Trabalho Prático 2

Brisa do Nascimento Barbosa

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
brisabn@ufmg.br

1. Introdução

Neste presente trabalho, teve-se como objetivo a avaliação de desempenho de diferentes algoritmos de ordenação, sendo eles cinco versões distintas do Quicksort e, logo, a comparação de sua melhor versão com o Mergesort e o Heapsort. Para além da implementação bruta de código, foi necessário também considerar também o significado de cada procedimento em termos de custo de processamento, assim foram considerados o número de comparação de elementos do vetor, o número de cópias de registros e o tempo total de cada ordenação. Os procedimentos utilizados serão pormenorizados adiante, com a especificação do uso de bibliotecas, métodos e conceitos de desenvolvimento em C++.

2. Métodos

O programa foi desenvolvido na linguagem C++, compilado em G++ da GNU Compiler Collection. O computador utilizado possui o processador i7 e memória RAM de 8GM. O sistema operacional instalado é o Windows 10, portanto, foi utilizada uma máquina virtual WSL (Subsistema Windows para Linux) para a compilação em Linux.

2.1. Estruturas de dados

A principal estrutura de dados utilizada foi a classe, utilizada para agregar todas as variações do Quicksort, permitindo considerá-las como parte de algo centralizado, e para a estrutura do Registro. Dessa forma, houve uma melhor modularização do código. Além disso, para o quicksort não recursivo e empilha inteligente, foi utilizada uma pilha encadeada.

2.2. Classes

Para o desenvolvimento do programa, foram montadas duas classes principais: Quicksort e Registro. A primeira possui funções próprias para cada versão de tal algoritmo de ordenação. Há também classes similares de Heapsort e Mergesort. Já o Registro armazena a chave, as strings e os números reais de cada registro. Ademais, para a pilha, existem quatro classes: TipoItem, TipoCelula, Pilha e PilhaEncadeada.

2.3. Funções

O Registro possui apenas duas funções básicas de gerar aleatoriamente as strings e os floats. Para fins de melhoria no tempo do código, foram desconsideradas as gerações aleatórias e esses atributos foram inicializados com valores padrões.

Cada função da classe Quicksort é uma versão quicksort, com isso, segue-se em seguida a descrição de cada um:

Quicksort Recursivo (1)

Esta é a versão clássica do quicksort e se baseia no paradigma de divisão e conquista. Neste sentido, é utilizada uma função auxiliar de partição que rearranja o vetor por meio de um pivô localizado no meio do vetor, assim, elementos maiores que ele são jogados para a direita e, menores, para a esquerda. Logo que identifica isso, os elementos são trocados. O processo continua recursivamente até que o indicador percorrendo a esquerda e a direita se cruzem.

Quicksort Mediana (2)

Esta versão adapta o quicksort clássico de forma que o pivô selecionado é a mediana entre k elementos do vetor. Para determinar a mediana, recorre-se a uma função “mediana”, onde é criado um vetor secundário que guarda k elementos aleatórios do vetor principal. Com isso, ordena-se este para encontrar a mediana utilizando uma função de ordenação de seleção. Após esse processo sucede-se a partição recursiva tal qual a versão (1).

Quicksort Mediana (3)

Esta versão adapta o quicksort clássico de forma que, quando a partição é menor que um determinado tamanho m, utiliza-se o algoritmo de seleção para ordená-la. Tal qual na versão (2), recorre-se a uma função secundária de seleção. Caso contrário, a partição é ordenada de maneira recursiva.

Quicksort não Recursivo (4)

Neste caso, ao invés de realizar partições recursivas, armazenam-se as faixas a serem ordenadas em uma pilha. Assim, enquanto a pilha não estiver vazia, determina-se a parte direita da faixa e a esquerda. Depois, empilham-se os intervalos, primeiro empilhando o de maior tamanho e, depois, o de menor tamanho. Logo, realizam-se partições ordenando esses intervalos e desempilhando após concluída.

Quicksort Empilha Inteligente (5)

Da mesma forma que o quicksort não recursivo, essa variação utiliza pilhas para armazenar os intervalos, com a única diferença que, ao invés de empilhar a maior partição primeiro, empilha-se a menor.

O heapsort e o mergesort funcionam da seguinte maneira no programa:

Heapsort

É parecido com o algoritmo de seleção, pois seleciona o maior elemento do vetor e o coloca na última posição. Este algoritmo constrói um heap (que funciona como uma árvore) de forma que, cada vez, rearranja o vetor para que o maior esteja no

lugar correto em relação ao maior anterior. Logo, é chamada recursivamente tal função de construir para cada subárvore do heap. Por fim, o vetor rearranjado estará ordenado.

Mergesort

Em princípio, este algoritmo divide o vetor em dois de maneira recursiva, os ordena, e une as partes ordenadas. A cada chamada, o intervalo é modificado dependendo de qual é o meio do vetor, assim, chama-se um mergesort do meio da direita e do meio da esquerda. Isso forma uma estrutura de árvore, que se organizará a cada nível e depois, juntará suas sub-árvores.

Além das funções dos algoritmos de ordenação utilizados para a análise computacional, também foram utilizadas funções específicas de pilha para os quicksorts não recursivos:

- *Empilha*: criar uma nova célula com o item passado como parâmetro;
- *Desempilha*: retira a célula do topo da pilha, de acordo com o princípio Last In, First Out

Já na main, a fim de gerar os conjuntos aleatórios, utilizou-se uma função que associa valores aleatórios a elementos de uma matriz dinâmica com 5 de comprimento da coluna e o tamanho de N como largura da linha. Logo, a cada conjunto a ser ordenado, chama-se uma função secundária que considera uma das colunas da matriz como um vetor. Assim, este vetor será ordenado pelo algoritmo de ordenação selecionado pelo prompt de comando.

Ademais, é importante pontuar que a estrutura de saída do arquivo gerado pela main é a seguinte, com quantos N forem requisitados na entrada:

N: *[tamanho do registro]*
---- [Métrica] ----
[estatística de cada um dos 5 conjuntos separada por vírgula]
Média = [média dos 5 conjuntos]

3. Análise de complexidade

Quicksort Recursivo (1): Esse algoritmo percorre todo o vetor a ser ordenado, ou seja, $O(n)$, e a recursão tem complexidade $O(\log n)$. No melhor caso, a complexidade é $O(n \cdot \log n)$. Como no programa o pivô nunca vai ser o mais à esquerda, já que escolhe o elemento do meio, o pior caso neste caso não pode ser $O(n^2)$, e sim será quando o elemento máximo ou mínimo é sempre escolhido como pivô. Em termos de espaço, essa versão tem complexidade $O(\log n)$,que é a "altura" da árvore de partições.

Quicksort Mediana (2): Além das complexidades da implementação recursiva explicitada acima, esse algoritmo realiza uma ordenação por inserção para ordenar o vetor secundário para obter a mediana. Logo, a complexidade no melhor caso é $O(n) + O(n \cdot \log n) = O(n \cdot \log n)$. Já no pior caso e no caso médio, é $O(n^2) + O(n \cdot \log n) = O(n \cdot \log n)$. Em termos de complexidade de espaço, a ordenação por

inserção ocupa O(1) no pior caso, assim, o pior caso é a mesma coisa que no item anterior, *O(log n).*

Quicksort Seleção (3): Tal algoritmo agrega ao quicksort clássico ao realizar ordenação por seleção caso a partição seja menor que um determinado tamanho m. Logo, a complexidade no melhor caso é $O(n^2) + O(n \cdot log\ n) = O(n \cdot log\ n)$, *assim como* no pior caso e no caso médio, é $O(n^2) + O(n \cdot log\ n) = O(n \cdot log\ n)$. Em termos de espaço, a complexidade é *O(log n),* já que no pior caso a seleção tem complexidade de espaço *O(1).*

Quicksort não Recursivo (4): Como essa variação se baseia no uso de pilhas, analisaremos a partir da complexidade dessa estrutura de dados. Primeiramente, o 'do while', representa O(n), e dentro dele, é realizada uma partição *O(log n)* e são empilhados e desempilhados itens, que representa O(1). Assim, a complexidade temporal é $O(n \cdot log\ n)$. Já em termos de espaço, é ocupado um espaço extra de O(n) pelas pilhas, o que tornará a complexidade de espaço a ser $O(n) + O(log\ n) = O(log\ n)$.

Quicksort Empilha Inteligente (5): Essa variação apenas muda a ordem de empilhamento do algoritmo, logo, manterá a complexidade de espaço e tempo do item anterior.

