Trabalho Prático 2

Gabriela Tavares Barreto

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

gabrielabarreto@dcc.ufmg.br

1 Introdução

A prosposta deste trabalho prático era implementar diferentes algoritmos eficientes de ordenação, sendo eles cinco variações do QuickSort, e além desses, o MergeSort e o HeapSort. A tarefa consistia também em analisar o desempenho de cada um deles, dadas entradas de diferentes tamanhos. As métricas a serem comparadas eram: tempo de execução, número de cópias de registros, e o número de comparações das chaves dos registros.

Para cumprir essa proposta foram empregados, além dos métodos de ordenação, uma classe que contabilizava e armazenava os valores de cópias e de comparações. Foi usada também a função getrusage, para medir o tempo de processamento de cada método. Mais detalhes sobre a implementação serão dados a seguir.

2 Método

O programa foi desenvolvido nas linguagens C e C++, e compilado pelo G++ da GNU Compiler Collection, usando o WSL. O computador utilizado tem 12 GB de RAM.

3 Implementação

3.1 Struct Registro

Essa struct contém um inteiro **chave**, quinze strings de duzentos caracteres cada, armazenadas em uma variável chamada **strings** e por fim, um array com dez floats, chamado **array**.

3.2 MétodosOrdenação

A parte central desse trabalho se encontra no arquivo de nome MétodosOrdenação. Nele, é declarada uma classe chamada **InfoOrdenação**, que tem dois inteiros armazenados: **num_copias e num_comp**, referentes ao número de cópias e de comparações. Pertencem a essa classe todos os métodos de ordenação do trabalho. Eles operam sobre vetores de **Reistro**. Em cada método, os valores de num_cópias e num_comp são atualizados de acordo com as operações feitas. As funções dessa classe são as seguintes:

3.2.1 QuickSort Recursivo

- **void quick_sort_recursivo(Registro *A, int n):** recebe um vetor de Registros **A** e seu tamanho **n**, e chama a função ordena.
- **void ordena(int esq, int dir, Item *A):** recebe uma posição à esquerda **esq** e a posição à direita **dir** do vetor A, e chama o método de partição. A partir da atualização dos índices de esquerda e direita, faz duas chamadas recursivas, dividindo o problema de ordenar o vetor em dois problemas menores.
- **void particao(int esq, int dir, int *i, int *j, Registro *A):** resumidamente, essa função seleciona o registro do meio do vetor como um **pivô x**, e executa comparações das chaves dos elementos do vetor com a chave de x, fazendo trocas nas posições dos registros. Ao final do algoritmo de partição, o vetor A[esq, ..., dir] está particionado de forma que:
  - Os elementos em A[esq], até A[j] tem chaves menores ou iguais a de x;
  - Os elementos em A[i] até A[dir] tem chaves maiores ou iguais a de x.

3.3 QuickSort Mediana

O QuickSort Mediana replica o funcionamento do QuickSort Recursivo, por uma diferença na função de partição. Em vez de selecionar simplesmente o item do meio do vetor como pivô, é feita uma coleta de **k** itens aleatórios do vetor dado, e deles é escolhida a mediana como o pivô. Mais detalhes sobre esse método estão abaixo:

- **int rand_pos(int esq, int dir):** dado os índices de esquerda e direita, retorna aleatoriamente um posição entre [esq, dir].
- **Registro mediana(Registro *A, int esq, int dir, int k):** essa função recebe um inteiro **k**. Ela verifica se o vetor tem tamanho pelo menos k. Caso não, retorna o item do meio do vetor para ser o pivô, como no QuickSort Recursivo. Caso sim, ela cria um array de Registros de tamanho k, cada um recebe um Registro de A dadas as posições aleatórias geradas por rand_pos. Depois disso, ela ordena o array usando o QuickSort Recursivo, e retorna a mediana desse array.
- **void particao_mediana(int esq, int dir, int *i, int *j, Item *A, int k):** funciona igual a função partição, exceto que, para atribuir um valor ao pivô, usa a função mediana citada acima.
- **void ordena_mediana(int esq, int dir, Item *A, int k):** comportamento análogo a função ordena, faz uma chamada a partição mediana e duas chamadas recursivas.
- **void quick_sort_mediana(Item *A, int n, int k):** chama ordena_mediana.

3.4 QuickSort Seleção

Nessa variação do QuickSort, para partições menores, em vez de ser usado o método da partição regular, a ordenação dos elementos é feita usando o algoritmo simples de seleção. Detalhes das funções criadas para essa implementação estão abaixo:

- **void troca(Registro *a, Registro *b):** recebe dois ponteiros para registros, e troca o valor dos elementos para os quais eles apontam, fazendo uso de uma variável auxiliar.
- **void selecao(Registro *A, int esq, int dir):** esse método percorre o vetor, procurando o n-ésimo menor elemento de A. Depois, ele troca a posição do item menor, com a do n-ésimo

Resgitros de A, usando a função troca.

- **void ordena_selecao(int esq, int dir, Registro *A, int m):** tem um comportamento análogo à função ordena, exceto que, ao verificar que a partição tem tamanho menor ou igual a **m**, não segue o roteiro de chamar a função partição mais duas chamadas recursivas, e no seu lugar usa o método de seleção.
- **void quick_sort_selecao(Item *A, int n, int m):** chama ordena_seleção.

3.5 QuickSort Não Recursivo

Essa varição do algortimo não faz chamadas recursivas. Em vez disso ela faz uso de uma estrutura de dados do tipo Pilha para fazer um controle das partições a serem ordenadas posteriormente.

Antes de descrever mais profundamente esse método, é necessário explicar primeiro a implementação da struct **Index** e das classes **Célula e Pilha**;

3.5.1 Index

Armazena dois inteiros, **esq** e **dir**, que representam os índices do vetor.

3.5.2 Célula

Armazena um objeto do tipo Index, chamado **index** e um ponteiro para Célula denominado **prox**. Sua única função é um construtor que inicializa prox como NULL.

3.5.3 Pilha

Armazena um ponteiro para Célula denominado **topo**, e um inteiro chamado **tamanho**, que guarda o número de itens que a pilha contém. Tem como funções:

- **Pilha():** construtor que inicializa topo como NULL e tamanho como zero.
- **bool vazia():** retorna true se o tamanho for zero.
- **Index desempilha():** desaloca a memória do topo da pilha, atualizando a variável de tamanho, e atribui ao index que está abaixo do topo a posição de topo.Por fim retorna o index removido do topo, com o uso de uma variável auxiliar.
- **void limpa():** usa a função vazia para verificar a pilha: enquanto ela não estiver vazia, chama a função desempilha.
- **∼Pilha():** destrutor da classe; enquanto a pilha não está vazia, vai desempilhando seus elementos.

