Documentação Trabalho Prático 2 da Disciplina de Estrutura de Dados

Daniel Oliveira Barbosa
Matrícula: 2020006450

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

danielolibar@gmail.com

1. Introdução

Esta documentação lida com a implementação de diferentes algoritmos de ordenação de modo a testá-los com vetores de itens gerados aleatoriamente a fim de comparar o número de cópias, o número de comparações e o tempo de execução dos algoritmos, percebendo também a influência que a localidade de referência tem nos tempos de execução dos algoritmos.

2. Método

Antes de implementar os algoritmos de ordenação, foi necessário implementar os vetores a serem ordenados pelos algoritmos. Primeiro, foi criado o TAD **Item** que é composto por uma chave (que será usada para a ordenação), 10 números reais e 15 cadeias de caracteres. Depois foi necessário implementar uma forma de gerar um vetor de **Item**'s de tamanho **N** dado uma semente. Isso foi feito através da função **gerarVetorAleatorio()**.

O passo seguinte foi implementar os 5 diferentes tipos de algoritmos de ordenação. A seguir daremos uma explicação breve da implementação de cada um (os quais foram feitos seguindo os modelos implementados nas vídeo aulas e slides disponibilizadas na metaturma da disciplina de Estrutura de Dados), incluindo quaisquer estruturas ou funções auxiliares utilizadas na sua implementação.

- **Quicksort Recursivo:**
    a. **Forma de ordenação:** ordena o vetor recursivamente, trocando itens do vetor de acordo com um pivô (escolhido como sendo o elemento do meio do vetor). Depois esse mesmo vetor é partido através da função **particaoRecursivo()** e depois se aplica o mesmo processo usado anteriormente: trocar itens daquela partição de acordo com o novo pivô (elemento do meio da partição).
- **Quicksort Mediana:**
    a. **Forma de ordenação:** utiliza a mesma forma de ordenação do quicksort recursivo com uma pequena mudança: ao invés de escolher o pivô como o elemento do meio do vetor, o pivô é escolhido como a mediana de **k** elementos aleatórios desse vetor.
    b. **Estruturas/Funções Auxiliares:** foi implementado uma função que retorna a mediana de k elementos escolhidos aleatoriamente de um vetor chamado de **pivoMediana()**. Essa função cria um vetor auxiliar com esses k elementos, ordena o vetor, e depois retorna a posição do elemento do meio do vetor auxiliar

no vetor original. Como foi necessário obter a posição do elemento da mediana no vetor original, optou-se por criar uma TAD auxiliar chamado **ItemAuxiliar** que é composto por um int **chaveAuxiliar** (usada na ordenação dos k elementos escolhidos aleatoriamente) e um int **posicaoNoVetorDesordenado** (usada para verificar qual a posição de cada elemento k no vetor original). Por fim, foi usado a função **quicksortAuxiliar()** (que é um quicksort recursivo) para ordenar o vetor de k **ItemAuxiliar**'s.

   c. **Objetivo:** evitar o pior caso do quicksort recursivo, que é quando o pivô é o maior ou menor elemento de todas as partições que são feitas, gerando duas sub-partições de tamanho **N** e **N - 1**

- **Quicksort Seleção:**
   a. **Forma de ordenação:** utiliza a mesma forma de ordenação do quicksort recursivo com uma pequena mudança: quando uma partição atinge **m** ou menos elementos, ao invés de continuar ordenando essa partição através do quicksort recursivo (fazendo partições até a ordenação completa), é utilizado o algoritmo de ordenação da seleção para ordenar aquela partição.
   b. **Estruturas/Funções Auxiliares:** foi implementada uma função que faz a ordenação por seleção chamado **selecaoAuxiliar()**.
   c. **Objetivo:** diminuir o número de cópias de **Item**'s do vetor.
- **Quicksort Não Recursivo:**
   a. **Forma de ordenação:** realiza as partições da mesma forma que o quicksort recursivo. Porém, ao invés de armazenar as partições a serem processadas na call stack, elas são armazenadas em uma pilha. Assim, após toda partição, empilhamos a partição da esquerda e processamos a partição da direita.
   b. **Estruturas/Funções Auxiliares:** para armazenar os estados das partições, foi implementado uma pilha através da classe **PilhaNaoRec** que é composta por **ItemQuicksortNaoRec**'s, que é um TAD que representa uma partição e são os nodos/células da pilha. Na **PilhaNaoRec** foram implementados métodos para empilhar, desempilhar e para chegar se a pilha está vazia. O **ItemQuicksortNaoRec** é composto pelos índices (dir e esq) do range do vetor a ser processado.
   c. **Objetivo:** implementar uma forma não recursiva do quicksort
   d. **OBS.:** no código, substituímos o nome "Não Recursivo" por "Não Recursivo Não Inteligente" para reforçar a ideia de que tanto o quicksort não recursivo quanto o quicksort empilha inteligente são quicksorts não recursivos, apenas com a mudança da ordem de processamento das partições, o que será visto a seguir.
- **Quicksort Empilha Inteligente:**
   a. **OBS.:** no código, substituímos o nome "Empilha Inteligente" por "Não Recursivo Inteligente" pelas razões citadas acima.
   b. **Forma de ordenação:** utiliza a mesma forma de ordenação do quicksort não recursivo com uma pequena mudança: ao invés de sempre empilhar a partição da esquerda e processar a partição da direita, empilhamos a partição maior e processamos a partição menor.
   c. **Estruturas/Funções Auxiliares:** é utilizado a mesma pilha **PilhaNaoRec** e

nodos/células **ItemQuicksortNaoRec**'s para armazenar as partições a serem processadas.

   d. **Objetivo:** reduzir o número de vezes que empilhamos o estado atual de uma partição em uma pilha, empilhando a maior partição e processando a menor.

- **Mergesort:**
   a. **Forma de ordenação:** divide um vetor de **N** elementos em vetores unitários de forma recursiva e depois volta juntando os vetores de cada etapa recursiva (que estarão necessariamente ordenados) até voltar ao vetor de **N** elementos porém ordenado.
   b. **Estruturas/Funções Auxiliares:** N/A
   c. **Objetivo:** evitar o pior caso ($O(n^2)$) de uma ordenação.
   d. **OBS.:** a implementação foi feita com base no código do site indicado na bibliografia.
- **Heapsort:**
   a. **Forma de ordenação:** constrói o heap através da função **constroi()**, depois inverte a posição do primeiro e último item do vetor e depois restablece as regras do heap (item na posição i é sempre maior do que os itens na posição 2i e 2i + 1) e repete isso até que o vetor esteja ordenado.
   b. **Estruturas/Funções Auxiliares:** N/A
   c. **Objetivo:** garantir uma complexidade O(**n * log n**).

Depois disso, foi necessário implementar um TAD para armazenar o número de comparações e cópias dos algoritmos que foi chamado de **Desempenho**. O tempo de execução não faz parte desse TAD pois foi optado mantê-lo como um dado externo. Além disso, também foi implementado uma forma de leitura de comandos do terminal através de funções da biblioteca **getopt** e também implementados formas de ativar as funções da biblioteca do **analisamem**, as quais foram usadas na análise experimental.

O programa foi desenvolvido utilizando a linguagem C++, compilador G++ da GNU Compiler Collection e sistema operacional Windows 10 Pro 10.0.19043 Build 19043.

3. Análise de complexidade

- **Quicksort Recursivo:**
   a. **Tempo:**
      i. **Pior Caso:** quando o pivô é o maior ou menor elemento de todas as partições que são feitas, gerando duas sub-partições de tamanho **N** e **N-1**. Assim, nesse caso, o algoritmo tem complexidade de tempo O(**n^2**), pois há **n** operações por chamada recursiva e há **n** chamadas recursivas.
      ii. **Melhor Caso:** quando o pivô divide o vetor em 2 partes iguais. Assim, nesse caso, o algoritmo tem complexidade de tempo O(**n * log n**), pois há **n** operações por chamada recursiva e há **log n** chamadas recursivas.
      iii. **Caso Médio:** dado os resultados obtidos por Sedgewick e Flajolet (1996, p. 17), temos que em média a complexidade é O(**n * log n**).
   b. **Espaço:** em todos os casos terá complexidade O(**n**) que seria o tamanho do vetor

sendo ordenado.