Heapsort: A função de construir o heap, no pior caso, percorre um galho inteiro da árvore binária, de altura log n. Isso é executado n/2 vezes (nós internos) e executa a cada heap construído. Assim, a complexidade do algoritmo é $O(n \cdot log\ n)$.

Mergesort: Funciona como uma estrutura de árvore de altura log n, e a operação de merge tem um custo linear n. Assim, a complexidade do algoritmo é $O(n \cdot log\ n)$. Tal qual o quicksort, é um algoritmo que utiliza o paradigma de divisão e conquista.

4. Estratégias de Robustez

A fim de firmar a robustez do código, foi utilizada a biblioteca <msgassert.h>, a qual define macros para verificar a precisão das funções.

Na classe de **pilha**, são definidos os seguintes asserts:

- Erro se a pilha estiver vazia ao desempilhar

Na classe de **Quicksort**, são definidos os seguintes asserts

- Erro se o intervalo for inválido
- Erro se o parâmetro k não for definido ou válido
- Erro se o parâmetro m não for definido ou válido
- Erro se o empilhamento do quicksort 4 ou 5 falha

Na **main**, são definidos os seguintes asserts

- Erro se é informado um número inválido de versão do quicksort;
- Erro se é informado um parâmetro k inválido;

- Erro se é informado um parâmetro m inválido;
- Erro se é informado uma semente inválida;
- Erro se nenhum método de ordenação é selecionado
- Erro na abertura dos arquivos de entrada e saída;
- Erro no fechamento do arquivos de entrada e saída;

5. Análise Experimental

Houveram duas etapas para a análise experimental dos algoritmos de ordenação. Primeiramente, analisou-se as variações do quicksort a fim de identificar a melhor, e, depois, comparou-se essa com os algoritmos de ordenação heapsort e quicksort.

Nesses casos, apenas as chaves dos registros foram consideradas, tendo em vista que o computador ficava extremamente lento ao considerar a cópia inteira das strings e dos floats. Um caso de exemplo foi com o quicksort 1, que chegou a variar de 0,2 milissegundos para 44,7 milissegundos considerando o Registro inteiro Nos algoritmos em que o número de cópias é alto, então mais lento o computador ficaria ao copiar os registros, já que demandava mais desempenho da memória, afinal a cada N, são 5 conjuntos de 15 strings e 10 floats, além da chave. Logo, considerar nas análises foi inviável por conta do processamento da máquina e tempo demandado.

5.1. Variações do Quicksort

A fim de avaliar o desempenho computacional do programa foram consideradas três métricas: número de comparações, número de cópias e tempo de execução (em milisegundos). Essas métricas foram obtidas pelas médias entre cinco conjuntos aleatórios de N elementos. Foram selecionados 7 valores de N diferentes: 1000, 5000, 10000, 50000, 100000, 500000, 1000000. As tabelas podem ser encontradas na seção 5.4.1.

Inicialmente, pôde-se perceber que, apesar dos valores de cópias se manterem os mesmos, há uma significativa variação em relação ao tempo de execução a cada vez que se roda o programa, já que essa métrica varia de acordo com o desempenho da máquina. Portanto, este não se mostrou como o melhor comparativo entre algoritmos, apesar de relevante para diferenças grandes de valores entre versões do quicksort. Por exemplo, o quicksort clássico aparenta ser mais rápido que o quicksort não recursivo implementado, já que a diferença se mostrou maior que 100 ms para vetores com um milhão de elementos em um dos testes realizados. Além disso, depois de diversos experimentos em diferentes momentos, percebeu-se uma variação maior entre o Quicksort Empilha Inteligente e o não Recursivo, que inicialmente parecia não variar tanto. Em um dos testes realizados com um milhão de elementos, chegou a variar em 300 ms. Portanto,

pôde-se concluir uma melhora aparente entre o (4) e (5) em relação ao tempo, mas não em desempenho computacional.

Quanto às outras métricas, essas tabelas expõem aperfeiçoamentos pertinentes a partir das modificações do algoritmo e seleções de valores k e m. Quanto ao quicksort (2), foram utilizados valores k=3, k=5 k=7, e percebeu-se que, com o aumento de k, o algoritmo tornava-se menos eficiente, especialmente no tempo, que demonstrou significativa variação, e principalmente por realizar muito mais cópias e comparações. Isso se dá pois, para obter a mediana, foi necessário utilizar um vetor auxiliar para armazenar os valores aleatórios e, em seguida, ordená-los por seleção, o que aumentou significativamente o número de comparações e cópias feitas, já que a função de calcular a mediana é chamada toda vez que a recursão acontece.

Já no quicksort (3), foram utilizados valores m=10 e m=100, e, ao contrário de (2), o desempenho melhorou com o aumento de k (apesar de que, para valores de m muito grandes, essa melhora não persiste, já que consideraria grande parte do vetor para uma ordenação de seleção, que tem complexidade quadrática). Quando selecionamos m=10, há uma pequena, mas não significativa, mudança no número de comparações, sendo melhor que ao quicksort clássico nesse quesito, mas ligeiramente pior quanto ao número de cópias realizadas. Já com m=100, a melhora é nítida: este é o melhor dos algoritmos implementados ao se comparar com quicksort clássico, tanto a métrica de cópias e comparações mostram valores inferiores e, apesar de gastar mais tempo nos testes realizados, a variação de milissegundos não é significativa. No entanto, após rodar o programa em outros momentos, o quicksort (3) foi sim o mais rápido.

Além da comparação das métricas, também é válido considerar as diferenças na localidade de referência do programa. A partir do uso da biblioteca memlog, foram obtidos logs da performance de cada versão do quicksort. Para melhor visualização dos gráficos, os testes foram realizados com um único tamanho de vetor N = 2000. Tais gráficos podem ser visualizados na seção 5.3 deste documento. No programa, são realizadas *leituras do memlog* ao se comparar elementos e *escrita do memlog* ao fazer cópias . Considerou-se cada troca de elementos do vetor como uma cópia.

Os gráficos gerados pelo gnuplot permitem verificar o comportamento básico do algoritmo de quicksort – que é comparar cada elemento com um determinado pivô. Isso é averiguado pelo comportamento “zigue-zague” tanto nas curvas de escrita (com ID 1) quanto leitura (com ID 0), já que ao comparar percorrendo o vetor, são acessados diversos endereços até finalizar a partição. Além disso, como cada algoritmo de ordenação foi realizado 5 vezes, uma para cada conjunto, há uma periodicidade no padrão de escrita e leitura.

Os algoritmos que se baseiam no quicksort recursivo (variação 1, 2 e 3) mostram uma similaridade quando o comportamento linear bem definido no gráfico de acesso. Isso ocorre pois a função de partição (que faz as cópias do vetor ordenado) é chamada recursivamente em cada um desses casos e realiza

corretamente a iteração pelo vetor. Já os gráficos de escrita e leitura da variação 4 e 5 são menos “limpos” tendo em vista que, apesar do comportamento parecido, há muitos mais acessos em “conglomerados”, já que está empilhando sempre em uma pilha. Assim, a partição pega o intervalo direto do empilhamento, e não recursivamente.

Percebeu-se também que a mudança de parâmetros k e m não mudava com grande relevância o comportamento dos gráficos. Foi interessante, contudo, perceber que, com valores menores de k, o gráfico parecia com o quicksort clássico.

Em relação à evolução da distância de pilha e distância de pilha total, todos os gráficos foram bem parecidos. Nada muda substancialmente em se tratando dos vetores armazenados, já que todos são gerados com as mesmas sementes e tamanhos e são , assim, a distância de pilha computada é bem parecida.

5.2. Quicksort X Mergesort X Heapsort

A princípio, após delicada análise de comparação das métricas e testes com diferentes tamanhos de vetores, evidenciou-se o quicksort de seleção com m=100 como o mais efetivo. Logo, comparemos esse com outros algoritmos de ordenação: o mergesort e o heapsort.

Para comparar os algoritmos, foram consideradas 5 sementes de aleatoriedade a cada tamanho de vetor distinto. Isso foi realizado, em termos de código, modificando a função randômica original, substituindo-a por uma função srand() que tem como semente um rand(), de forma a gerar uma nova semente cada vez dentro do while que realiza a ordenação para cada conjunto. Após rodar o programa, obtemos as tabelas presentes na seção 5.4.1.

Com mais aleatoriedade no código, pode-se perceber uma melhora do quicksort 3 com m=100, passando a melhorar de forma significativa no tempo em relação à ordenação com valores menos aleatórios. Isso também ocorre com qualquer outra variação do quicksort. A melhora calculada foi de, em média 40% para comparação e o tempo, mas, quanto ao número de cópias, isso piorou em média 35%. Isso provavelmente pode ser explicado que, como os números são mais aleatórios, o número de diferentes cópias será maior.