A função **void quick_sort_nao_recursivo(Item *A, int n)** faz uso das estruturas acima citadas da seguinte forma:

1. É criada uma pilha e uma variável do tipo index. São atribuídos aos campos esq e dir de index, os valores do tamanho total do vetor, sendo esq=zero e dir=n. Daí ela empilha esse item.
2. Depois disso, a função entra em um loop de do..while, que tem como condição a pilha não estar vazia para parar.

3. A seguir ela verifica se o índice da direita é maior do que da esquerda, se sim ela chama a função partição, atualiza os campos esq e dir de index, com os índices da partição esquerda, e o coloca na pilha.
4. Se o índice da direita não for o maior, significando que a ordenação já foi feita, ela desempilha um item da pilha.
5. As etapas 3 e 4 são repetidas até que a pilha esteja vazia, ou seja, a ordenação foi completa.

3.6 QuickSort Empilha Inteligente

Essa variação do quickSort, é uma otimização do Quicksort Não Recursivo. No QuickSort Não Recursivo, sempre é empilhado primeiro o lado esquerdo do vetor, e o direito é processado. Já no Empilha Inteligente, é verificado qual é o lado maior do vetor. Assim que ele é determinado, os campos esq e dir de index são atualizados, e o index é empilhado. Dessa forma a partição menor sempre é processada antes.

3.7 MergeSort

Essa função também usa o método de dividir para conquistar para ordenar o vetor. Ela divide o vetor repetidamente, até o partir em partes de tamanho um. A partir daí, ela vai reconstruindo o vetor, comparando os ellementos de cada parte, e os ordenando de acordo. As funções que compõe a implementação desse método são:

- **void merge(Registro *A, int esq, int meio, int dir):** essa função cria dinamicamente dois vetores menores, **vet_esq e vet_dir,** que servem como auxiliares para ordenar o vetor A. Isso é feito da seguinte forma: vet_esq recebe os itens do lado esquerdo de A, e vet_dir os itens do lado direito. Depois, ambos vetores são percorridos em busca da menor chave. Quando ela é identificada, o vetor original é atualizado em A[esq]. Isso é feito continuamente atualizando A[esq+1].. até A[dir]. No final da função, vet_esq e vet_dir tem a memória desalocada.
- **merge_sort(Registro *A, int esq, int dir):** enquanto o índice esquerdo esq for menor que o índice direito dir, ela calcula o meio do vetor, e faz sua divisão por meio de duas chamadas recursivas, uma que recebe os índices de esq até meio, e a outra de meio+1 até dir. Depois que essas chamadas recursivas retornam, a função merge é chamada, que ordena o vetor.

3.8 HeapSort

Essse método de ordenação possui o mesmo princípio de funcionamento que a ordenação por seleção. Ademais, ele faz uso de uma fila de prioridades, denominada heap. O heap é uma representação vetorial de uma árvore, que vai das posições 1 a n. Ele tem as seguintes características:

- O primeiro elemento do vetor representa a raiz da árvore, e é o item com a maior chave.
- A chave do nó pai é maior que a dos nós filhos.
- Os nós 2k e 2k+1 são filhos da esquerda e direita do nó k, para $1 \leq k \leq n/2$.

As funçõs usadas na implementação do HeapSort são dadas a seguir:

- **void refaz(Registro *A, int esq, int dir):** dado o índice da esquerda como nó pai, essa função analisa se seus seus filhos 2esq e 2esq+1 tem chaves menores, senão ela faz uma troca dos itens do vetor.

- **void constroi(Registro *A, int n):** chama a função refaz para cada nó pai do heap. Dessa forma, o heap é construído.
- **void heap_sort(Registro *A, int n):** primeiro, ela chama a função constrói para A, tornando-o em um heap. Após a construção do heap, necessariamente o elemento com a maior chave do vetor estará na primeira posição do vetor. Então, é chamada a função troca para A[0] e A[dir]. Depois disso o valor de dir é atualizado, e a função refaz é chamada novamente para A, do índice 0 até o índice dir-1, e assim por diante, até que todo vetor tenha sido devidamente ordenado.

3.9 Funções extras da classe InfoOrdenação

- **int get_comp():** retorna num_comp.
- **int get_copias():** retorna num_copias.
- **void reset():** zera as variáveis num_comp e num_copias.

3.10 Programa principal

O programa principal irá chamar, a depender dos parâmetros passados pelo terminal, algum dos métodos de ordenação citado na seção anterior. Ao executar o método, ele gera um arquivo de saída informando o número de comparções e cópias feitas, além do tempo de execução do método. Para isso, foram criadas as funções a seguir:

- **void menu_metodos():** imprime o menu de métodosde ordenação disponíveis, e o inteiro entre 1 a 7 ao qual correspondem.
- **void uso():** imprime o menu de usuário, explicando como passar corretamente os parâmetros junto ao executável.
- **void parse_args(int argc, char ** argv):** essa função usa como método auxiliar a função getopt da biblioteca getopt.h. Ela faz a leitura dos parâmetros passados pelo usuário, e guarda qual método de ordenação executar, o valor da semente a ser usada, além de armazenar também, caso necessário, os valores de k e m a serem usados no QuickSort Mediana(k) e QuickSort Seleção(m). Ela salva também o nome do arquivo de entrada e o de saída, onde os dados dos algoritmos serão escritos. Ao identificar algum parâmetro inválido, ou a ausência de parâmetros, chama a função uso para imprimir o menu e encerra o programa.
- **void escreve_info(std::ofstream * file, int tamanho, InfoOrdenacao info, double tempo_total):** esse método escreve informações no arquivo file sobre o algoritmo, usando os parâmetros de tempo (tempo_total) e de contagem armazenados em info.
- **double mede_tempo(struct rusage uso_antes, struct rusage uso_depois):** essa função obtém o tempo de usuário e de sistema de cada struct, e calcula a diferença entre elas, retornando assim o tempo de processamento total gasto pelo algoritmo.
- **int main(int argc, char ** argv):** o programa principal roda a função parse_args. Depois, ele faz a abertura do arquivo de entrada para sua leitura, usando um objeto do tipo ifstream, cria o arquivo de saída para escrita usando um objeto do tipo ofstream, e seta a semente usando a função srand. O main faz a leitura do arquivo de entrada, linha por linha, onde estão os tamanhos de vetor a serem avaliados. Para cada tamanho de vetor N lido, são gerados de forma sequencial, usando um for, 5 vetores com chaves aleatórias, determinadas pela função rand(). O método de ordenação é chamado para o vetor, usando uma variável

da classe InfoOrdenação, de forma que a quantidade de cópias e comparações feitas são nela armazenadas. O método mede_tempo também é chamado. Ao final, as informações sobre o algoritmo são escritas no arquivo de saída usando a função escreve_info. Para cada tamanho de N, a variável info e a variável tempo_total são zeradas. Por fim os arquivos de entrada e saída são fechados, e o programa é encerrado.