- **Quicksort Mediana:**
    a. **Tempo:** evita o pior caso do quicksort recursivo O(**n**^{2}) onde o pivô é sempre o menor ou o maior elemento de todas as partições. Contudo, há um custo adicional para se encontrar a mediana que é dado pelo número **k** de elementos que serão usados para encontrar a mediana, que é constante para um dado valor de **k**. Porém, como o cálculo do pivô e consequentemente da mediana é feito para cada chamada recursiva, podemos dizer que o custo não é constante. De qualquer maneira, como esse custo é assintoticamente menor que **n * log n**, podemos afirmar que ele não altera a complexidade de tempo.
    b. **Espaço:** em todos os casos terá complexidade O(**n**), que seria o tamanho do vetor sendo ordenado, mais o espaço ocupado por todos os vetores de **k** elementos necessários para encontrar as medianas de cada chamada recursiva. Dado que esse espaço adicional é assintoticamente menor que O(**n**), podemos afirmar que ele não altera a complexidade de espaço.
- **Quicksort Seleção:**
    a. **Tempo:** segue a mesma complexidade de tempo do quicksort recursivo pois as partições que chegam a ser processadas por seleção são assintoticamente irrelevantes quando comparadas a todas as partições processadas normalmente pelo quicksort recursivo.
    b. **Espaço:** segue a mesma complexidade de espaço do quicksort recursivo, pois uma ordenação por seleção não necessita memória adicional.
- **Quicksort Não Recursivo:**
    a. **Tempo:** segue a mesma complexidade de tempo do quicksort recursivo pois empilhar a partição a ser processada posteriormente no quicksort não recursivo acaba por ter o mesmo custo de tempo (assintoticamente) da chamada recursiva que é feita no quicksort recursivo.
    b. **Espaço:** em todos os casos terá complexidade O(**n**), que seria o tamanho do vetor sendo ordenado, mais o espaço ocupado pela pilha. Dado que esse espaço adicional é no pior caso O(**n**) e no melhor caso O(**log n**), temos que O(**n**) + O(**n**) = O(**n**) para a complexidade de espaço.
- **Quicksort Empilha Inteligente:**
    a. **Tempo:** segue a mesma complexidade de tempo do quicksort recursivo pois empilhar a partição a ser processada posteriormente no quicksort empilha inteligente acaba por ter o mesmo custo de tempo (assintoticamente) da chamada recursiva que é feita no quicksort recursivo.
    b. **Espaço:** em todos os casos terá complexidade O(**n**), que seria o tamanho do vetor sendo ordenado, mais o espaço ocupado pela pilha. Dado que esse espaço adicional é no pior caso O(**log n**), temos que O(**n**) + O(**log n**) = O(**n**) para a complexidade de espaço.
- **Mergesort:**
    a. **Tempo:** o mergesort primeiro divide o vetor de **N** elementos em vetores unitários (operação de custo **log n)** e depois volta juntando iterando por cada elemento do vetor (operação de custo **n**). Logo a complexidade de tempo é O(**n * log n**).
    c. **Espaço:** em todos os casos terá complexidade O(**n**), que seria o tamanho do

vetor sendo ordenado, mais o espaço adicional ocupado pelos vetores auxiliares, que totalizam em um tamanho **n.** Logo a complexidade de espaço é O(**n**).

- **Heapsort:**
    a. **Tempo:** a função constroi, que é chamada uma vez, possui complexidade de tempo O(**n * log n**) e a função refaz tem, que é chamada **n - 1** vezes, possui complexidade de tempo O(**log n**). Logo, O(**n * log n**) + O(**log n**) = O(**n * log n**) para a complexidade de tempo.
    d. **Espaço:** em todos os casos terá complexidade O(**n**) que seria o tamanho do vetor sendo ordenado.

4. Estratégias de robustez

As estratégias de robustez criadas para evitar erros de input de usuários pelo terminal foram implementadas através de uma combinação de ”erroAsserts” da biblioteca **msgassert**. Aqui estão as estratégias utilizadas:

- ErroAssert utilizado para verificar se o tipo de ordenação correto foi selecionado (“quicksort”, “mergesort”, “heapsort”)
- ErroAssert utilizado para verificar se a versão do quicksort é passado quando o tipo de ordenação selecionado é o quicksort.
- ErroAssert utilizado para verificar se o valor de **k** foi passado quando é selecionado o quicksort mediana (-v 2)
- ErroAssert utilizado para verificar se o valor de **m** foi passado quando é selecionado o quicksort seleção (-v 3)
- ErroAssert utilizado para verificar se a semente, arquivo de entrada e arquivo de saída foram passados (pois são obrigatórios para qualquer tipo de ordenação).
- ErroAssert utilizado para verificar se o memlog deve ser ativado e se deve ser gerado o log
- ErroAssert utilizado para verificar se os arquivos de entrada foram abertos e se o arquivo de saída foi criado com sucesso.

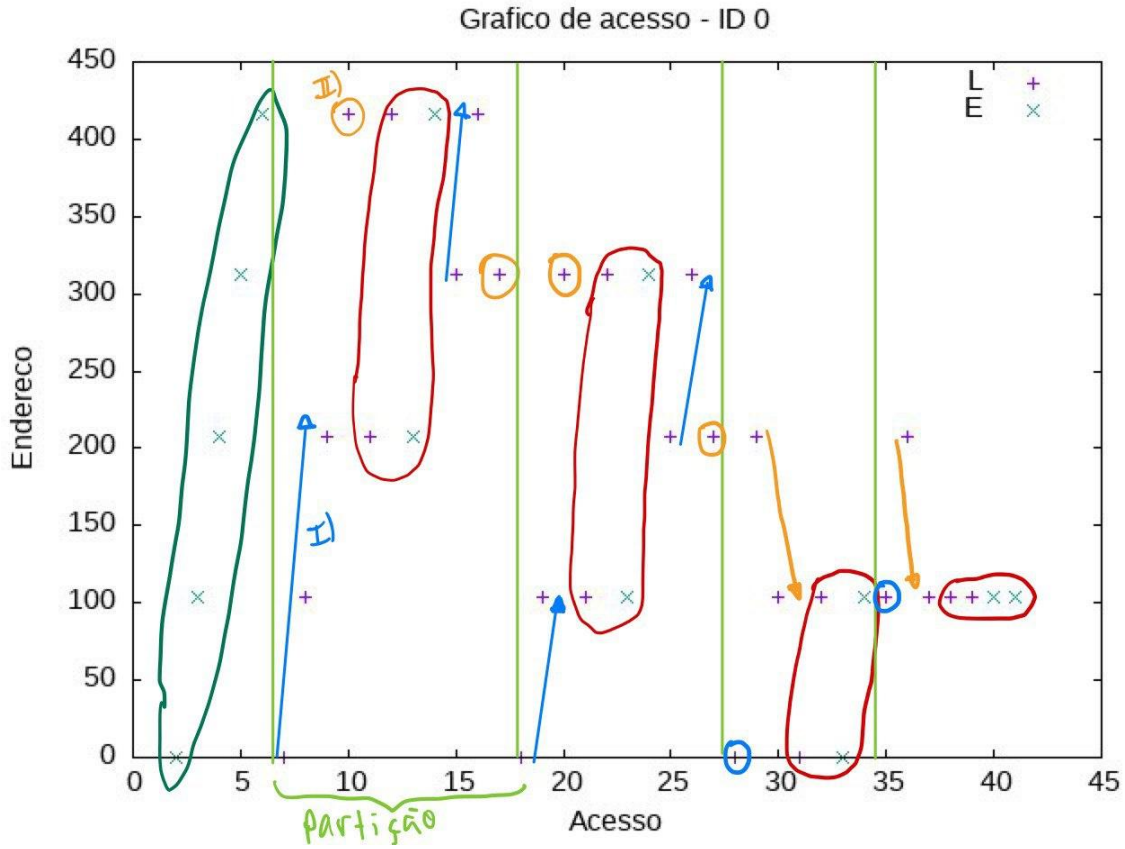
5. Análise Experimental

Antes de partirmos para as análises em si, estabeleceremos as tecnologias, os parâmetros e as pressuposições utilizadas para fazer as análises.
Em termos de tecnologias usadas, para a análise de localidade de referência foi usado o **analisamem**, com auxílio do **gnuplot** para a geração dos gráficos. Foi pressuposto que uma troca de dois itens de um vetor corresponde a 3 cópias de vetor (inclusive foi instruído que fosse feito dessa maneira no fórum pela prof. Gisele Lobo Pappa). Também vale ressaltar que, devido a particularidades no **analisamem** que já foram observados pela prof. Gisele, não foram contabilizados os acessos aos pivôs nos quicksorts para fins de análise de análise de localidade de referência. Isso também foi feito com o aval da prof. Gisele documentado no fórum do TP. Dado isso, agora faremos uma análise da localidade de referência de cada um dos algoritmos de quicksort que nos trarão bases para explicar os resultados obtidos nos testes de desempenho (tempo de execução, número de comparações e número de cópias).