Já em relação aos outros algoritmos de ordenação, é nítida a prevalência do quicksort como o que possui melhor desempenho computacional, haja vista, que em todas as métricas, este é o que apresenta melhores resultados, ou seja, o desempenho da memória e temporal é mais atrativo, gasta menos memória e é mais rápido. Em relação ao tempo, a diferença com o mergesort não foi tão abrangente em determinados testes, mas isso dependeu muito do desempenho da máquina no momento que o programa foi rodado (houve variações, por exemplo, de 200 ms até 400 ms). Então, caso a máquina seja mais potente, o mergesort demonstrou como um bom substituto do quicksort caso o objetivo seja o menor custo de tempo. Mas caso o objetivo seja desempenho da memória, este não é mais válido, já que demanda demasiadamente mais. Isso se dá porque o mergesort usa um vetor temporário, logo, não é *in-place* como o quicksort.

Essas diferenças demonstram o cerne da diferença de paradigmas dos algoritmos e seus métodos de ordenação. Ambos os algoritmos que se baseiam em divisão e conquista são os mais rápidos, mas o quicksort demonstra sua superioridade quando analisamos o custo de cópias e comparações. Com o heapsort, é necessário fazer a troca de elementos toda vez, e no mergesort, é necessário escrever todos os elementos em outro vetor para depois copiá-lo novamente para o original. Então o custo de memória desse último é muito alto. Já no quicksort, não é realizada troca a não ser que seja necessário realizar, e no melhor caso, o número de comparações é o mínimo.

Ainda, ao analisar os gráficos gerados pelo gnuplot, o comportamento de cada algoritmo fica ainda mais claro. Comparamos a relação das escrita/leitura do memlog com o princípio do algoritmo:

- **Quicksort**: percorre o vetor e troca elementos dependendo de um pivô → O seu comportamento no gráfico sobe até determinados valores, isto é, percorre posições, troca e realiza recursão.
- **Heapsort**: seleciona o maior elemento e rearranja recursivamente até que estejam todos em ordem crescente → O seu comportamento no gráfico de escrita é “triangularmente”, ele está acessando cada endereço de memória procurando o maior.
- **Mergesort**: este é o mais intrigante. O algoritmo ordena os subgalhos de uma estrutura de árvore. → O seu comportamento no gráfico mostra ramificações em “fractal”.

Os gráficos de evolução de distância de pilha e distância são menos interessantes, já que são bastante parecidos. O que mais chamou atenção foi que o gráfico do heapsort é mais preenchido do que do heapsort e do quicksort, demonstrando que há menos distância entre as pilhas, ou seja, parecem estar mais próximas na memória, já que sempre acessa várias posições para achar o maior.

5.4. Tabelas de Métricas

1. **Variações do Quicksort**

   Tabelas obtidas com semente de aleatoriedade = 10.

Quicksort 1			
N	Comparações	Cópias	Tempo
1000	7969	2560	0.20936
5000	47219	15619	1.24306
10000	105001	33591	1.5766
50000	645100	194377	9.23004
100000	1361015	412400	17.5767
500000	7763017	2334766	84.9168
1000000	16496345	4899966	178.901

Quicksort 2 (k=3)			
N	Comparações	Cópias	Tempo
1000	6970	4529	0.27542
5000	44106	25422	1.09454
10000	97231	53208	2.70196
50000	558771	293757	14.4375
100000	1188630	611530	26.3526
500000	6779659	3331939	139.925
1000000	14121863	6911187	292.775

Quicksort 2 (k=5)			
N	Comparações	Cópias	Tempo
1000	7631	6382	0.27152
5000	46195	34796	1.3026
10000	100721	71810	2.52902
50000	580619	386839	14.7491
100000	1206046	798412	31.3
500000	6763269	4270958	175.927
1000000	14206325	8782936	363.155

Quicksort 2 (k=7)			
N	Comparações	Cópias	Tempo
1000	8869	8185	0.34474
5000	52544	43723	1.7685
10000	110679	89983	3.64946
50000	621119	478553	17.6644
100000	1311755	980837	39.2866
500000	7332722	5180843	215.381
1000000	15042115	10618496	445.845

Quicksort 3 (m=10)			
N	Comparações	Cópias	Tempo
1000	7818	2743	0.17938
5000	46327	16529	0.999
10000	103352	35405	1.65882
50000	636564	203511	9.80946
100000	1343916	430624	20.2434
500000	7677887	2425971	108.932
1000000	16325432	5082064	241.975

Quicksort 3 (m=100)			
N	Comparações	Cópias	Tempo
1000	6964	2077	0.21462
5000	42424	13164	1.00834
10000	95514	28617	2.1088
50000	596298	169355	9.33658
100000	1262557	362842	18.6882
500000	7271813	2086598	102.136
1000000	15512024	4403822	214.993

Quicksort 4			
N	Comparações	Cópias	Tempo
1000	7969	2560	0.22954
5000	47219	15619	1.20106
10000	105001	33591	2.55714
50000	645100	194377	14.9933
100000	1361015	412400	24.9928
500000	7763017	2334766	143.806
1000000	16496345	4899966	292.265

Quicksort 5			
N	Comparações	Cópias	Tempo
1000	7969	2560	0.2517
5000	47219	15619	1.0769
10000	105001	33591	2.04226
50000	645100	194377	11.408
100000	1361015	412400	20.6108
500000	7763017	2334766	129.339
1000000	16496345	4899966	273.537

2. Quicksort X Mergesort X Heapsort

Tabelas obtidas gerando uma semente aleatória para cada valor de m. Foi considerado o quicksort 3 com m = 100 por ter se apresentado como o mais eficiente no programa.

Quicksort 3 (m=100)			
N	Comparações	Cópias	Tempo
1000	3192	2380	0.2078
5000	17872	17104	1.16092
10000	37083	38937	1.61736
50000	188795	250408	7.47098
100000	393697	550689	14.1045
500000	1754440	3346881	69.9839
1000000	3600592	7188630	151.79

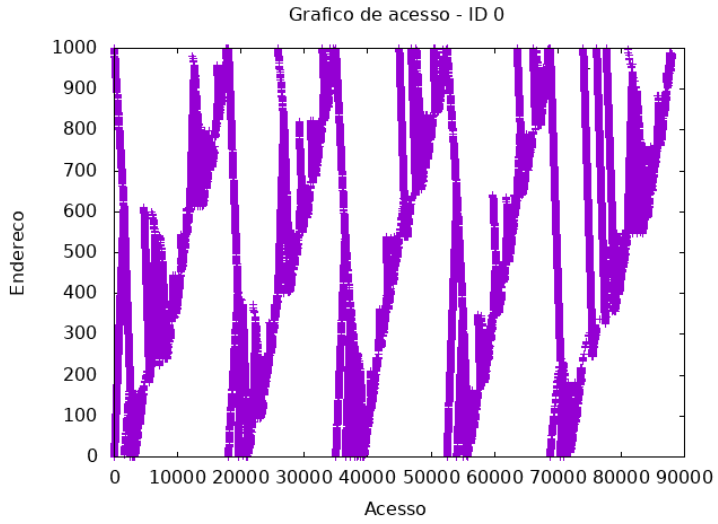
Heapsort			
N	Comparações	Cópias	Tempo
1000	9725	7483	0.2435
5000	62498	46840	1.42574
10000	136930	101529	3.11826
50000	834526	607811	17.9211
100000	1788673	1295199	43.3645
500000	10408039	7440281	204.744
1000000	22049512	15702066	368.568

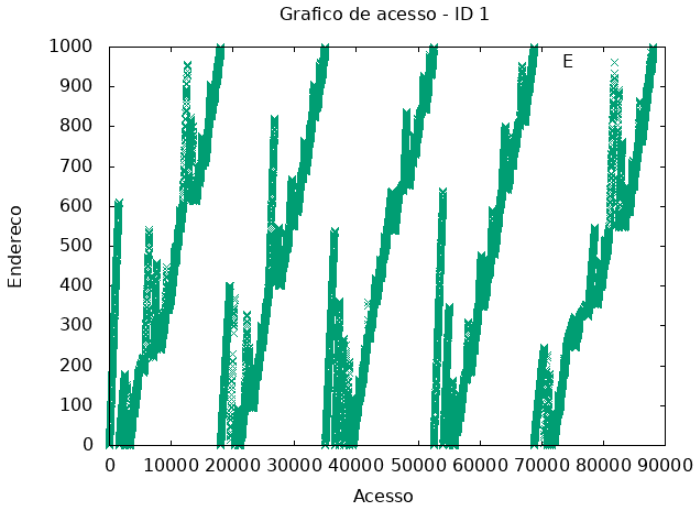
Mergesort			
N	Comparações	Cópias	Tempo
1000	8621	19952	0.2273
5000	54514	123616	1.379
10000	118639	267232	3.28666
50000	706388	1568928	24.0205
100000	1510900	3337856	37.536
500000	8681472	18951424	148.538
1000000	18338154	39902848	292.564