4 Análise de Complexidade

Nesta seção, será apresentada a análise de complexidade de tempo e espaço para os métodos apresentados na seção 2. O valor n aqui significa o tamanho do vetor.

Observação: Todas as funções construtoras, funções get e funções set do trabalho tem complexidade de tempo e espaço $O(1)$, pois realizam sempre o mesmo número de operações ao serem chamadas, e usam um número constante de variáveis auxiliares.

4.1 Classe InfoOrdenação

4.1.1 QuickSort Recursivo

De forma geral, o método de ordenação QuickSort tem a seguinte complexidade de tempo:

- **Pior caso:** $O(n^2)$. Ocorre quando o pivô é escolhido repetidamente da pior maneira possível: como um dos extremos do vetor.
- **Caso médio:** $O(n \log(n))$, pois de acordo com Sedgewick e Flajolet, o número médio de comparações $C(n)$ do algortimo é: $1,386n \log(n)$–$0,846n$.
- **Melhor caso:** $O(n \log(n))$. No melhor caso, o vetor sempre é dividido em dois subvetores de tamanho igual.

E agora, sobre a complexidade de espaço: o algoritmo de QuickSort tem um funcionamento "no local", termo traduzido do inglês "in-place". Isso quer dizer que o algoritmo mexe no vetor de entrada sem utilizar uma estrutura de dados auxiliar. Porém, um espaço extra de armazenamento é sim usado para as variáveis auxiliares do método(como na função ordena e na função partição). No QuickSort o vetor de entrada é sobrescrito ao longo que a função executa, por meio da troca da posição de seus elementos, que ocorre na função partição. Assim, contando o espaço extra de memória necessário para método, incluindo chamadas recursivas e variáveis auxiliares, a complexidade de espaço é $O(\log(n))$.

Observação: as otimizações aplicadas ao QuickSort não tem complexidade assintótica de tempo superior ao do QuickSort Recursivo. Algumas dessas otimizações tornam o algoritmo mais ágil como ficará mais claro na análise experimental deste trabalho, mas não superam a complexidade do QuickSort Recursivo.

4.1.2 QuickSort Mediana(k)

As otimização de escolher o pivô como a mediana de k elementos aleatórios do vetor, visa evitar o pior caso ($O(n^2)$) do QuickSort. Para obter a mediana é adicionado um custo a todas chamadas da função partição de ordenar um vetor de k elementos com o QuickSort Recursivo. Esse custo ficará entre k^2 e $k \log(k)$. As equações de recorrência para o algoritmo tem o seguinte formato:

No pior caso: $T(n) = n + k^2 + T(n-1)$. E no melhor caso: $T(n) = n + k \log(k) + T(n/2)$. Essas equações de recorrência resultam no seguinte:

Pior caso: $C(n) = \sum_{i=1}^{n} k^2 + \sum_{i=1}^{n} i = nk^2 + \frac{n(n+1)}{2} = O(n^2)$

Melhor caso: como $k \log(k)$ tem um valor constante, podemos aplicar o Teorema Mestre, e encontrar a mesma complexidade do melhor caso do QuickSort Recursivo: $C(n) = O(n \log(n))$.

Observação: estamos considerando que k assumirá valores pequenos em relação a n. Somente nesse caso ele não tem valor significativo, fazendo com que a complexidade de tempo do QuickSort Mediana permaneça a mesma do QuickSort Recursivo. Se k fosse próximo a n, a complexidade no pior caso seria $O(n^3)$. Ademais, seu custo de espaço é da mesma ordem que o método recursivo, para k's pequenos. Se k fosse um valor grande, próximo a n, o custo de espaço seria $O(k) \approx O(n)$.

4.1.3 QuickSort Seleção(m)

O método de seleção não é adaptável, e sempre tem o custo $O(n^2)$. Assim, ao usar esse método para partições de tamanho menor ou igual a m no QuickSort, estamos mudando a equação de recorrência da função tal que ela fique da seguinte forma:

No pior caso:

$T(n) = n + T(n-1)$, para $n > m$

$T(n) = n^2$, para $n \leq m$

Disso, encontramos o seguinte:

$C(n) = \sum_{i=1}^{m} i^2 + \sum_{i=m+1}^{n} i = \frac{m(m+1)(2m+1)}{6} + \frac{n(n+1)}{2} - \frac{m(m+1)}{2}$

Logo, $C(n) = O(n^2)$, para valores de m pequenos o suficiente. Se m tiver um valor próximo de n, o custo sobe e temos que $C(n) = O(n^3)$.

No melhor caso:

$T(n) = n + T(n/2)$, para $n > m$

$T(n) = n^2$, para $n \leq m$

Novamente, temos que o efeito na complexidade assintótica devido a mudança na equação de recorrência irá depender do valor da variável inserida m. Para valores de m pequenos, a complexidade assintótica do método também terá, eu seu melhor caso, ordem $O(n \log(n))$. Já se m assumir um valor próximo a n, essa complexidade sobe, e a fica $O(n^3)$.

4.2 QuickSort Não Recursivo e QuickSort Empilha Inteligente

Esses algoritmos terão a mesma complexidade de tempo do QuickSort Recursivo. Porém, quanto a complexidade de espaço essa funções se diferem do QuickSort Recursivo. Por mais que sua orgem de complexidade também seja $O(\log(n))$, essas funções gastam menos memória na prática por usar uma estrutura de dados do tipo Pilha, em relação a empilhar chamadas recursivas na stack.

4.3 HeapSort

A complexidade assintótica de tempo do Heapsort é sempre $O(n \log(n))$, qualquer que seja a entrada. Esse algortimo também é do tipo "in-place", e sua complexidade de espaço é $O(1)$.