- **Análise dos gráficos de acesso dos Quicksorts**

Na análise da localidade de referência, usamos a semente 10 para gerar um vetor de 5 posições a fim de facilitar a compreensão do que está acontecendo com o programa em termos de localidade de referência. Além disso, no quicksort mediana, usamos k = 3 e no quicksort seleção usamos m = 3, dado que o vetor possui apenas 5 posições.
Como veremos a seguir, todos os quicksorts seguem o mesmo padrão na maior parte dos gráficos. Portanto começaremos explicando o quicksort recursivo para facilitar a compreensão dos outros quicksorts.

a. Quicksort Recursivo



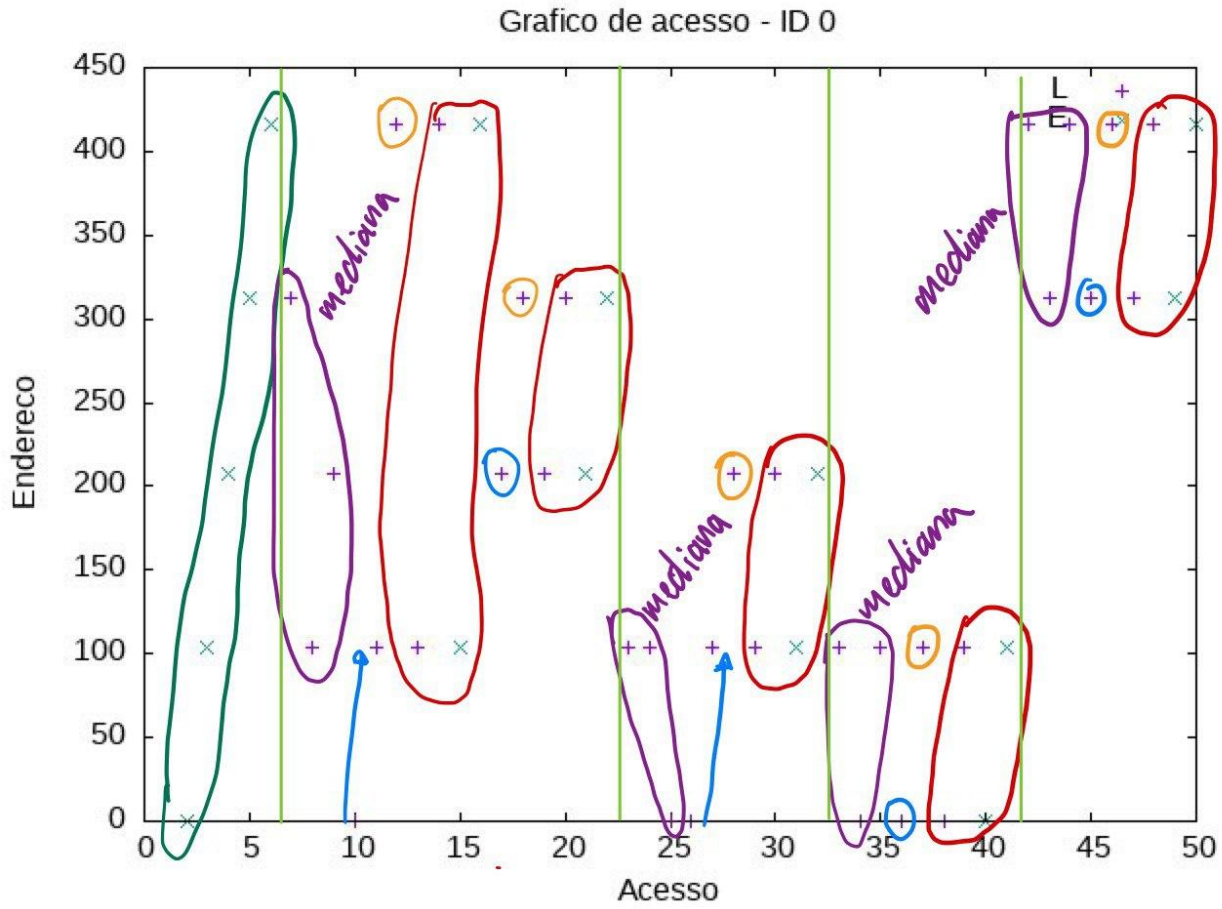
Acima temos o gráfico de acesso do quicksort recursivo. Veja que as primeiras 5 escritas (circuladas em verde escuro) correspondem à inicialização de todas as 5 posições do vetor. Logo depois, temos 3 leituras crescentes (indicadas por uma seta azul) que corresponde ao índice **i** caminhando da esquerda para direita até encontrar um item com chave maior que a chave do pivô. Depois temos uma leitura circulada em laranja mostrando que o índice **j** não caminhou da esquerda para a direita pois a chave de **j** já era menor que a chave do pivô. Dado que os índices **i** e **j** pararam de caminhar, e o índice **i** está em uma posição anterior ou igual ao **j**, é realizado a troca dos itens localizados em **i** e **j**, representado pelas 2 leituras consecutivas seguidas por 2 escritas consecutivas circuladas em vermelho. Por fim, o processo de caminhamento dos índices **i** e **j** continuam até que eles se cruzam, chegando ao fim da primeira partição indicada pela vertical em verde claro. Veja que esse mesmo processo se repete durante

toda a execução do algoritmo:

1. O índice **i** "caminha" pela seta azul (ou continua no mesmo lugar caso esteja circulado em azul),
2. Depois o índice **j** "caminha" pela seta laranja (ou continua no mesmo lugar caso esteja circulado em laranja),
3. Depois faz-se a troca dos itens da posição **i** e **j** através de 2 leituras consecutivas seguidas por 2 escritas consecutivas circuladas em vermelho,
4. E por fim, as etapas 1), 2) e 3) se repetem até que os índices **i** e **j** se cruzem, dando fim ao processamento daquela partição (delimitada pelas verticais em verde claro).

Todos os quicksorts, por terem a função de partição, compartilham um padrão de acesso semelhante, senão idêntico.

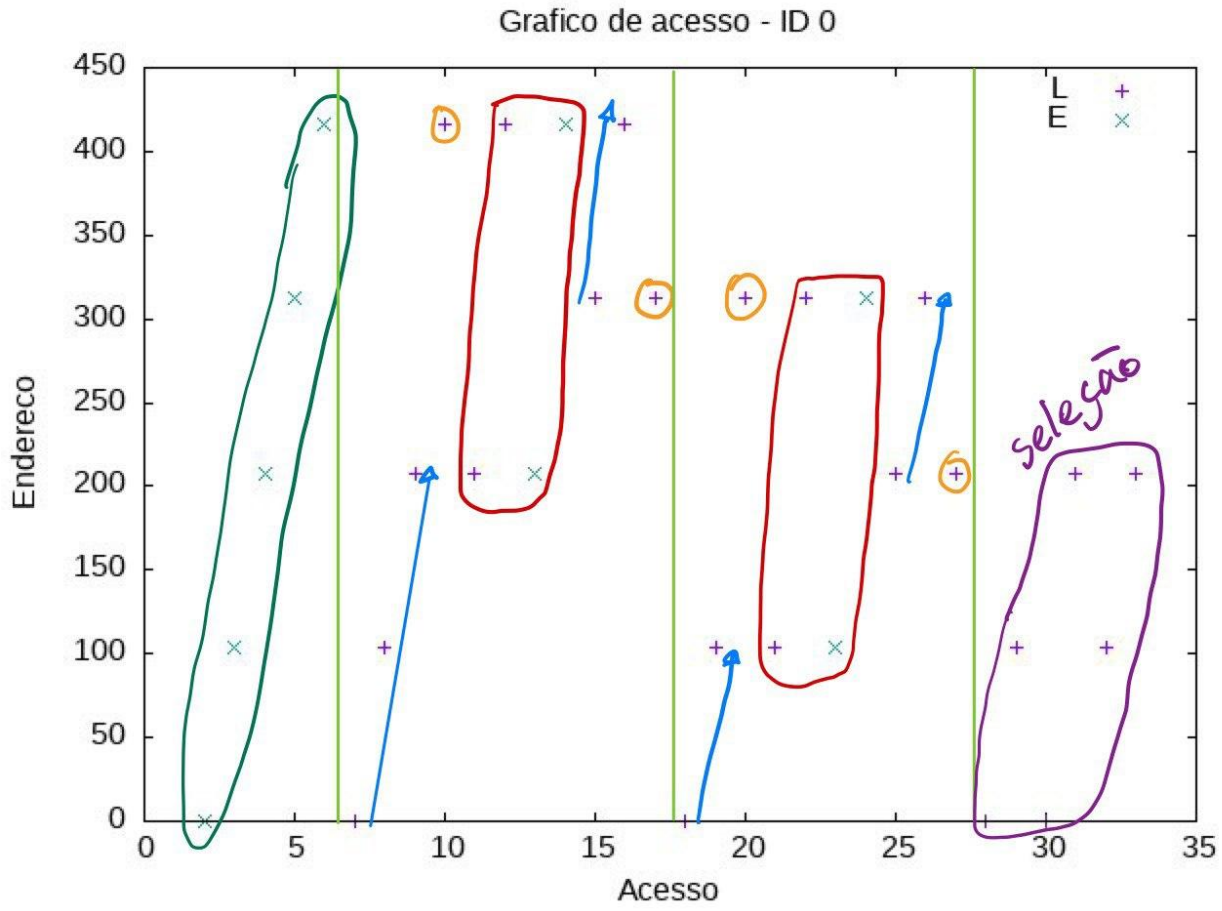
b. Quicksort Mediana (k = 3)