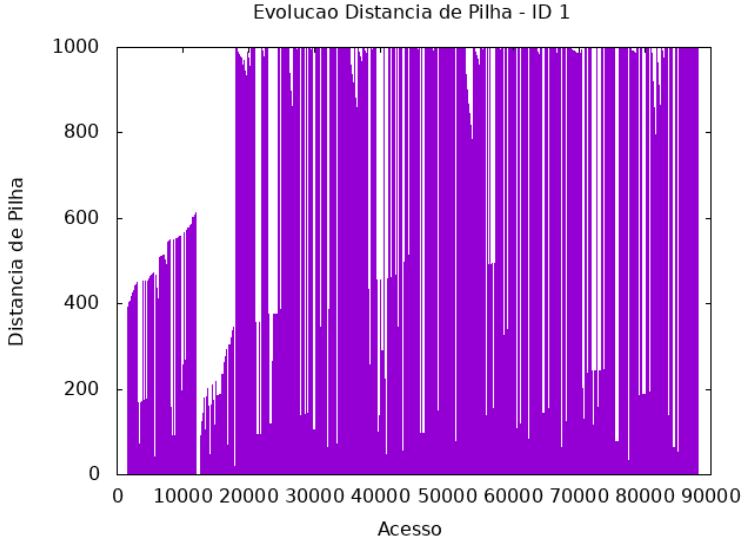
5.3. Gráficos

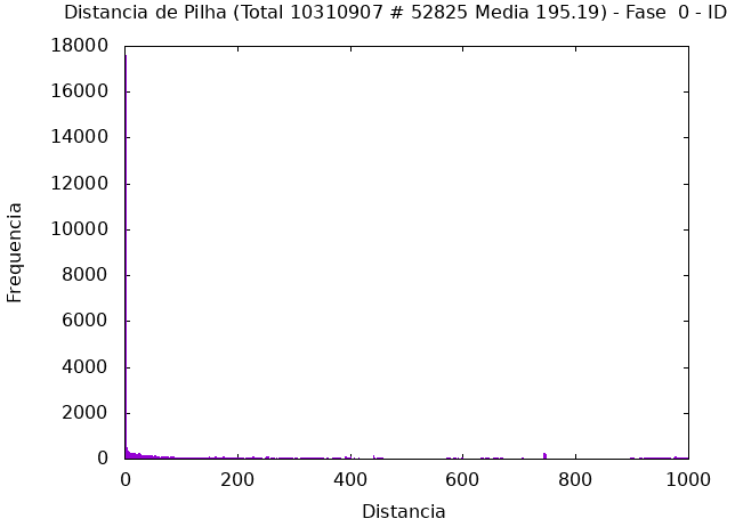
Para melhor visualização, a leitura está no ID 0 e a escrita no ID 1.