4.4 MergeSort

A complexidade assintótica de tempo do MergeSort é sempre $O(n \log(n))$, qualquer que seja a entrada. Como esse método precisa de memória adicional proporcional ao tamanho do vetor de entrada, sua complexidade de espaço é $O(n)$.

4.5 Pilha

Aqui o valor n representa o tamanho da pilha.

- **bool vazia():** complexidade de tempo e espaço $O(1)$, pois realiza uma única operação, usando sempre o mesmo espaço na memória.
- **void empilha(Index index) e Index desempilha():** ambas funções tem complexidade de tempo e espaço $O(1)$, pois sempre realizam o mesmo número de operações, e usam um número de variáveis auxiliares constante ao alocar/desalocar a célula da pilha.
- **void limpa():** complexidade de tempo $O(n)$, pois chama a função desempilha n vezes para esvaziar a pilha.
- **~Pilha():** tem complexidade de tempo $O(n)$, pois chama a função limpa().

4.6 Main

Como o main chama um dos algortimos de ordenação acima citados, sua complexidade de tempo será a do método de ordenação usado. Suas funções auxiliares, como imprime_menu, mede_tempo e escreve_info tem custo constante de tempo e de espaço. Já parse_args é $O(n)$, sendo n aqui o número de parâmetros reconhecíveis passados pelo temrinal. Sendo assim, todas essas funções tem um custo insignificante perante aos métodos de ordenação. Por fim a complexidade de espaço do main é $O(n)$, pois ele aloca um vetor de tamanho n de acordo com a entrada fornecida.

5 Estratégias de Robustez

O programa feito depende de entradas do usuário para saber que comandos seguir. Logo, foi necessário pensar em que erros ele poderia cometer, e em como lidar com eles. Os erros possíveis são:

- usuário informar parâmetros inválidos, ou esquecer de passar algum parâmetro;
- o usuário não passar todos os parâmetros necessários para a execução do método de ordenação(por exemplo, solicitar a execução do quicksort mediana sem passar o valor de k, ou não informar o valor da semente);

Como estratégia de programação defensiva, foram estabelecidos parâmetros de verificação, que retornam mensagens de erro quando percebido algum dos erros acima, e que encerram a execução do programa. Assim, é garantida a execução do programa apenas quando parâmetros válidos são informados.

Essa verificação é feita usando objetos do tipo **erroAssert**, definido no header MsgAssert.h, criado pelo professor Wagner Meira. Esse objeto verifica uma condição, e caso ela seja falsa, ele imprime uma mensagem explicando o erro na tela e encerra o programa. Ele foi usado para verificar as seguintes condições:

- se os parâmetro m ou k informados são maiores que zero, e menores que o tamanho N do vetor,
- se o método de ordenação selecionado é válido,
- se um arquivo de entrada foi informado,
- se um nome para o arquivo de saída foi fornecido,
- se uma semente foi informada.

6 Análise experimental

6.1 Análise do QuickSort e suas variações

6.1.1 Parte 1

Nesta primeira parte da análise sobre o QuickSort. Serão abordadads as métricas de número de cópias, número de comparações e tempo de processamento de cada algoritmo.

Para analisar o QuickSort, cada método de ordenação foi testado em vetores de tamanho 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000. Cada algoritmo foi rodado 5 vezes, sobre vetores diferentes de tamanho N, tendo sido utilizado o valor 10 como semente de chaves aleatórias. Assim, os valores das tabelas a seguir são uma média de 5 execuções feitas de cada algoritmo, sobre cada tamanho de N.

A coleta de dados dos algoritmos executados foi feita de forma que os cinco vetores usados para cada tamanho N fossem iguais para todos algoritmos, permitindo aumentar a precisão da comparação entre os algoritmos.

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	13333	8603	0.000697
N=5000	82590	51375	0.0064012
N=10000	181305	109308	0.0153142
N=50000	1065696	628009	0.101868
N=100000	2272323	1325042	0.241712
N=500000	12776780	7433897	1.57446
N=1000000	26814228	15593359	3.28002

Tabela 1: QuickSort Recursivo

Como esperado, o número de comparações do QuickSort Recursivo está dentro dos limites do pior caso($O(n^2)$) e do melhor caso($O(\log(n))$. Além disso, como os veotres de entrada tinham chaves aleatórias, percebesse que a performance do algoritmo está bem próxima do caso médio ($O(\log(n))$), confirmando, como apresentado na literatura, que o QuickSort tem uma boa performance em entradas aleatórias.

Para analisar o QuickSort Mediana, o método foi testado nos mesmos vetores usando k=3, k=5 e k=7. Os resultados encontrados estão a seguir.

A otimização do QuickSort Mediana visa evitar o pior caso do QuickSort, selecionando de forma inteligente o pivô da partição. Pegando o pivô como a mediana de 3, para vetores de tamanho a

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	14347	14104	0.0016518
N=5000	85917	78389	0.0083274
N=10000	182274	164489	0.019331
N=50000	1056081	904348	0.126492
N=100000	2256972	1876960	0.286119
N=500000	12520376	10229598	1.68878
N=1000000	26250065	21168947	3.41317

Tabela 2: QuickSort Mediana, k=3

partir de N=10.000 que ele fez menos comparações que o QuickSort Recursivo. Porém, seu tempo de execução não superou o método recursivo em qualquer momento. Isso aconteceu porque um tempo extra é gasto na função partição_mediana em relação a partição do método recursivo, para selecionar k elementos aleatórios do vetor, gerar k cópias adicionais e ordenar esse vetor k. Assim, por mais que para valores de N grandes o número de comparações usando o QuickSort Mediana tenha sido menor, devido a escolha inteligente do pivô, o número de operações de cópias foi maior que no QuickSort Recursivo em todos os casos, e isso teve um custo no tempo de execução do algoritmo que o impediu de superar o QuickSort Recursivo.

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	15427	15325	0.0013298
N=5000	89922	84924	0.0088906
N=10000	190730	177142	0.0199236
N=50000	1086189	970247	0.126239
N=100000	2281397	2012729	0.27107
N=500000	12718200	10906860	1.60711
N=1000000	26547153	22539661	3.40618

Tabela 3: QuickSort Mediana, k=5

Para k=5, o número de comparações foi menor que o do QuickSort tradicional somente para N maior ou igual a 500.000. O tempo de execução superou o do QuickSort Recusivo não foi superado em qualquer caso.