Acima temos o gráfico de acesso do quicksort mediana. Veja que a ordem em geral continua a mesma: **1) i** caminha (seta/círculo azul), **2) j** caminha (seta/círculo laranja), **3)** troca (círculo vermelho), **4)** os passos 1, 2 e 3 se repetem até que **i** e **j** se cruzem, chegando ao fim da partição (vertical verde claro). Porém, a diferença principal aqui é que temos a determinação da mediana através da função **pivoMediana()**, cujos acessos estão circulados em roxo. Como escolhemos k = 3, é necessário ler o vetor 3 vezes para selecionar os k elementos aleatoriamente. Essas 3 leituras extras por chamada recursiva aumentam o custo do algoritmo. Também vale salientar

que o número de trocas aumentou de 3 (no quicksort recursivo) para 5 (no quicksort mediana) e, consequentemente, o número de cópias também aumentou. Dado que o TAD **Item** ocupa muita memória, o custo de se fazer uma cópia é grande (obs.: essa observação será reutilizada em toda a análise experimental). Assim, além do aumento do custo total devido ao custo adicional de se determinar a mediana de k elementos, também há o custo adicional decorrente do aumento do número de cópias feitas. Esses dois fatores serão essenciais para compreender a disparidade do desempenho do quicksort mediana quando fizermos as análises de desempenho. Um detalhe importante é que para valores de k cada vez maiores, temos um custo adicional associado à determinação da mediana proporcionalmente maior devido ao aumento ainda maior de acessos, piorando ainda mais o desempenho.

c. Quicksort Selecao (m = 3)



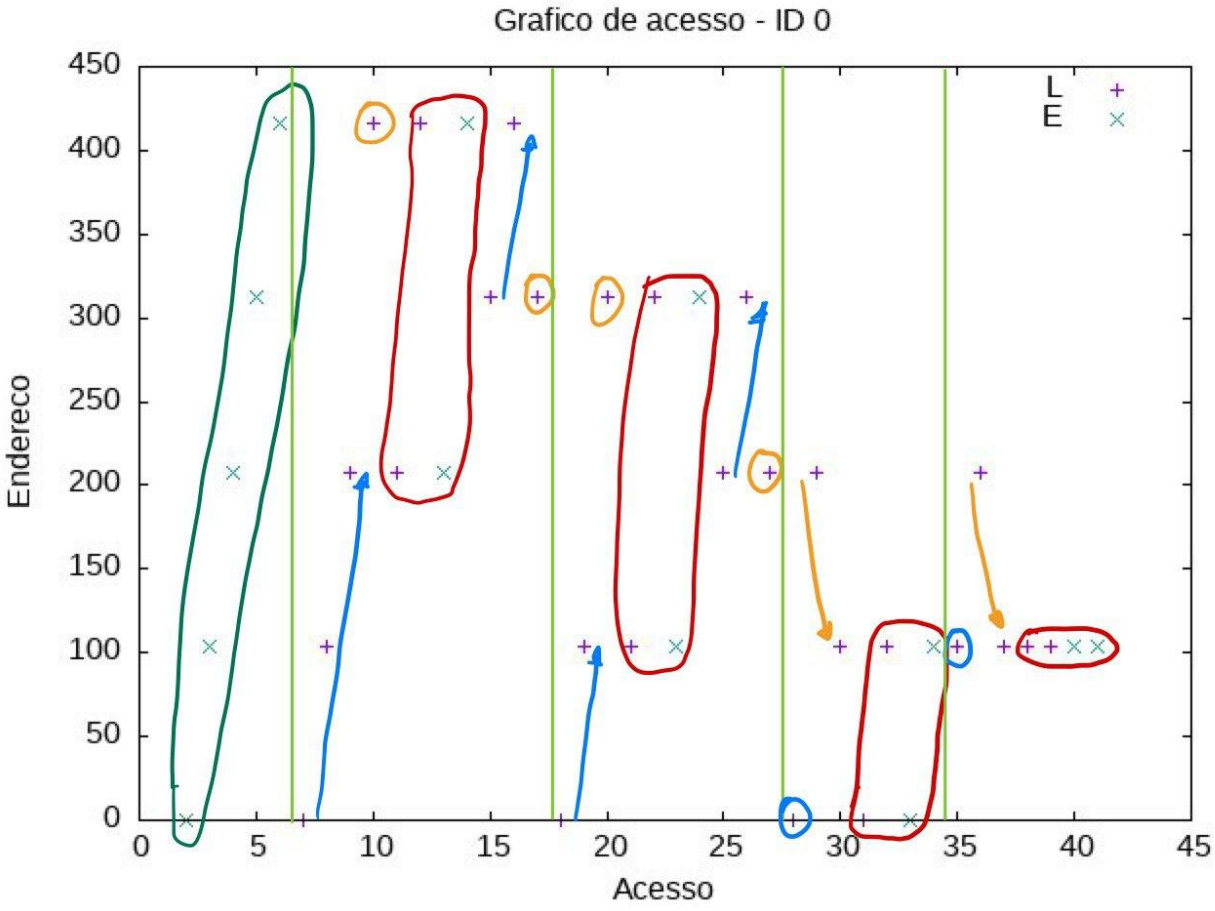
Acima temos o gráfico de acesso do quicksort seleção. Veja que a ordem em geral continua a mesma. Inclusive, ela é idêntica ao quicksort recursivo até o final da segunda partição. Porém, já na terceira partição, o tamanho da partição é igual a 3, e como determinamos que m = 3, a partir daí, a ordenação se dá por seleção (circulada em roxo). Vale explicitar que o número de trocas - e consequentemente cópias - diminui quando comparado com o quicksort recursivo. Como já havíamos determinado que cópias geram muito custo, isso traz uma redução considerável do custo total do algoritmo. Nesse caso de teste específico (vetor de 5 posições gerado a partir da semente 10 e um valor de m = 3), a função que faz a ordenação por seleção

(**selecaoAuxiliar()**) é chamada apenas uma vez (circulado em roxo). Contudo, em casos onde o número de elementos do vetor é maior, vemos que o número de acessos aumenta significativamente dependendo para valores de m cada vez maiores. Assim, apesar de ter uma redução do número de cópias, há um aumento desproporcional de número de acessos totais devido ao aumento no número de comparações da seleção, gerando um aumento de custo ao invés de uma diminuição. Dessa maneira, há um trade-off entre número de cópias e número de comparações que é diretamente ligada à escolha do valor de m (fato que será explicado melhor nos gráficos de distância de pilha)

d. Quicksort Não Recursivo

O gráfico de acesso do quicksort não recursivo é idêntico em todos os sentidos ao quicksort recursivo pois a forma de acessar o vetor não muda. Dado isso, vamos omitir esse gráfico.

e. Quicksort Empilha Inteligente

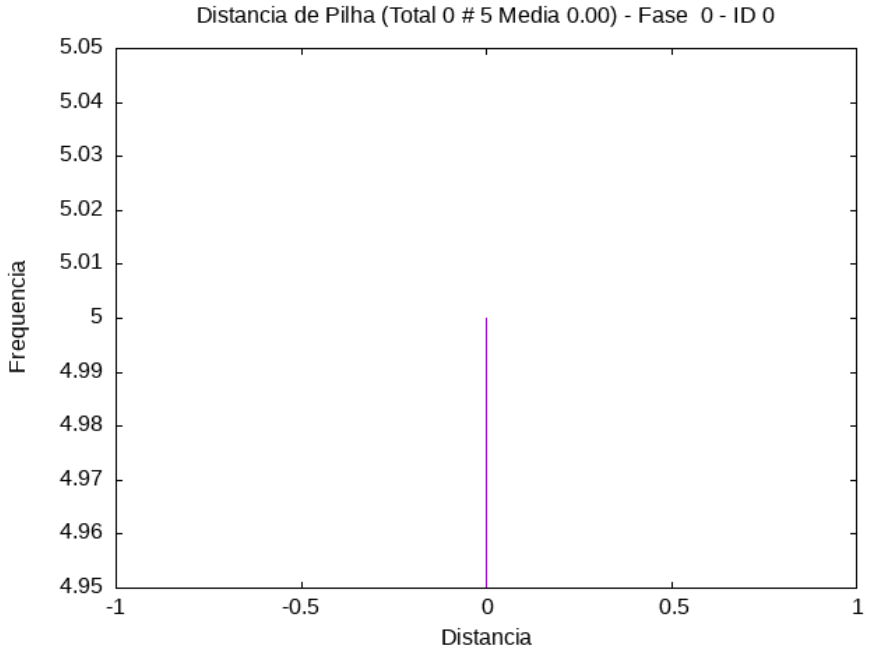


Acima temos o gráfico de acesso do quicksort empilha inteligente. Veja que nesse caso de teste específico (vetor de 5 posições gerado a partir da semente 10), o gráfico é idêntico ao gráfico de acesso do quicksort recursivo e não recursivo. Porém, esse fato ocorreu pois, nesse caso de teste específico, a partição da esquerda (que é sempre empilhada no quicksort recursivo e não recursivo) era sempre a maior partição (que é sempre empilhada no quicksort empilha