1. Quicksort Recursivo

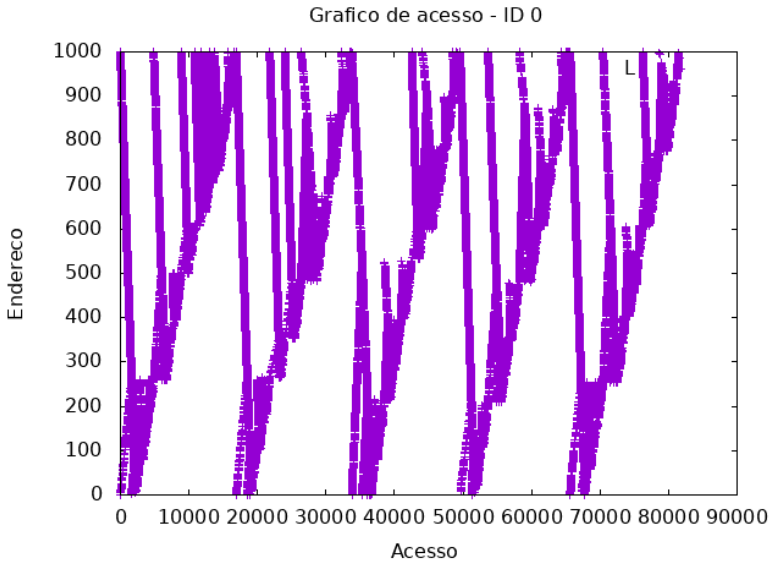


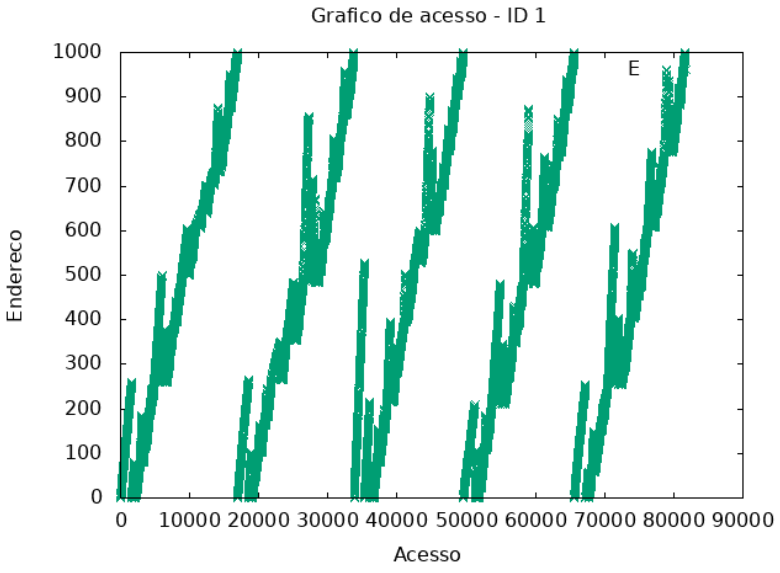


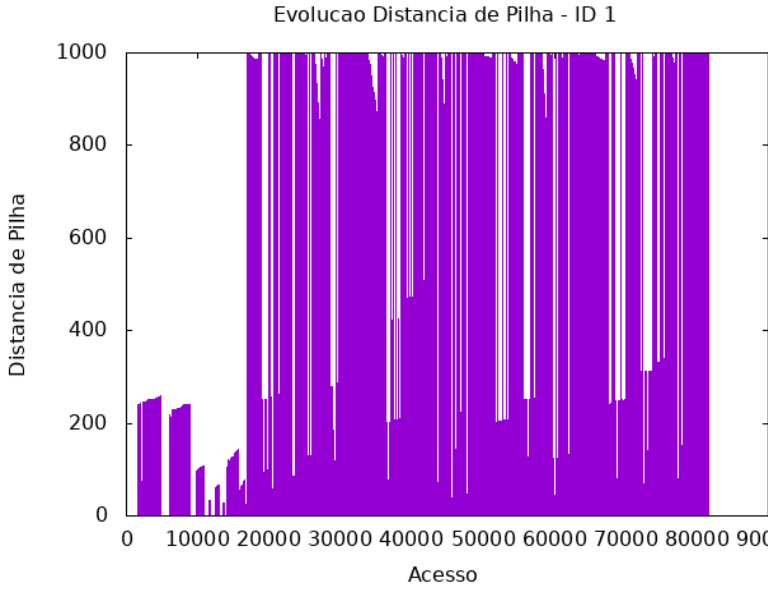


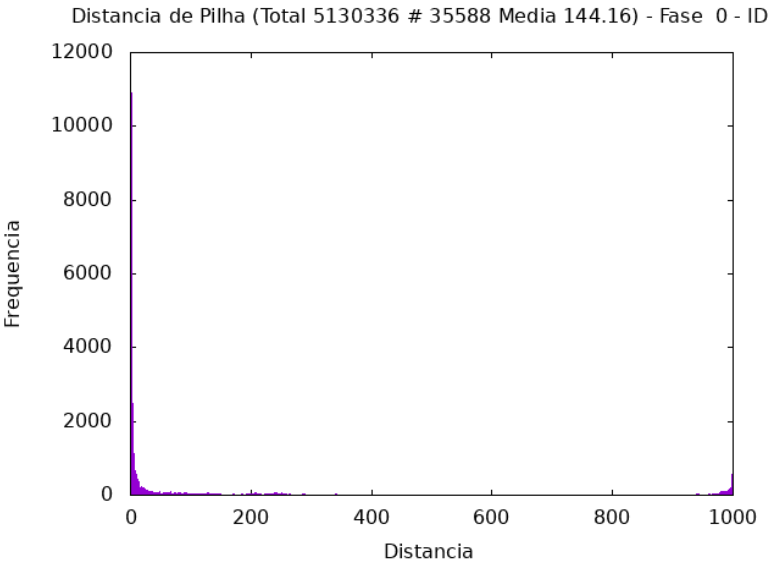


2. Quicksort Mediana

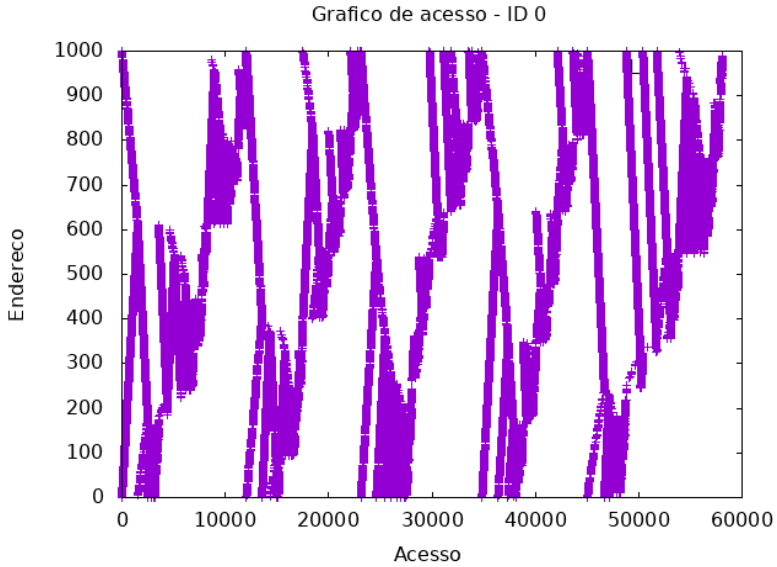


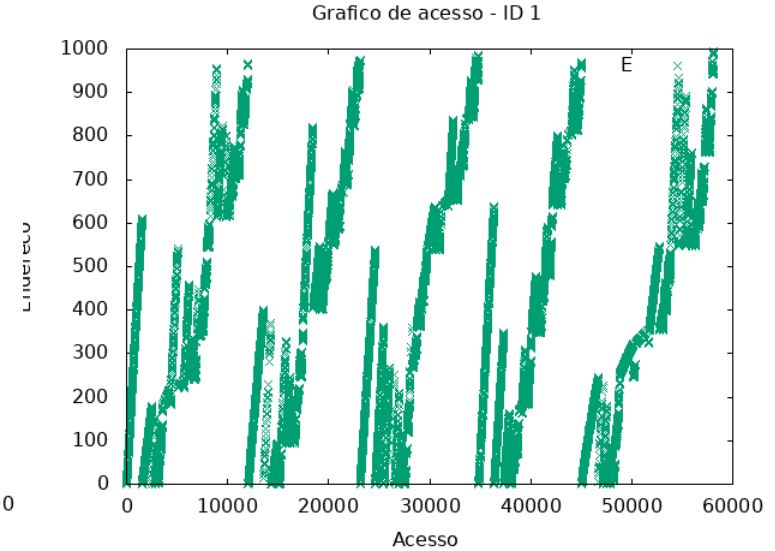


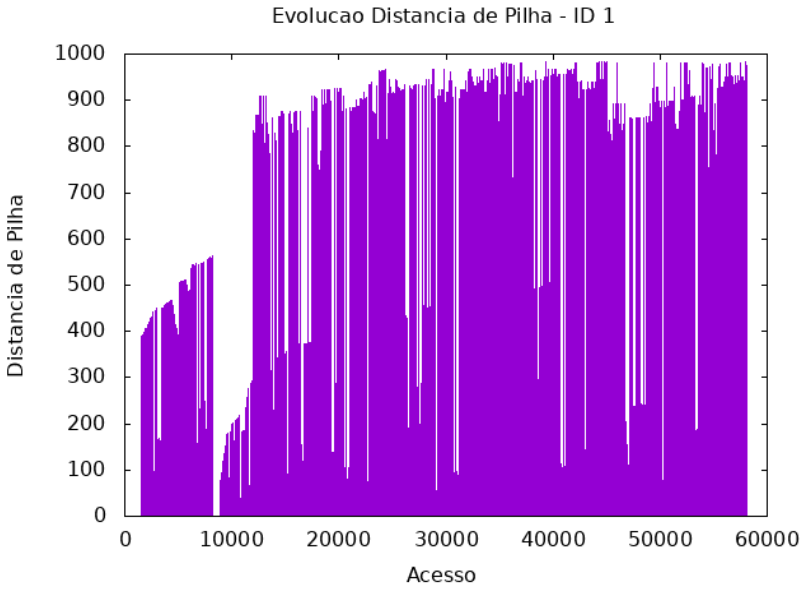


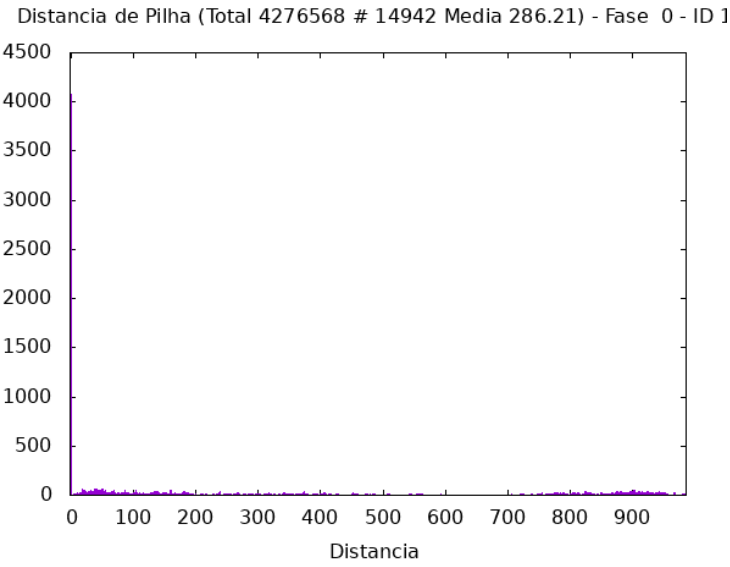


3. Quicksort Seleção

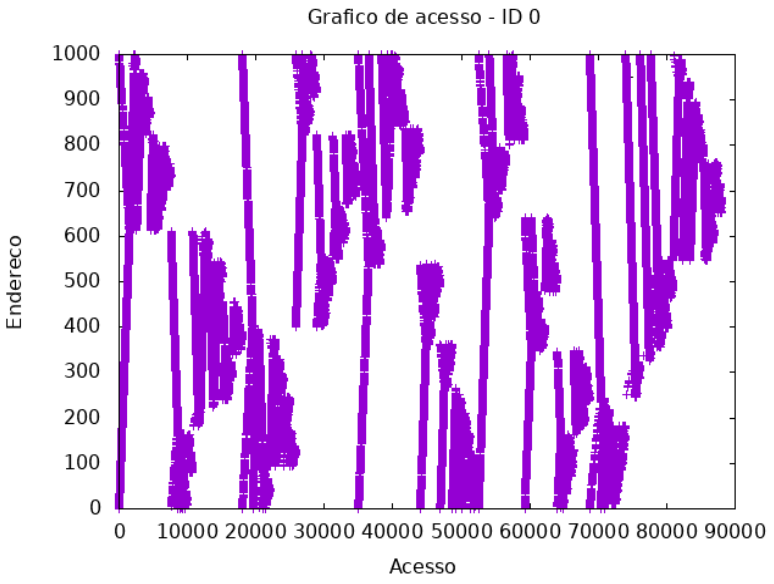


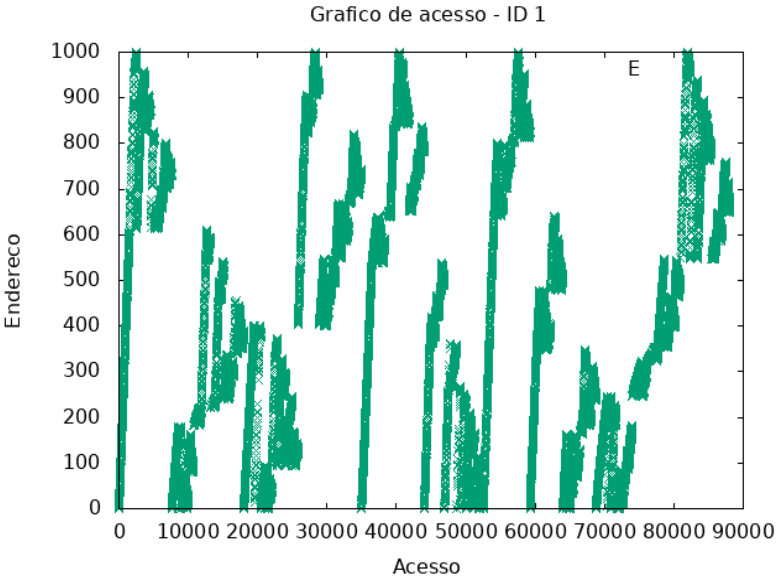


