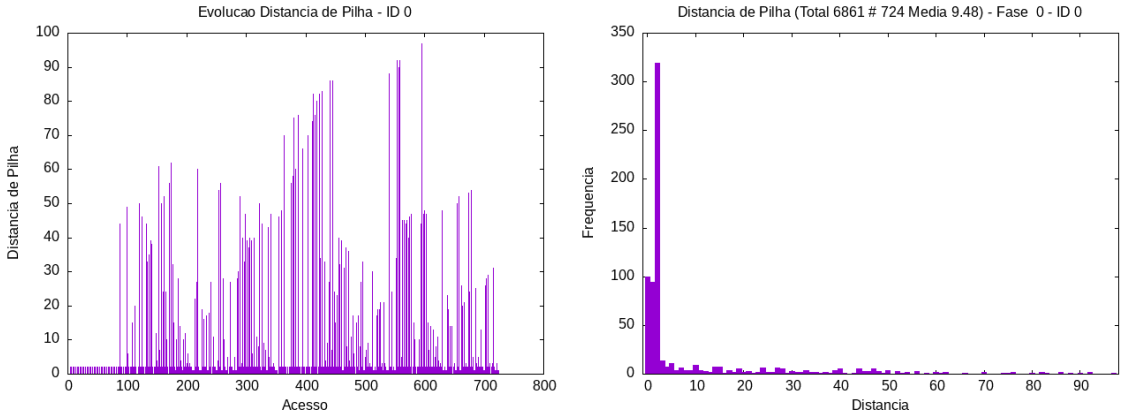
As figuras 9 e 10 demonstram a principal diferença entre essas três versões do Quicksort, que até então vêm tendo métricas bem similares quanto a tempo, número de comparações e cópias. Podemos ver nos gráficos da figura 10 que o Quicksort Empilha Inteligente tem a menor DP, com 6653 no total. Ou seja, ao escolher a menor das partições para ordenar primeiro, esse algoritmo acessa posições próximas com mais frequência, utilizando a memória de maneira mais eficiente.



(a) Evolução da distância de pilha (b) Distância de pilha total

Figura 11: Histogramas de pilha do Quicksort Mediana

Analisando o gráfico acima, percebemos que, apesar da média da distância de pilha do Quicksort Mediana ser menor que quase todos os outros métodos de ordenação, a distância de pilha total é a maior registrada e a frequência dos acessos é consideravelmente maior.



(a) Evolução da distância de pilha

(b) Distância de pilha total

Figura 12: Histogramas de pilha do Quicksort Seleção

Por fim, os gráficos do Quicksort Seleção mostram uma média de DP maior que os outros, porém a soma das distâncias de pilha é uma das menores registrada, ficando atrás apenas do Quicksort Empilha Inteligente.

6 Conclusões

Para esse trabalho, foram utilizados os conhecimentos sobre algoritmos de ordenação adiquiridos na disciplina de **Estruturas de Dados** para analisar e comparar a eficiência de algumas variações do algoritmo Quicksort e de outros dois algoritmos de ordenação, Merge Sort e Heap Sort.

A estrutura mais desafiadora de ser desenvolvida foi o Quicksort Mediana, pois essa função contém vários passos menores. Além disso, foi necessário uma maior atenção ao fato de que o vetor se mantém o mesmo durante todo o processo, apenas alterando-se as variáveis `inicio` e `fim` para cada subvetor ordenado.

A parte mais importante do trabalho, além do desenvolvimento dos algoritmos de ordenação, foi o desenvolvimento dos testes, identificando quais variáveis alterar e quais manter para gerar resultados mais claros e consisos. Essa etapa foi interessante pois, além da aplicação dos conceitos aprendidos em sala para testagem de programas, foi possível aplicar também conhecimentos de estatística e construção de gráficos que sejam claros e entendíveis pelo o leitor.

Durante a implementação do sistema, também foi utilizado o **gdb**, depurador incorporado do VSCode, para encontrar erros de execução e lógica que não poderiam ser facilmente diagnosticados ao rodar o programa normalmente. Foi utilizado também o software **Valgrind** com testes de tamanhos menores para identificar problemas de alocação e liberação incorretas de memória e para rastrear possíveis *memory leaks* que possam ter passado despercebidos.

Por fim, a formulação do relatório se mostrou novamente como uma oportunidade para utilizar os conhecimentos do sistema de preparação de documentos LaTeX, uma ferramenta muito útil durante o percurso acadêmico, na escrita de artigos e documentos científicos, pois tem um foco no conteúdo escrito sem se preocupar com a formatação final.

Referências

Adriano, M. (29 may 2009). O que são enums e como utilizá-los melhor em C++.
https://murilo.wordpress.com/2009/05/29, [accessed on Nov 3].

Akway (29 jul 2009). How do I get a specific range of numbers from rand()?
https://stackoverflow.com/questions/1202687, [accessed on Nov 7].