inteligente). Entretanto, se fosse usado um vetor em que, em pelo menos uma das partições, isso não ocorresse, o gráfico seria diferente. A diferença seria que os acessos do gráfico do quicksort empilha inteligente ocorreriam em posições de memória mais próximas que os acessos dos gráficos do quicksort recursivo e não recursivo. Esse fenômeno será explicado melhor quando os gráficos de distância de pilha forem analisados. Mas de qualquer forma, como todos os outros quicksorts recursivos, a ordem geral continuará a mesma: **1) i** caminha (seta/círculo azul), **2) j** caminha (seta/círculo laranja), **3)** troca (círculo vermelho), **4)** os passos 1, 2 e 3 se repetem até que **i** e **j** se cruzem, chegando ao fim da partição (vertical verde claro).

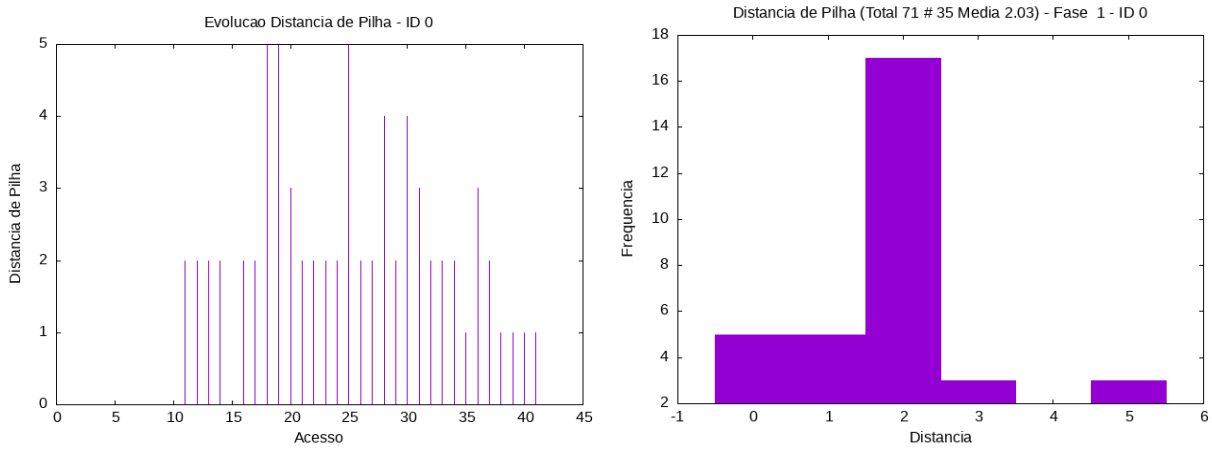
- **Análise dos gráficos de distância de pilha fase 0 dos Quicksorts**

Veja que, para todos os algoritmos (incluindo o mergesort e o heapsort), o gráfico de distância de pilha da fase 0 é a fase onde o vetor aleatório é gerado pela função **gerarVetorAleatorio()**. Como essa função apenas inicializa cada posição do vetor consecutivamente, há sempre uma distância de pilha 0 pois acessamos cada posição do vetor apenas uma vez. Logo, como fizemos a análise de localidade de referência sempre com um vetor de 5 posições, todos os gráficos de distância de pilha serão da seguinte forma:



- **Análise dos gráficos de distância de pilha fase 1 e evolução de distância de pilha dos Quicksorts**

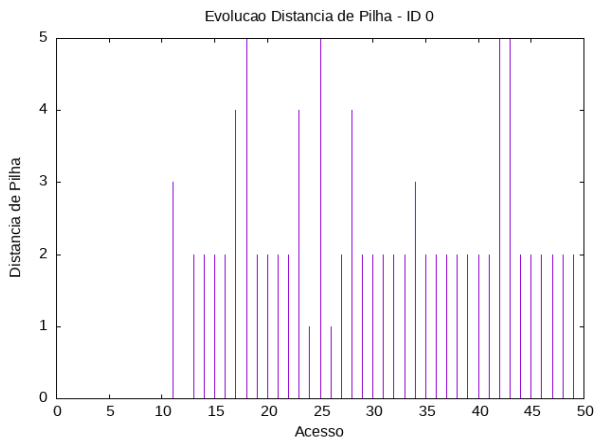
    a. **Quicksort Recursivo, Não Recursivo e Empilha Inteligente**

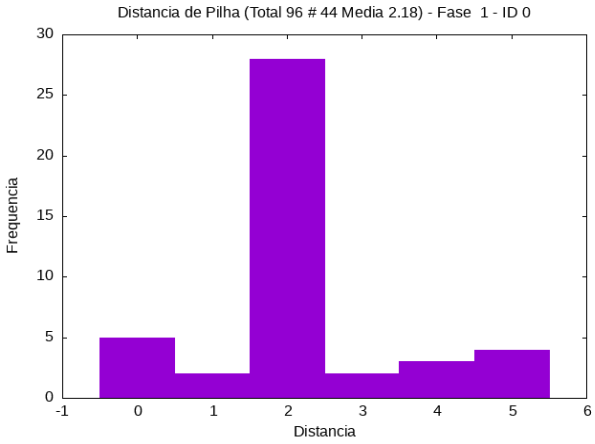


Gráficos do Quicksort Recursivo, Quicksort Não Recursivo e Quicksort Empilha Inteligente

Não por coincidência, os gráficos de distância e pilha e evolução de pilha do quicksort recursivo, quicksort não recursivo e do quicksort empilha inteligente são idênticos. É natural que os gráficos do quicksort recursivo e o quicksort não recursivo sejam idênticos, pois o padrão de acesso ao vetor é o mesmo para ambos (fato que já foi observado nos gráficos de acesso anteriormente). Já no gráfico do quicksort empilha inteligente, os gráficos são idênticos aos gráficos do quicksort recursivo e não recursivo pois como foi dito anteriormente, nesse caso de teste específico (vetor de 5 posições gerado a partir da semente 10), a partição da esquerda coincidia ser sempre a maior partição. Entretanto, se fosse usado um vetor em que, em pelo menos uma das partições, isso não ocorresse, veremos que o gráfico de distância de pilha e de evolução de pilha do quicksort empilha inteligente teriam uma frequência maior de acessos com distância de pilha maior (como 3, 4 e 5) quando comparado com os gráficos do quicksort recursivo e não recursivo. Esse fenômeno ocorre pois ao processar partições menores primeiro, o quicksort empilha inteligente acessa posições de memória mais próximas quando comparado ao processamento de partições maiores. Logo, devido ao aproveitamento do princípio da localidade por acessar posições de memória mais próximas, o quicksort terá desempenho melhor, como veremos mais adiante.

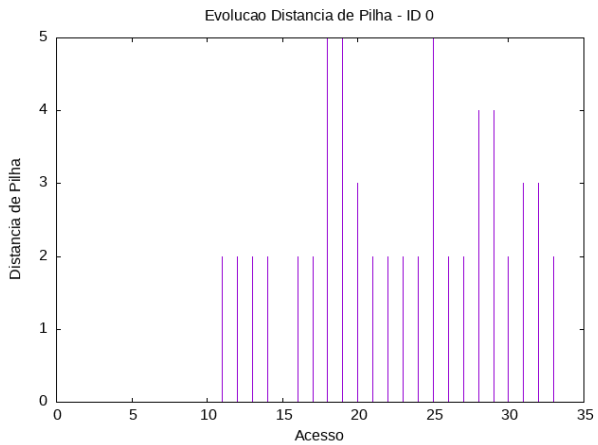
b. Quicksort Mediana (m = 3)