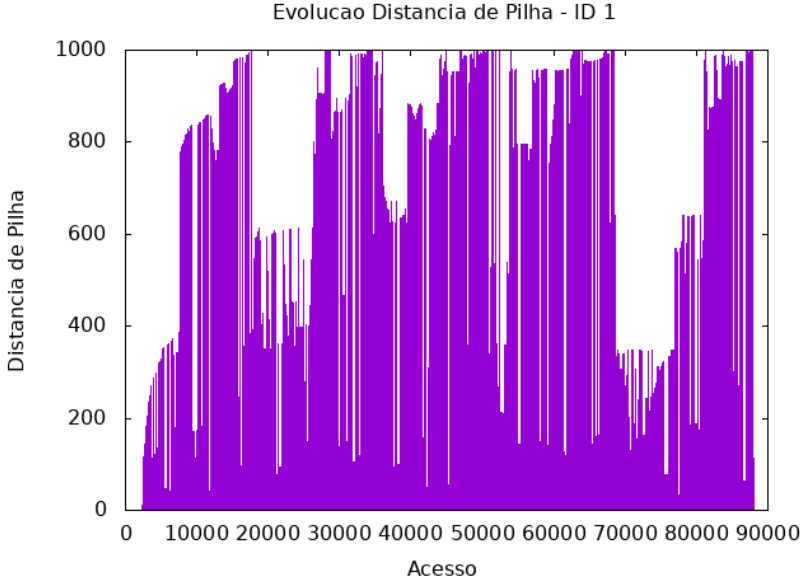


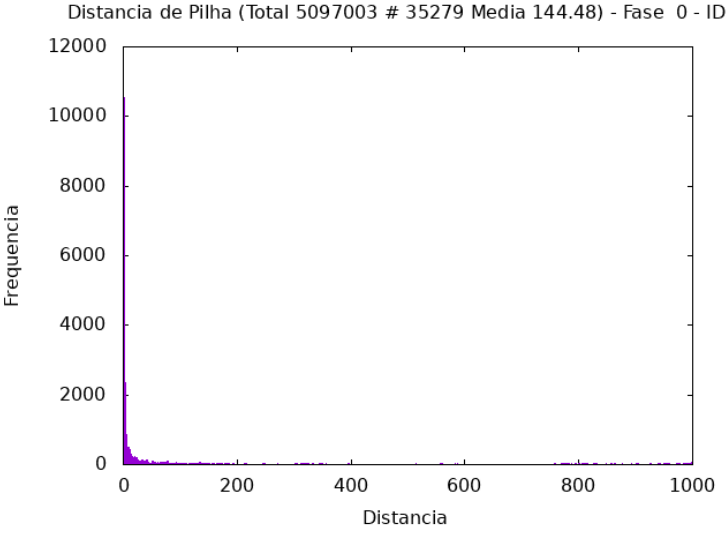


4. Quicksort não Recursivo

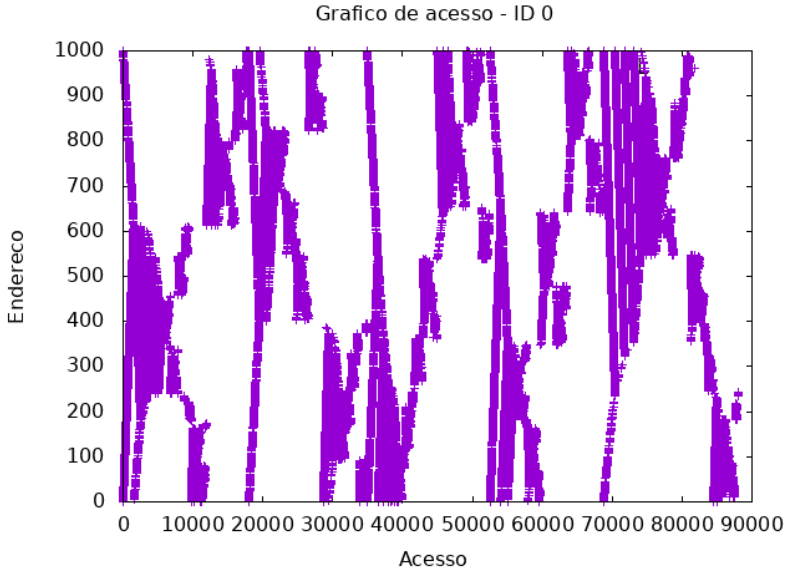


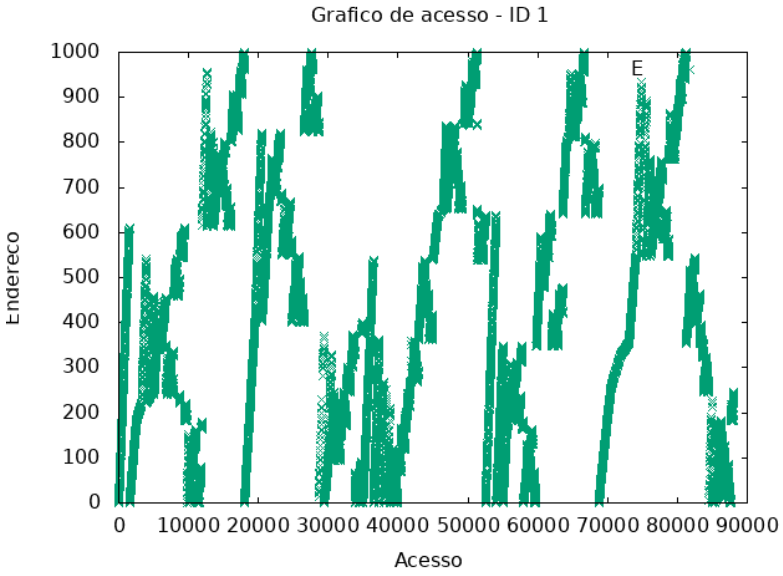


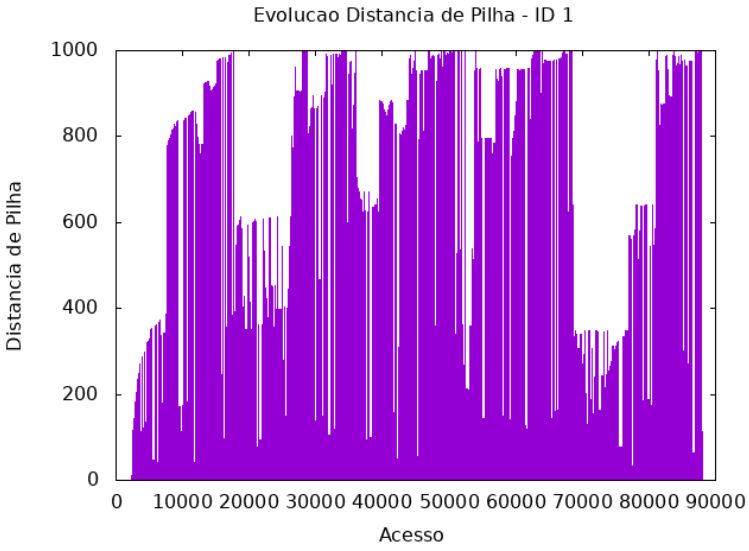


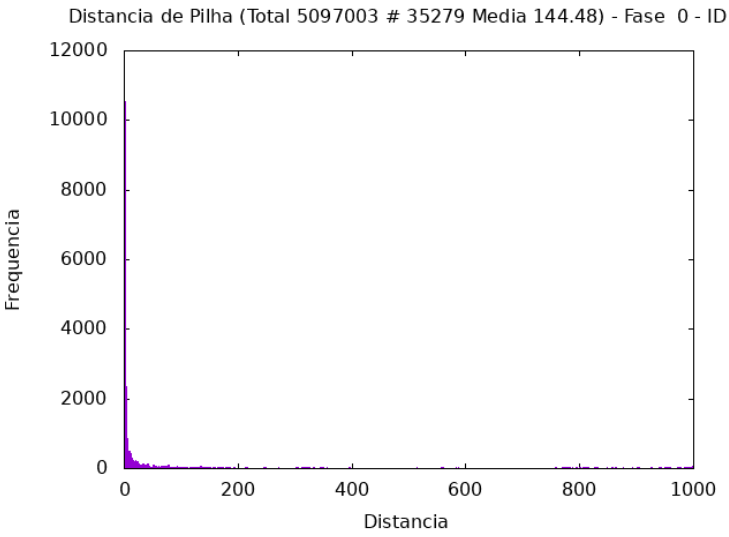


5. Quicksort Empilha Inteligente

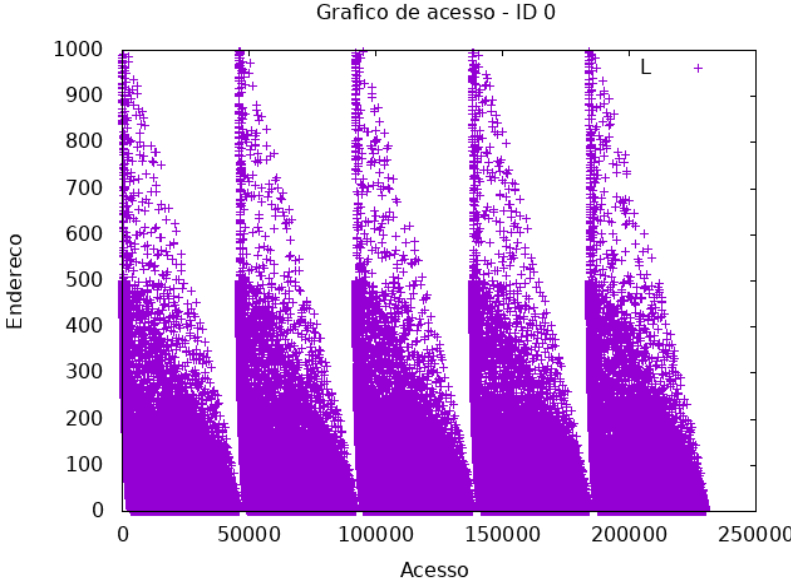


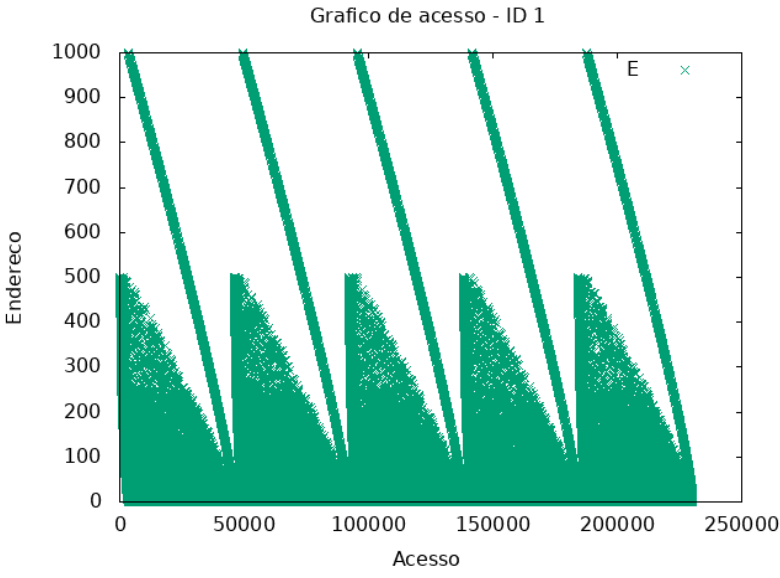


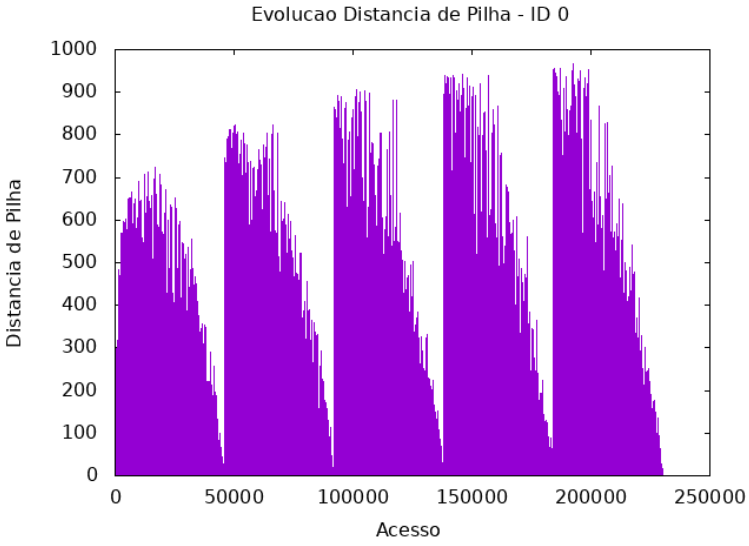


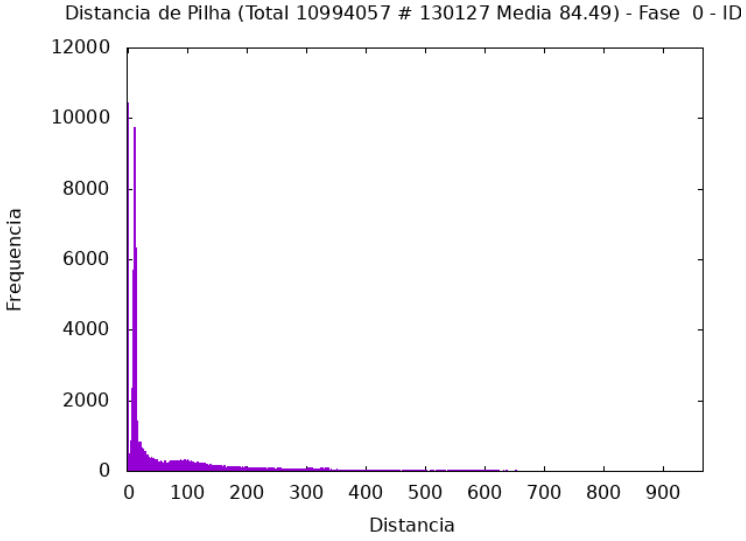