Heap Sort - GeeksforGeeks (16 mar 2013).
https://www.geeksforgeeks.org/heap-sort/, [accessed on Nov 10].

Corob-msft (3 aug 2021). Namespaces (C++).
https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp, [accessed on Nov 3].

Dos, C. (22 apr 2004a). Quicksort.
https://pt.wikipedia.org/wiki/Quicksort, [accessed on Nov 9].

Dos, C. (14 jun 2004b). Heapsort.
https://pt.wikipedia.org/wiki/Heapsort, [accessed on Nov 9].

Dos, C. (7 may 2005a). Merge sort.
https://pt.wikipedia.org/wiki/Merge_sort, [accessed on Nov 9].

Dos, C. (7 may 2005b). Selection sort.
https://pt.wikipedia.org/wiki/Selection_sort, [accessed on Nov 9].

Gupta, S. (24 sep 2010). How to pass optional arguments to a method in C++?
https://stackoverflow.com/questions/3784114, [accessed on Nov 3].

Jammin (30 sep 2012). How do I generate a random number between two variables that I have stored?
https://stackoverflow.com/questions/12657962, [accessed on Nov 3].

Merge Sort Algorithm - GeeksforGeeks (15 mar 2013).
https://www.geeksforgeeks.org/merge-sort/, [accessed on Nov 10].

The Open Group (2022). drand48.
https://pubs.opengroup.org/onlinepubs/7908799/xsh/drand48.html, [accessed on Nov 3].

Walter (13 jun 2012). What are inline namespaces for?
https://stackoverflow.com/questions/11016220, [accessed on Nov 3].

Instruções de Compilação e execução

Para executar o programa, siga os passos:

1. Acesse o diretório raiz `tp`;
2. Utilizando um terminal, utilize o comando `make` para compilar o código;
3. Com esse comando, um arquivo `run.out` será criado no diretório `bin`;
4. Execute usando `./bin/run.out` e os argumentos desejados em seguida:
    - `-v <número da versão>` (versão do algoritmo de ordenação escolhido)
    - `-k <número de medianas>` (número de itens que serão escolhidos para calcular o pivô, caso utilize Quicksort Mediana - versão 2)
    - `-m <número do limiar>` (tamanho limiar que decidirá a ordenação utilizada pela função, caso utilize Quicksort Seleção - versao 3)
    - `-s <número da semente>` (semente utilizada na função `rand()` da criação dos vetores)
    - `-i <nome do arquivo>` (arquivo de entrada)
    - `-o <nome do arquivo>` (arquivo de saída)
    - `-l <nome do arquivo>` (arquivo de registro do memlog - opcional)

Exemplos de comandos de entrada

* Os comandos que não inserirem o argumento `-l` terão a saída do memlog escrita no arquivo padrão `/bin/memlog.out`

- **Quicksort Recursivo:**

```
./bin/run.out -v 1 -s 10 -i entrada.txt -o saida1.txt
```

- **Quicksort Mediana:**

```
./bin/run.out -v 2 -k 7 -s 10 -i entrada.txt -o saida2.txt
```

- **Quicksort Seleção:**

```
./bin/run.out -v 3 -m 100 -s 10 -i entrada.txt -o saida3.txt
```

- **Quicksort não Recursivo:**

```
./bin/run.out -v 4 -s 10 -i diretorio/entrada.txt -o saida4.txt
```

- **Quicksort Empilha Inteligente:**

```
./bin/run.out -v 5 -s 10 -i entrada.txt -o diretorio/saida5.txt
```

- **Heap Sort:**

```
./bin/run.out -v 1 -s 10 -i entrada.txt -l bin/memLogMediana.out
```

  * Esse comando terá sua saída escrita no arquivo padrão `/bin/saida.txt` pois não inseriu a flag `-o`

- **Merge Sort:**

```
./bin/run.out -v 1 -s 10 -i entrada.txt -o saida7.txt
```

Formato do arquivo de entrada

O arquivo de entrada deve ser um arquivo de texto (em formato `.txt`, por exemplo) contendo na primeira linha um número n representando a quantidade de vetores que serão testados, seguido de n linhas contendo os tamanhos dos vetores que serão testados.

Exemplo de formato de arquivo de entrada

entrada.txt

```
6
100
500
1000
5000
50000
100000
```

Formato do arquivo de saída

O arquivo de saída é composto de uma tabela contendo, após o cabeçalho, uma linha para cada vetor enviado contendo o tamanho do vetor, tempo total da ordenação em segundos, número de comparações de chave e número de cópias de registro.

Exemplo de formato de arquivo de saída

quicksort_recursivo_teste2.out

Seed:10			Número de entradas: 6
Ordenação:	Quicksort Seleção		m = 10
Tamanho	Tempo	Comparações	Cópias
100	0.000409	827	597
500	0.005382	5488	3741
1000	0.006441	13031	8130
5000	0.034158	79523	48684
50000	0.636688	1.05108e+06	613110
100000	1.52942	2.14663e+06	1.34327e+06