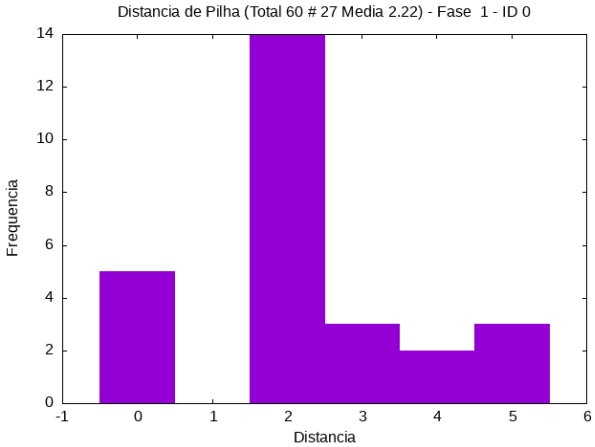




Veja que nos gráficos do quicksort mediana, o número de acessos em geral aumenta devido à necessidade de acesso ao vetor k vezes a cada chamada recursiva para determinar o pivô como a mediana desses k elementos (fato já observado nos gráficos de acesso). Como a posição desses k elementos é escolhida aleatoriamente, vemos um aumento no número de acessos de distância de pilha 4 e 5, desfavorecendo o princípio da localidade. Logo, apesar de que escolher o pivô como a mediana de k elementos do vetor garante que não teremos o pior caso do quicksort recursivo O($\mathbf{n^2}$), o número de acessos aumenta, a distância de pilha entre esses acessos aumenta, desfavorecendo o princípio da localidade e o número de cópias (operação muito custosa) aumenta significativamente. Portanto, em termos práticos, esse algoritmo se torna muito custoso, o que se torna evidente nos resultados de desempenho que veremos mais à frente.

c. Quicksort Seleção(k = 3)





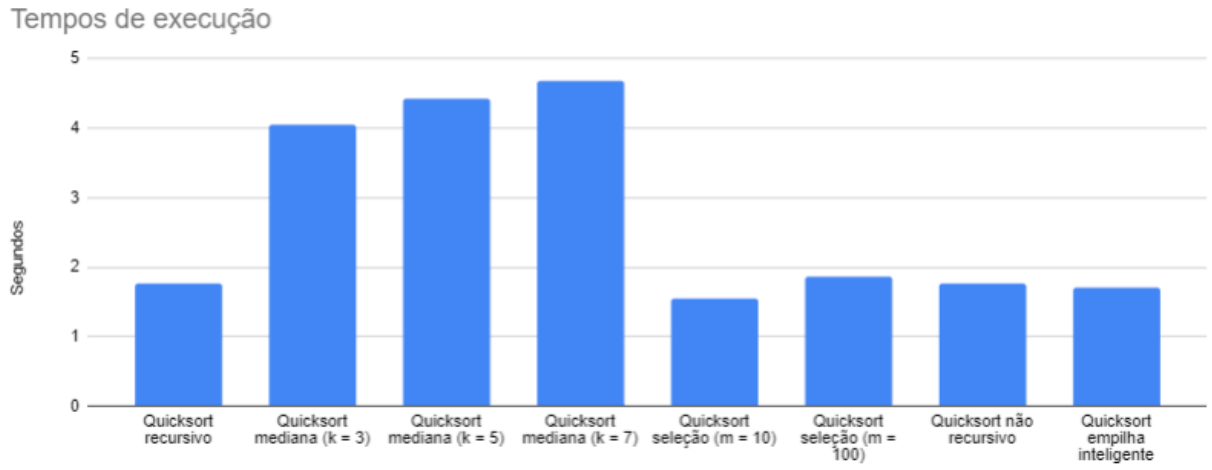
Veja que no quicksort seleção, os gráficos mantiveram o mesmo número de acessos de distância de pilha 3, 4 e 5 quando comparados com o quicksort recursivo e não recursivo, conservando o princípio da localidade. Como já determinamos antes (na análise dos gráficos de acesso), para vetores maiores, o número de acessos aumenta significativamente para valores de m cada vez maiores, chegando ao ponto de neutralizar os benefícios da redução do número de cópias. Isso se dá pois, além de aumentar o número de acessos por conta do algoritmo da seleção, há

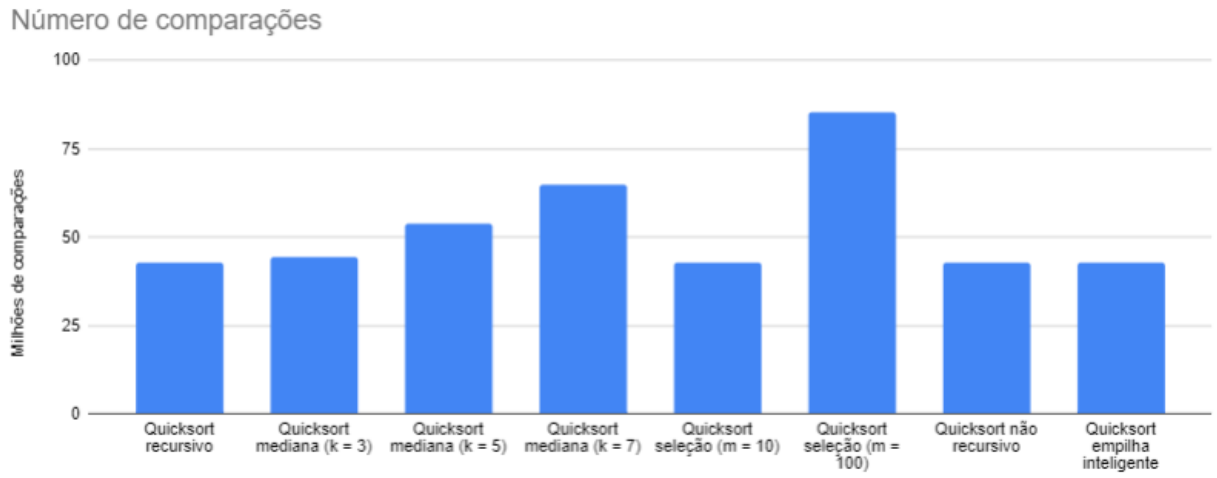
também um aumento número de acessos de distância de pilha maiores pois há menos acessos de distância de pilha maior na ordenação por seleção de um vetor de m = 10 posições do que em um vetor de m = 100 posições, por exemplo. Ou seja, para valores de m cada vez maiores, há um número de acessos cada vez maiores com distância de pilhas cada vez maiores, desfavorecendo o princípio da localidade. Logo, há um trade-off entre aumentar o valor de m, que garante menos cópias mas aumenta o número de acessos e aumenta os acessos de distância de pilha maiores, ou diminuir o valor de m, que garante menos acessos e diminui os acessos de distância de pilha maiores mas aumenta o número de cópias. Isso será demonstrado na análise de desempenho a seguir.

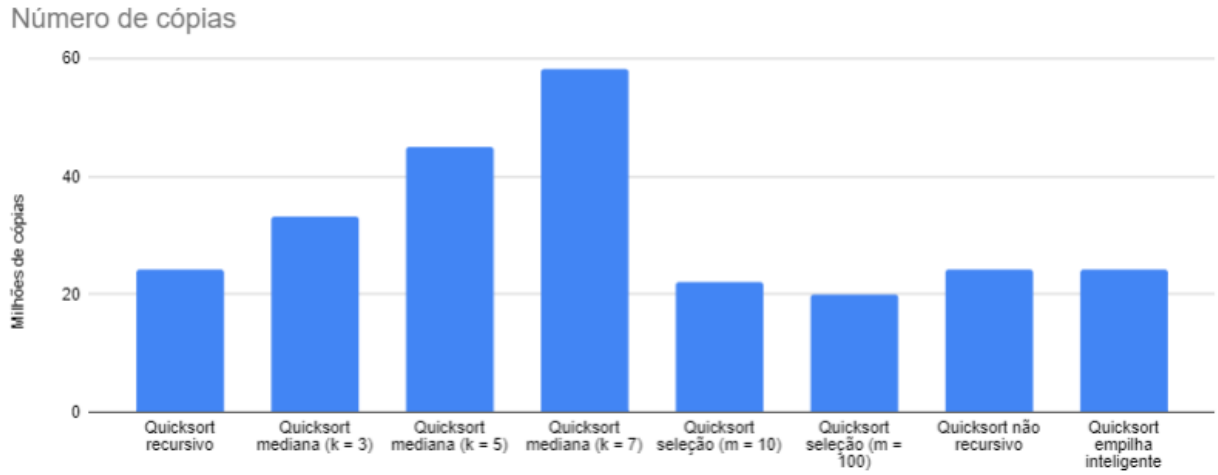
- **Parte 1: Impacto de variações do Quicksort**

Aqui foram feitos testes para as sementes 10, 58, 649, 396284, 98278992 e foram tirados a média entre eles. Também foram analisados os quicksorts mediana com k = 3, 5, e 7 e quicksort seleção com m = 10 e 100.