6. Heapsort






Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
100
200
300
400
500
600
700
800
900
1000
0
50000
100000
150000
200000
250000


Distancia de Pilha (Total 10994057 # 130127 Media 84.49) - Fase 0 - ID
Frequencia
Distancia
0
2000
4000
6000
8000
10000
12000
0
100
200
300
400
500
600
700
800
900

7. Mergesort

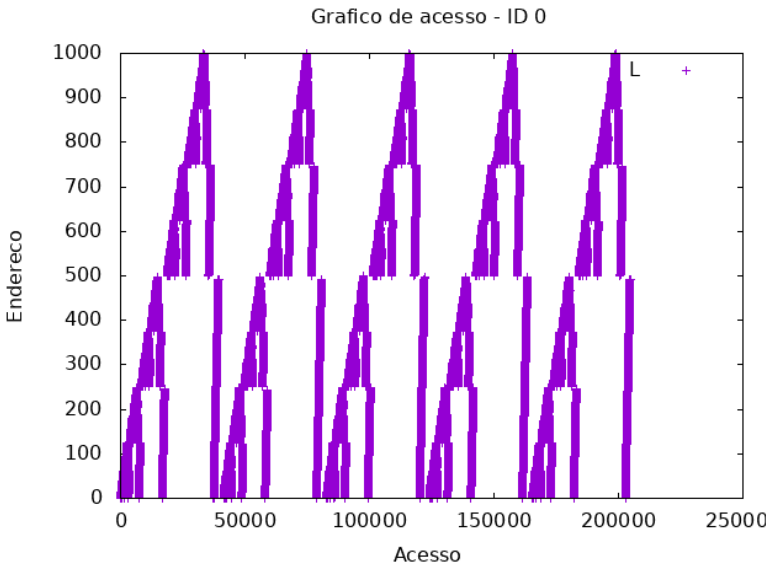


Grafico de acesso - ID 0
L
Endereco
Acesso
0
100
200
300
400
500
600
700
800
900
1000
0
50000
100000
150000
200000
25000