Porém, podemos ver que k=5 superou k=3 em tempo de processamento para N maior ou igual a 100.000, mesmo tendo feito nesses casos um número de cópias e comparações maior que para k=3. Isso se explica pelo fato de que para k=5 o número de chamadas da função partição deve ter sido provavelmente menor do que para k=3, pois ao escolher o pivô como a mdeiana de cinco, é evitado de forma ainda mais eficiente o pior caso do QuickSort.

Para k=7 o QuickSort Mediana teve seu pior desempenho. Em relação a k=5, não houve qualquer ganho no tempo de processamento, e para $N \geq 500.000$, ele demorou ainda mais. Ou seja, para k=7 vemos que o ganho de termos menos chamadas da função partição e menos comparações das chaves é contrabalanceado pelas operações de cópia e comparação necessárias para a obtenção

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	16199	16035	0.001225
N=5000	93476	87895	0.0094434
N=10000	197534	183317	0.0191686
N=50000	1110480	1001681	0.127188
N=100000	2349489	2073490	0.277975
N=500000	12948635	11225673	1.65008
N=1000000	26906691	23197224	3.52108

Tabela 4: QuickSort Mediana, k=7

da mediana. Conlui-se que essa otimização do QuickSort teve seu melhor desempenho para k=5, e não superou o QuickSort Recursivo de forma geral, mas somente em alguns casos específicos.

Agora, vamos analisar os resultados do QuickSort Seleção. O método foi testado para m=10 e m=100. Para m=10, o desempenho do QuickSort Seleção foi bem similar ao QuickSort Recursivo. Não havendo ganhos significativos na velocidade de execução da ordenação.

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	13015	8209	0.0006552
N=5000	81032	49401	0.0065292
N=10000	178529	105090	0.020084
N=50000	1050598	607476	0.110974
N=100000	2241800	1284370	0.242933
N=500000	12625485	7231137	1.53968
N=1000000	26513287	15186858	3.29281

Tabela 5: QuickSort Seleção, m=10

Ao observar as colunas da quantidade de cópias e comparações, vemos que essas métricas estão bem similares as do QuickSort Recursivo, para todos tamanhos de vetor. Isso justifica o fato de ambos terem mostrado tempos similares de execução.

Já para m=100, houve ganhos notáveis no tempo de processamento para N=1000, N=500.000 e N=1.000.000 em relação ao QuickSort Recursivo, sendo que para os outros tamanhos de vetor o QuickSort Seleção teve um desempenho análogo ao do recursivo. isso acontece porque o método de seleção faz menos movimentações que o QuickSort. Se compararmos a tabela 6 com a tabela 1, podemos ver que o QuickSort Seleção fez menos cópias para todos tamanhos de vetor. A operação de cópia é mais custosa em termos de tempo que a de comparação de chaves, pois a cópia envolve a movimentação de Registros inteiros do vetor. Assim, por mais que o QuickSort Seleção faça muito mais comparações que o Recursivo, seu desempenho no geral foi o melhor até aqui em termos de tempo de execução.

Agora, vamos abordar os resultados da análise do QuickSort Não Recursivo. Como esperado, se compararmos a tabela 1 com a tabela 7, vemos que o número de comparações e cópias dos algoritmos

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	37203	5833	0.0005376
N=5000	208622	37231	0.0064984
N=10000	431214	81064	0.019796
N=50000	2322694	486542	0.104402
N=100000	4812539	1042190	0.231062
N=500000	25410988	6021571	1.47297
N=1000000	51990053	12772681	3.22378

Tabela 6: QuickSort Seleção, m=100

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	13333	8603	0.000606
N=5000	82590	51375	0.0065022
N=10000	181305	109308	0.0209988
N=50000	1065696	628009	0.113617
N=100000	2272323	1325042	0.258707
N=500000	12776780	7433897	1.63458
N=1000000	26814228	15593359	3.31327

Tabela 7: QuickSort Não Recursivo

foi igual. Esse resultado era aguardado visto que o QuickSort Recursivo e Não Recursivo não se diferem na forma de ordenar o vetor. Ele se diferem somente quanto ao uso da stack. Enquanto o método recursivo empilha várias chamadas de função na stack, o método não recursivo não ocupa esse espaço, fazendo uso de uma Pilha alocada dinamicamente para controlar as partições.

O QuickSort Empilha Inteligente é uma versão otimizada do QuickSort Não Recursivo. A diferença entre eles é que o primeiro método sempre processa antes a menor partição, enquanto o segundo empilha sistematicamente o lado esquerdo do vetor, e procesa o lado direito, sem verificar o tamanho dos lados. Isso justifica o porquê do Empilha Inteligente ter um melhor desempenho que o Não Recursivo: na tabela 8 vemos que ele ganhou em tempo de execução para todos os tamanhos de vetor.

É importante apontar que, embora ambos algoritmos façam uso da não recursividade, essa otimização só mostrou efeito relevante quando usada de forma mais astuta no método empilha inteligente.

O desempenho do Empilha Inteligente se destaca também em relação ao QuickSort Recursivo, o qual superou no tempo de execução. Entre o QuickSort Seleção usando m=100, e o Empilha Inteligente, há pouca diferença em termos de desempenho. Ambos mostram uma vantagem muito similar sobre o QuickSort Recursivo.

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	13333	8603	0.000586
N=5000	82590	51375	0.0057018
N=10000	181305	109308	0.018079
N=50000	1065696	628009	0.109897
N=100000	2272323	1325042	0.238506
N=500000	12776780	7433897	1.4714
N=1000000	26814228	15593359	3.20137