Algoritmo	Tempo de execução (segundos)	Número de comparações	Número de cópias
Quicksort recursivo	1,7611	42864365	24184437
Quicksort mediana (k = 3)	4,05511	44294729	33289365
Quicksort mediana (k = 5)	4,43493	53720591	45072067
Quicksort mediana (k = 7)	4,68662	64784339	58330336
Quicksort seleção (m = 10)	1,53943	42679802	22055305
Quicksort seleção (m = 100)	1,86096	85384453	19986051
Quicksort não recursivo	1,76492	42864365	24184437
Quicksort empilha inteligente	1,69552	42864365	24184437







a. **Quicksort Mediana:** veja que, como previsto na análise de localidade de referência, o quicksort mediana performou pior que todos os outros algoritmos. Isso se deu pelo aumento no número de acessos totais, incluindo cópias e comparações. Como também foi previsto, para valores cada vez maiores de k, há um desempenho cada vez pior devido ao aumento do número total de acessos. Portanto, o quicksort mediana realmente deve ser reservado para o caso em que o pior caso não pode ser tolerado e deve ser escolhido sempre k = 3.
b. **Quicksort Seleção:** veja que esse algoritmo se revelou ser o mais eficiente em todas as métricas de desempenho, como previsto pela análise de localidade de referência. Isso se dá pela redução do número de cópias necessárias, cujo custo é alto. Como foi também observado anteriormente, há um trade-off entre escolher um m maior (garantindo menos cópias porém mais acessos e acessos distâncias de pilhas maiores) ou um m menor (garantindo mais cópias porém menos acessos e acessos distâncias de pilhas menos). No caso dos nossos testes, a escolha do m = 10 se sobressaiu, mas isso dependerá de variáveis como as sementes utilizadas e também especificações da máquina. Contudo a conclusão

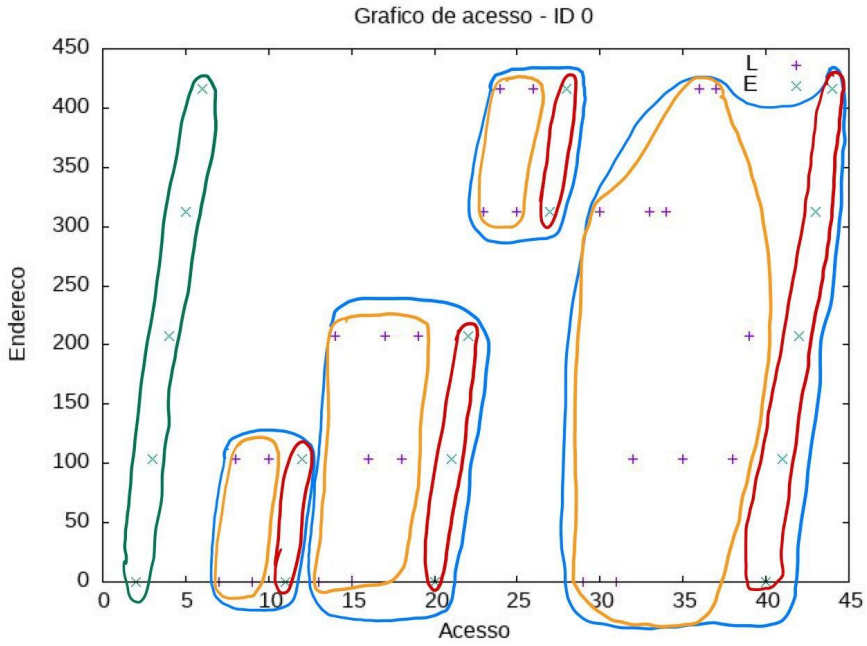
principal é que há um trade-off na escolha de m.

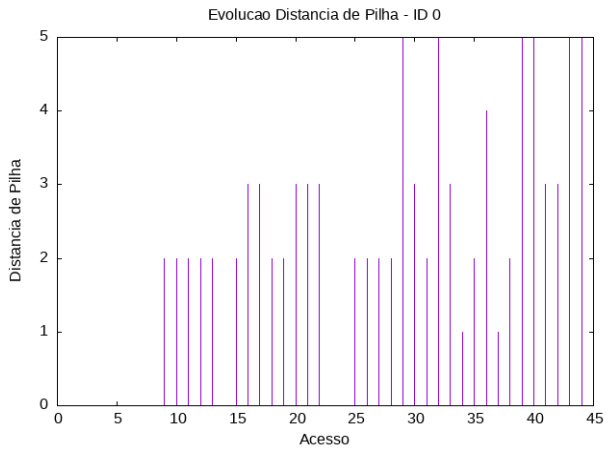
c. **Quicksort Não Recursivo:** veja que a diferença dos tempos de execução do quicksort recursivo e não recursivo foram irrelevantes e foram dados por fatores externos além da implementação pois em diferentes testes um podia sair mais rápido que o outro e vice-versa. Além disso, o número de comparações e cópias são idênticas ao quicksort recursivo. Esses fatos citados acima se dá pois armazenar os índices da partição a ser processada posteriormente no call stack (no caso do quicksort recursivo) ou na pilha **PilhaNaoRec** implementada (no caso do quicksort não recursivo) é virtualmente equivalente quando levamos em consideração as métricas de desempenho em questão.
d. **Quicksort Não Recursivo:** veja que no caso do quicksort empilha inteligente, como visto na análise de localidade de referência, estamos otimizando a forma que armazenamos os índices da partição a ser processada posteriormente. Isso se traduz no melhor desempenho de tempo de execução do quicksort empilha inteligente quando comparado ao quicksort não recursivo, apesar de ter número de comparações e cópias idênticas

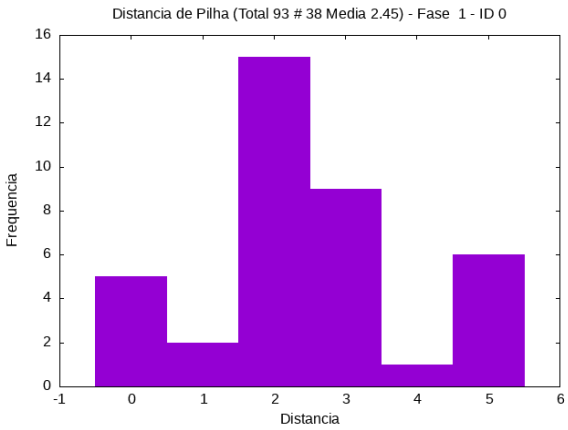
Portanto, podemos concluir que o quicksort seleção com m = 10 foi o mais eficiente de todos, dado que foi efetivo em reduzir o número de cópias, que são custosas devido à grande memória ocupada pelos **Item**'s do vetor. Também foi mais eficiente que o quicksort seleção com m = 100 pois o número de comparações a mais necessárias para realizar o algoritmo de seleção para partições com 100 ou menos elementos foi mais custoso do que fazer a cópia dos itens adicionais que seriam feitas caso tivéssemos escolhido m = 10. Assim, usaremos o quicksort seleção m = 10 para a segunda parte do trabalho.

- **Análise dos gráficos de localidade de referência do Heapsort e do Mergesort**