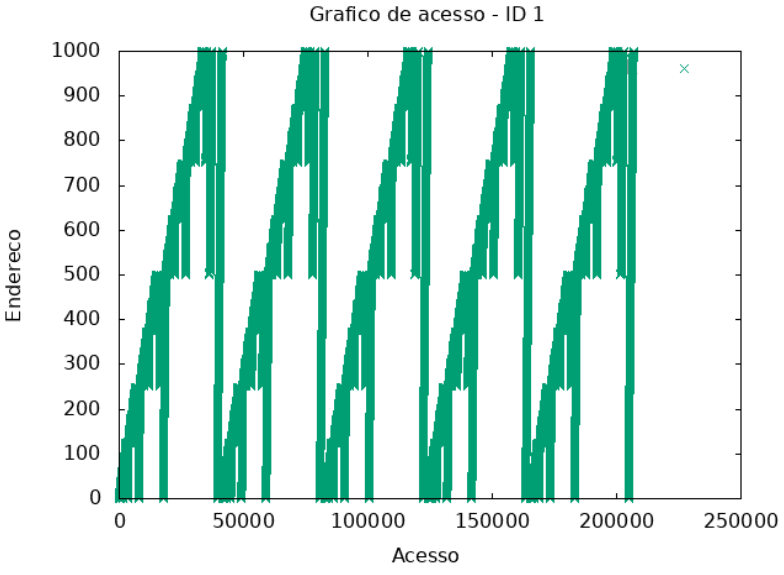
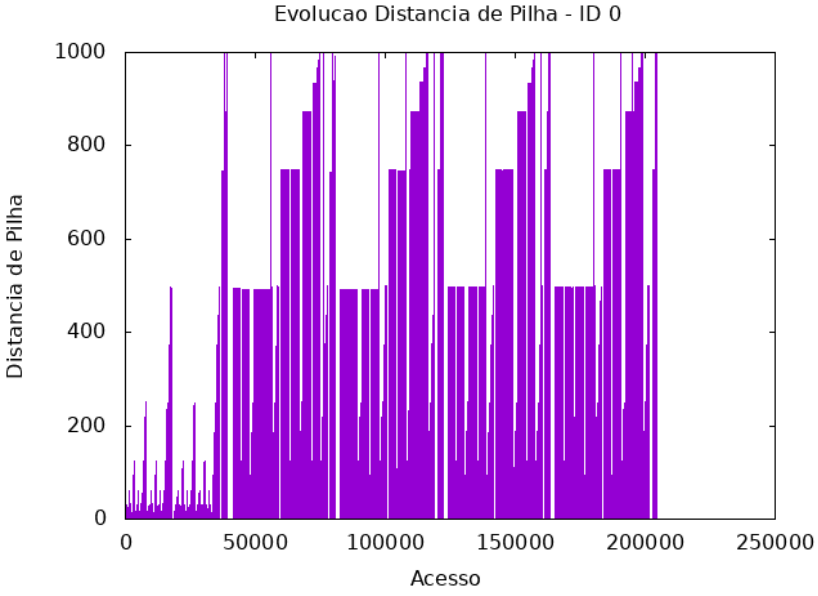
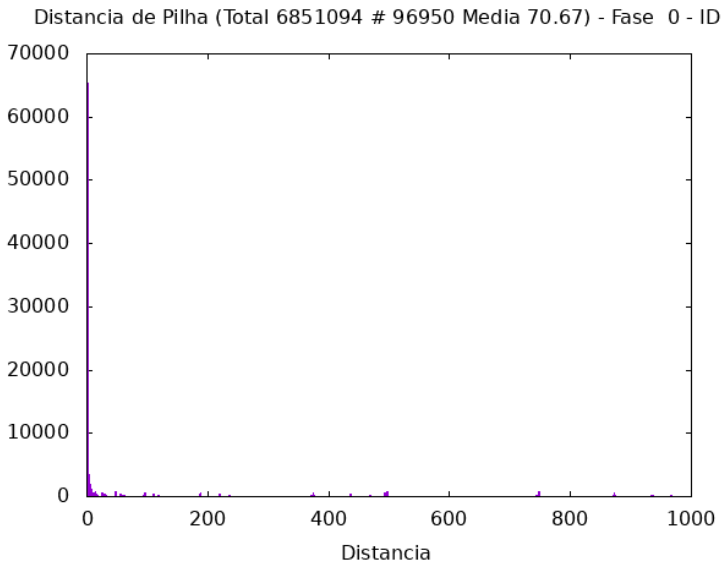


Grafico de acesso - ID 1
Endereco
Acesso
0
100
200
300
400
500
600
700
800
900
1000
0
50000
100000
150000
200000
250000


Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
200
400
600
800
1000
0
50000
100000
150000
200000
250000


Distancia de Pilha (Total 6851094 # 96950 Media 70.67) - Fase 0 - ID
Distancia
0
10000
20000
30000
40000
50000
60000
70000
0
200
400
600
800
1000

Observação: os gráficos de evolução distância de pilha e distância total de pilha são altamente parecidos para o ID 0 e ID 1, não sendo necessário sua repetição neste documento.

6. Conclusão

A partir do desenvolvimento do programa, pode-se compreender com maior afinco o desempenho computacional de algoritmos de ordenação. Para tal, foi necessário implementar adequadamente funções que realizam este procedimento, para, então poder compará-los pelo número de comparação de chaves, número de cópias de registro e o tempo de execução. Dessa forma, foram testadas cinco variações do quicksort e, após identificar a melhor, comparou-se com o heapsort e o mergesort.

Contudo, ao longo da análise dos algoritmos, houve um desafio considerável quanto à compreensão dos gráficos gerados da biblioteca <*memlog*>, já que não foi claro identificar a lógica de comportamento desses. Ademais, ao gerar os dados das métricas de cada algoritmo, foi necessário despender um considerável tempo para analisar o porquê da variação desses valores. Apesar do custo, isso foi verdadeiramente útil para entender melhor o funcionamento do código, ao invés de apenas fazê-lo funcionar.

Além disso, pela alta quantidade de valores testados, houve uma dificuldade considerável quanto ao desempenho da máquina, pois, apesar de gerar corretamente os resultados esperados e em tempo hábil, desgastavam muito da máquina virtual WSL, principalmente ao considerar as strings e os números reais do registro. Assim, a cada vez que rodava o programa, ficava mais lento, e ao produzir os dados do memlog, foi necessário realizar “inicialização limpa” do computador mais de uma vez para otimizar o WSL.

Por fim, com a finalização do programa e subsequentes análises computacionais, pode-se verificar a relevância da compreensão de algoritmos de ordenação, nesse caso do quicksort, heapsort e mergesort, para um entendimento correto do armazenamento de memória e número de operações realizadas em um código – que podem ficar significativamente grandes.

Bibliografia

- Cormen , T., Leiserson, C, Rivest R., Stein, C. *Introduction to Algorithms, Third Edition, MIT Press, 2009.*
  Chapter 1: Foundations.
- Chaimowicz, L. and Prates, R. (2020). *Slides virtuais da disciplina de estruturas de dados*. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções de compilação e execução

O makefile possui dois comandos principais: make (compila arquivos recentemente modificados) e make all (compila todos os arquivos). Para rodar corretamente, deve-se utilizar um terminal e digitar make para compilar.

Em seguida, há três opções de algoritmo de ordenação: *quicksort*, *heapsort* e *mergesort*. Se for um quicksort, coloque *-v [variação do quicksort]*. Há cinco variações possíveis, enumeradas de 1 a 5. Os argumentos *-k [número de elementos para mediana]* e *-m [tamanho da partição]* informam, respectivamente, os parâmetros do quicksort 2 e 3. Para informar o arquivo de entrada, o argumento é *-i [nome do arquivo]*; para produzir um arquivo de saída, o argumento é -o *[nome do arquivo].* Além disso, para gerar uma semente de ordenação, coloque *-s [valor da semente]*.

Exemplos de linha de comando após compilado o programa são:

Quicksort 1: ./bin/run.out **quicksort -v** 1 **-s** 10 **-i** input.txt **-o** output.txt
Quicksort 2: ./bin/run.out **quicksort -v** 2 **-k** 3 **-s** 10 **-i** input.txt **-o** output.txt
Quicksort 3: ./bin/run.out **quicksort -v** 3 **-m** 10 **-s** 10 **-i** input.txt **-o** output.txt
Quicksort 4: ./bin/run.out **quicksort -v** 4 **-s** 10 **-i** input.txt **-o** output.txt
Quicksort 5: ./bin/run.out **quicksort -v** 5 **-s** 10 **-i** input.txt **-o** output.txt
Heapsort: ./bin/run.out **heapsort -s** 10 **-i** input.txt **-o** output.txt
Mergesort: ./bin/run.out **mergesort -s** 10 **-i** input.txt **-o** output.txt

Onde ./bin/run.out é o executável e input e output são, respectivamente, os arquivos de entrada e saída.

.