Tabela 8: QuickSort Empilha Inteligente

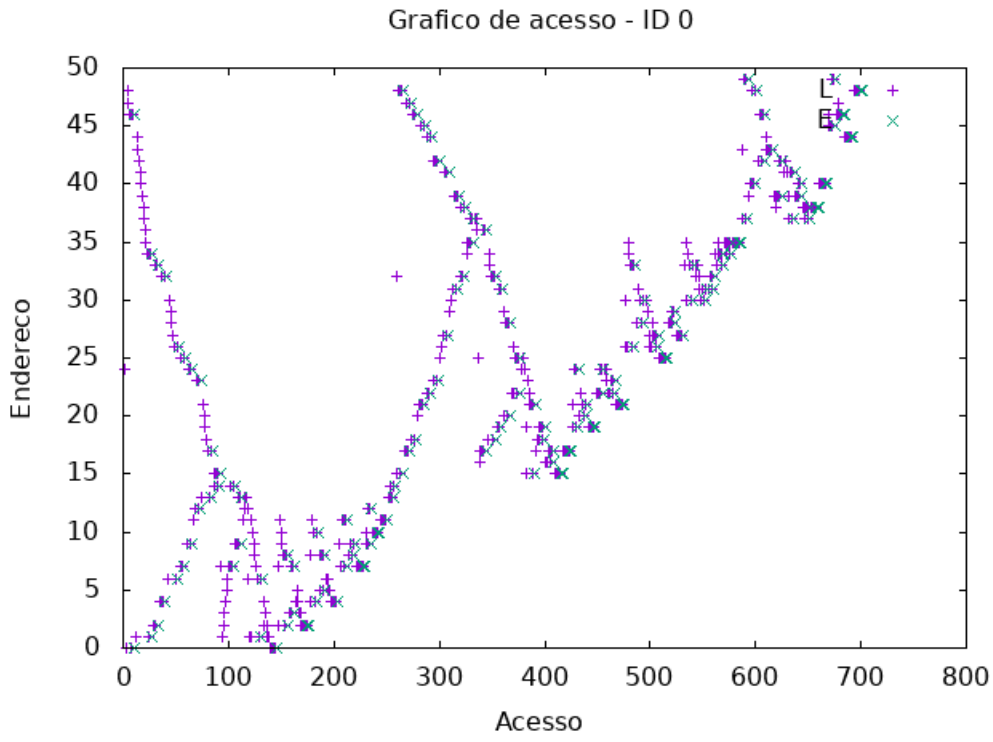
6.1.2 Parte 2 - Localidade de Referência

Para a análise de localidade de referência foram usadas as funções de registro de acesso e leitura e escritas na memória da biblioteca Memlog.

Cada algoritmo foi testado para um vetor de 50 elementos, exceto o QuickSort Mediana, que foi testado com um vetor de tamanho 30. Como o QuickSort Mediana pega valores aleatórios do vetor, ou seja, de forma não sequencial, para vetores maiores que 30 o range de acesso de memória ficava muito grande, o que impossibilitava o uso do AnalisaMem.

Por fim, a partir dos relatórios gerados das execuções, usando o Analisamem e o GnuPlot, foram produzidos os gráficos abaixo.

1.QuickSort Recursivo

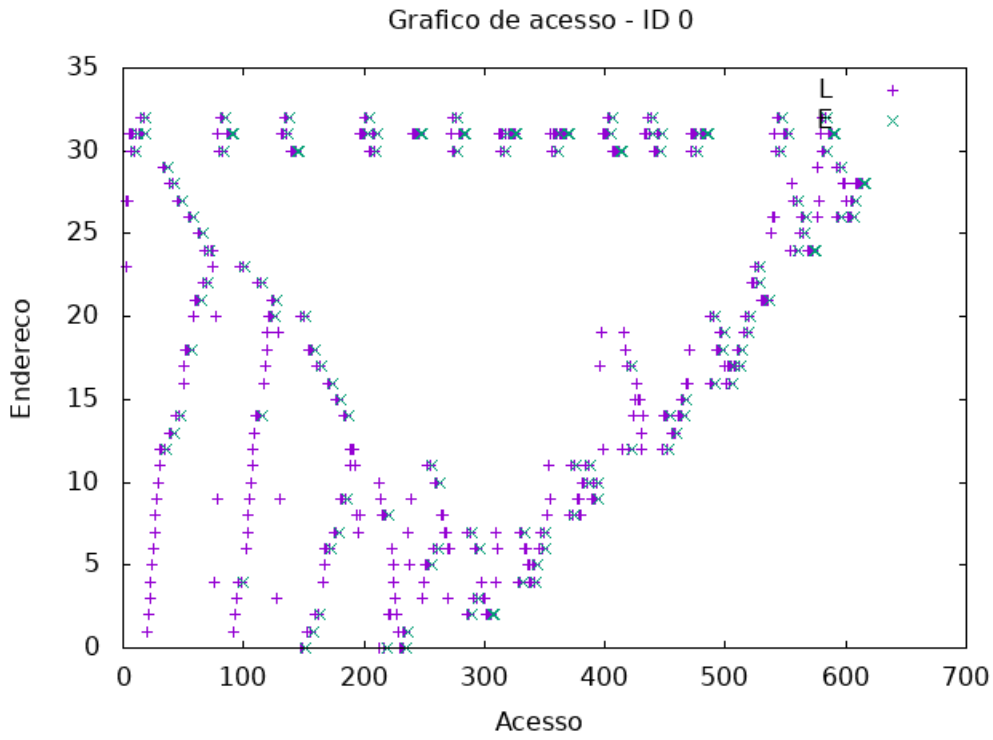


Neste primeiro gráfico vê-se que bem no meio do eixo y está o primeiro acesso a memória, que ocorre quando atribuímos ao pivô a mediana do vetor. Depois disso vemos acessos sequenciais ocorrendo em paralelo: uma sequência que nasce de cima, do endereço 49 e outra que sai de baixo, do endereço 0. Elas representam, respectivamente, o percurso do índice j e do i no método de partição. Próximo ao acesso 100, perto do endereço 14 é quando os índices se cruzam.

A partir daí, passar a ser ordenado o lado esquerdo da partição, ou sejam do endereço 0 ao 14. Por isso as leituras e escritas, até em torno do acesso 270, se mantém na parte debaixo do gráfico. A partir desse momento, toda partição esquerda foi ordenada, e o lado direito que passará a ser ordenado. Vemos uma leitura separada das outras, no endereço 33: novamente, temos a leitura do pivô sendo feita, agora para processar o lado direito.

Esse padrão se repete por todo o gráfico. A cada vez que o índice i e j se cruzam, a partição mais a esquerda é processada,até que a última partição de todas, a mais a direita, ao final do vetor, é processada, e a ordenação está completa.