a. **Mergesort**

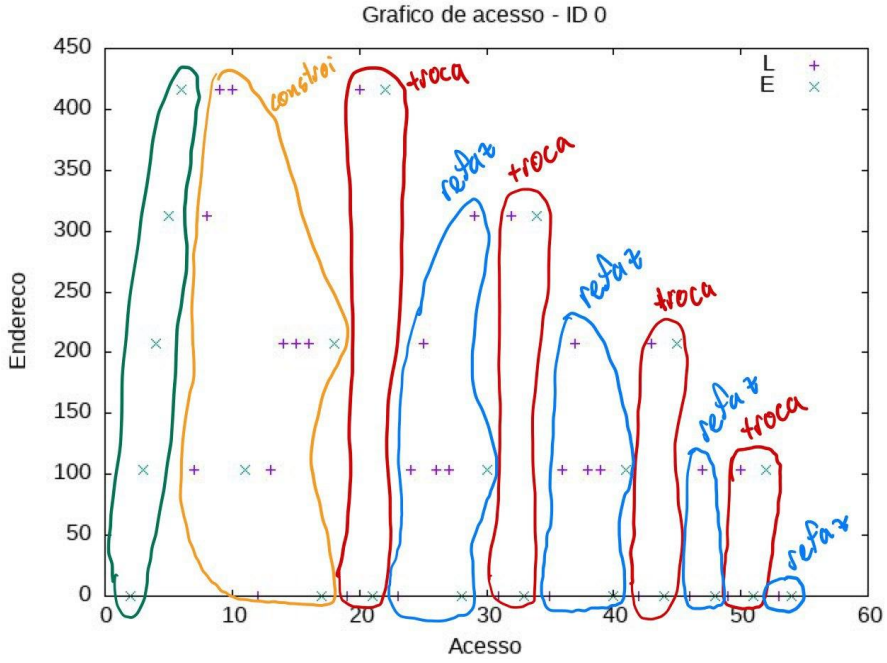


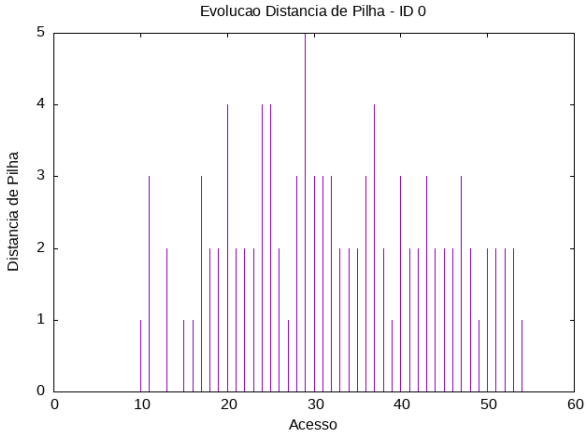


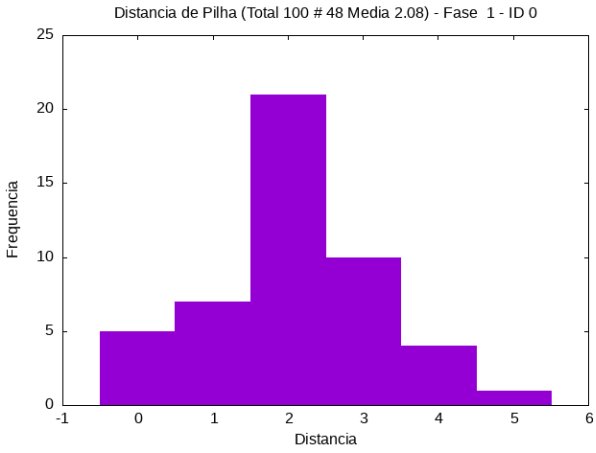


Veja que os primeiros 5 acessos para escrita (circulados em verde) correspondem à inicialização do vetor. Depois, cada conjunto de acessos circulados em azul correspondem à chamada da função **merge()**. A função **merge()** é composta por duas partes: primeiro o vetor auxiliar é preenchido de maneira ordenada (representado pelas leituras circuladas em laranja) e depois o vetor é preenchido com os valores ordenados armazenados no vetor auxiliar (representado pelas escritas circuladas em vermelho). Veja que na última chamada do **merge()**, há 5 escritas (circuladas em vermelho) que correspondem ao preenchimento de todo o vetor com os valores ordenados armazenados no vetor auxiliar, chegando ao fim da ordenação.
Veja que, em questões de distância de pilha, o número de acessos com distância de pilha maior aumentou significativamente quando comparado com o quicksort seleção, desfavorecendo o princípio da localidade de referência. Isso, aliado ao fato de que faz mais cópias, traduz diretamente em um aumento nos tempos de execução.

b. Heapsort

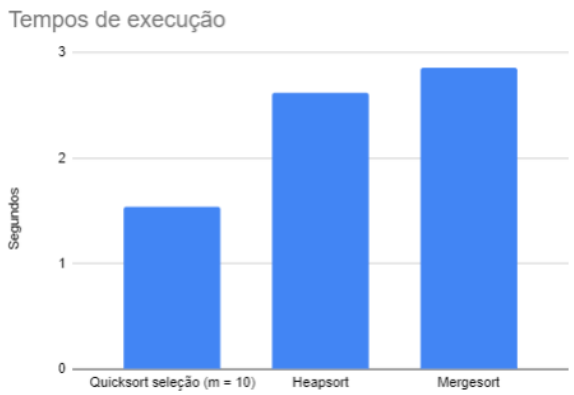


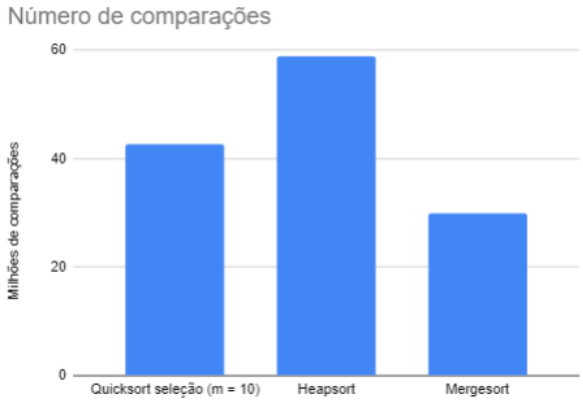


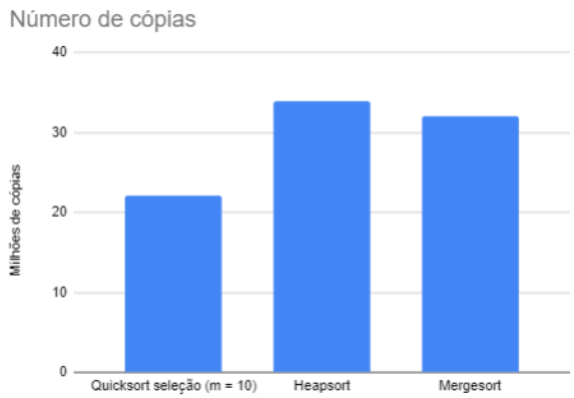


Veja que os primeiros 5 acessos para escrita (circulados em verde) correspondem à inicialização do vetor. Depois, os acessos circulados em laranja correspondem à construção do heap pela função **constroi()**. Após isso, temos uma intercalação entre trocas de dois itens do vetor (circulado em vermelho) e chamadas à função **refaz()** (circulados em azul) para restabelecer o heap (item na posição i é sempre maior do que os itens na posição 2i e 2i + 1).
Veja que, em questões de distância de pilha, o número de acessos aumentou significativamente quando comparados ao quicksort seleção e mergesort. O quicksort seleção faz menos cópias e menos acessos que o heapsort, tendo portanto um desempenho claramente melhor quando levados em consideração os tempos de execução. Já o mergesort possui menos acessos porém os acessos são feitos com uma distância de pilha maior, desfavorecendo o princípio da localidade e gerando um desempenho em termos de tempo de execução pior que o heapsort, que possui mais acessos.

- **Parte 2: Quicksort Seleção (m = 10) X Mergesort X Heapsort**







Algoritmo	Tempo de execução (segundos)	Número de comparações	Número de cópias
Quicksort seleção (m = 10)	1,53943	42679802	22055305
Heapsort	2,6222	58980158	33907429
Mergesort	2,85821	29950169	32085928

Como previsto pelas análises de localidade de referência, o heapsort, por ser mais eficiente quanto ao princípio da localidade, teve desempenho em termos de tempo de execução melhores que o mergesort, mesmo que o mergesort tivesse menos acessos. Contudo, o quicksort seleção m = 10 foi o algoritmo com melhor desempenho devido ao menor número de acessos em geral (incluindo cópias e comparações).

6. Conclusão

O trabalho prático teve como objetivo demonstrar como o número de comparações, cópias e também a localidade de referência afetam os algoritmos de ordenação. Dessa análise foi possível observar que a análise de complexidade permite ter apenas uma ideia do desempenho do algoritmo. Porém, questões como localidade de referência e memória ocupada pelos itens do vetor a ser ordenado podem ter grande impacto sobre o desempenho de um algoritmo de ordenação.
Com todo esse processo, pude revisar e entender mais a fundo como funcionam diferentes algoritmos de ordenação e em quais situações elas melhor se aplicam. Além disso, pude dar continuidade no aprendizado de C++ em geral. Em suma, conclui-se que o trabalho teve sucesso no seu objetivo de ensino sobre algoritmos de ordenação.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011
- Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third

Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012
- Vídeo Aulas e Slides disponibilizadas na Metaturma da disciplina de Estrutura de Dados
- Referência para Mergesort: https://joaoarthurbm.github.io/eda/posts/merge-sort/

8. Instruções de compilação e execução

- Vá para o diretório raiz do projeto (/TP)
- Digite na linha de comando (para limpar arquivos desnecessários do diretório e compilar o programa): **make compile**
- Se deseja executar um quicksort recursivo, digite na linha de comando:
    - **./bin/tp2 quicksort -v 1 -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um quicksort mediana, digite na linha de comando:
    - **./bin/tp2 quicksort -v 2 -k <número de elementos para determinar mediana> -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um quicksort seleção, digite na linha de comando:
    - **./bin/tp2 quicksort -v 3 -m <tamanho máximo necessário para seleção> -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um quicksort não recursivo, digite na linha de comando:
    - **./bin/tp2 quicksort -v 4 -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um quicksort empilha inteligente, digite na linha de comando:
    - **./bin/tp2 quicksort -v 5 -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um mergesort, digite na linha de comando:
    - **./bin/tp2 mergesort -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Se deseja executar um heapsort, digite na linha de comando:
    - **./bin/tp2 heapsort -s <semente> -i <nome arquivo de entrada> -o <nome arquivo de saída>**
- Para ativar o analisamem, basta incluir a flag "**-p <nome do arquivo de log a ser gerado>**" e para gerar os logs basta incluir a flag "**-l**".