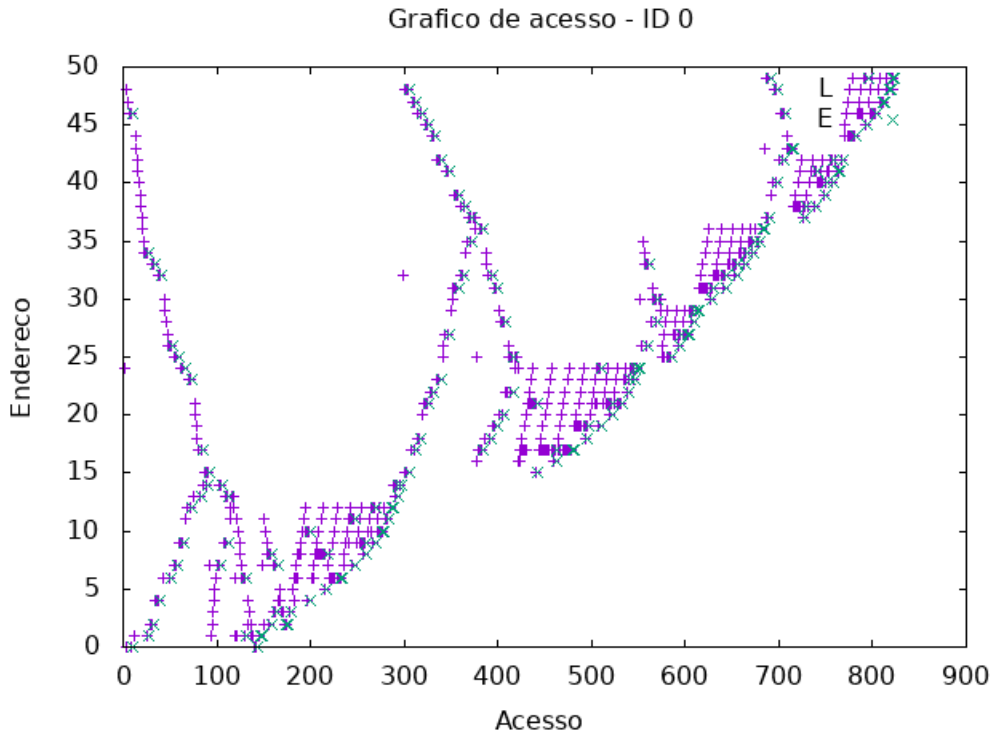
2.QuickSort Mediana, k=3



O gráfico do QuickSort Mediana apresenta um padrão de acesso similar ao QuickSort Recursivo. Ele se difere entrando em dois pontos. Em vez de termos um acesso único para definir o pivô, são feitas três leituras em posições aleatórias. Por isso no acesso 0 o eixo y está marcado com leituras mais de uma vez. Isso se repete por todo gráfico: assim que uma nova partição vai ser processada, temos registros de acesso em posições aleatórias, como perto do acesso 90, ou do acesso 120, por exemplo.

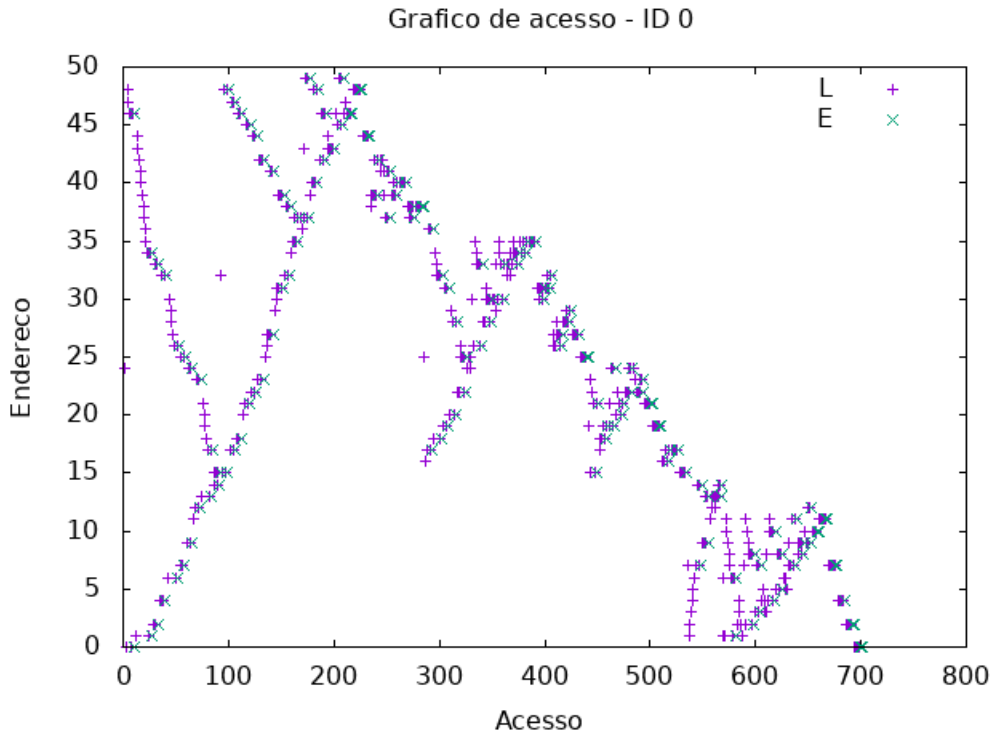
Além disso, vemos na parte superior do gráfico, nos endereços acima de 30, um padrão repetido de acessos e escritas em 3. Eles representam os acessos e leituras na construção do vetor de tamanho k e na sua ordenação, lembrando que esse vetor extra é usado para obter a mediana.

3.QuickSort Seleção, m=10



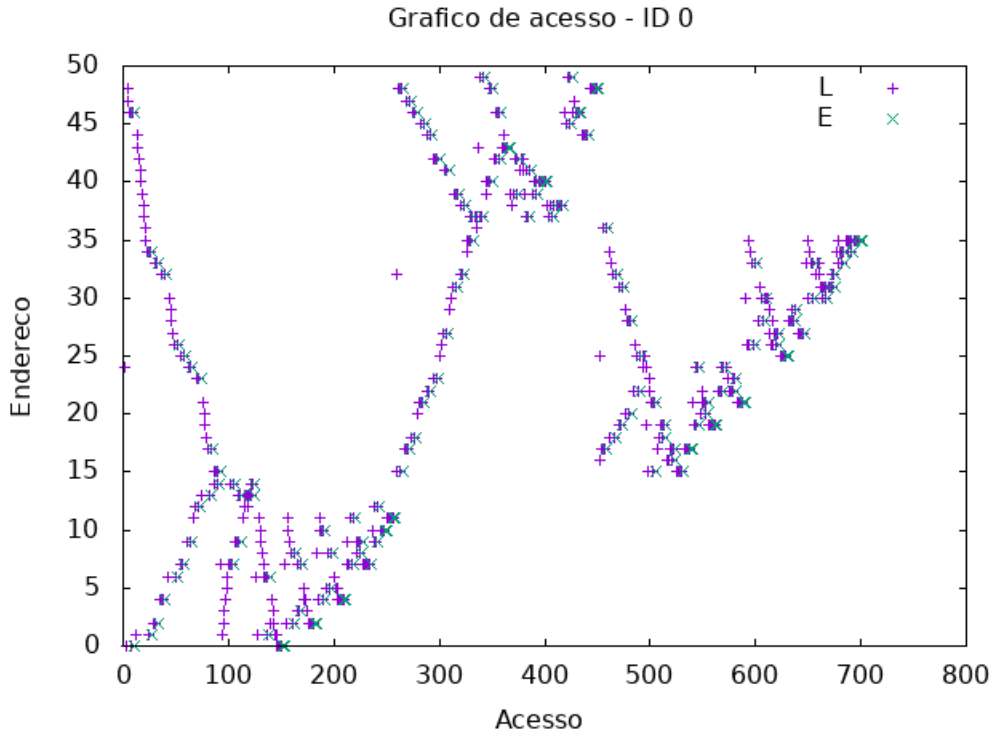
Agora, vamos comentar os padrões no gráfico do QuickSort Seleção. Podemos ver claramente quando a seleção ocorre, perto do acesso 170: ao encontrar uma partição com tamanho menor que m, vemos uma sequência de m-1 acessos acontecendo, do menor para o maior endereço da partição, depois m-2 acessos, saindo do segundo menor endereço até o maior, e assim por diante, até que a partição está toda ordenada. Percebe-se que, a cada vez que a função de seleção percorre o vetor, temos apenas dois registros de escrita, indicando a troca de posição do n-ésimo menor elemento com a n-ésima posição.

4. QuickSort Não Recursivo



Acima, está o gráfico da implementação não recursiva. Vemos que as partições são processadas de forma oposta aos gráficos anteriores: primeiro a partição mais a direita é processada, para que depois a esquerda possa ser percorrida. Por isso este gráfico tem seu último acesso no menor endereço do vetor.

5.QuickSort Empilha Inteligente



Mostraremos agora o funcionamento do QuickSort Empilha Inteligente. O gráfico com seus acessos expõe, com clareza seu mecanismo de processar sempre primeiro a menor partição. O algortimo aqui está ordenando o mesmo vetor que o QuickSort Recursivo processou, porém mostra acessos em uma ordem bem diferente. Se compararmos os dois gráficos, vemos um comportamento similar até o acesso 330. A partir daí o QuickSort Empilha Inteligente se difere: enquanto no algoritmo recursivo vemos uma sequência de acessos no lado mais à esquerda do vetor, que é o maior, aqui temos acessos no lado direito, por ser a menor partição. A forma de ordenação fica diferente até o final, tanto que o algoritmo termina de processar o vetor no endereço 35.

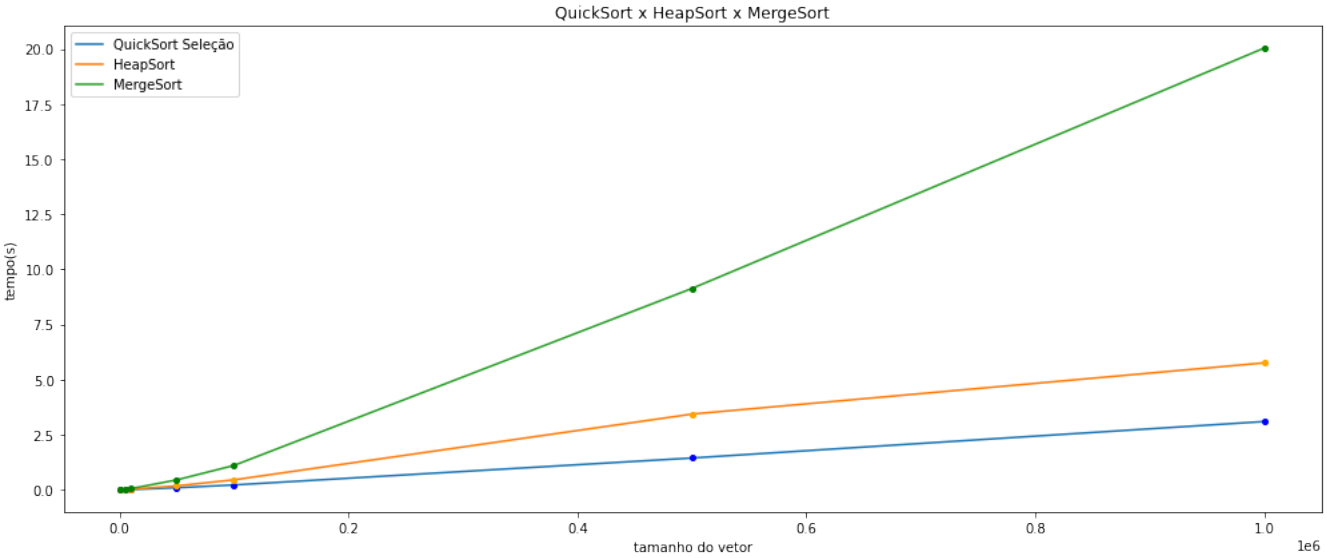
6.2 Parte 2 - QuickSort Seleção x HeapSort x MergeSort

Para a segunda parte da avaliação experimental desse trabalho, fiz a escolha arbitrária de usar o QuickSort Seleção, com m=100, como versão do QuickSort para comparar com o HeapSort e o MergeSort.

Foram coletados dados para essa análise da seguinte maneira:

- Cada algortimo foi testado com vetores de tamanho 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000.
- Cada algoritmo foi testado para cada tamanho com cinco vetores diferentes.
- Para gerar as chaves dos registros, foram usadas as seguintes sementes: 2, 10, 33, 89, 104. Ou seja, cada um dos cinco vetores de tamanho N teve suas chaves geradas com uma semente diferente.
- Por fim, foi feita uma média desses valores, que foram plotados em gráficos usando as bibliotecs numpy e matplotlib da linguagem Python.

Ao final dessa documentação, na seção extra apêndice, encontra-se uma tabela com todos os dados colhidos para análise. A partir deles foram gerados os gráficos que serão explicados a seguir.



Observação: os pontos no gráfico representam os tempos medidos.

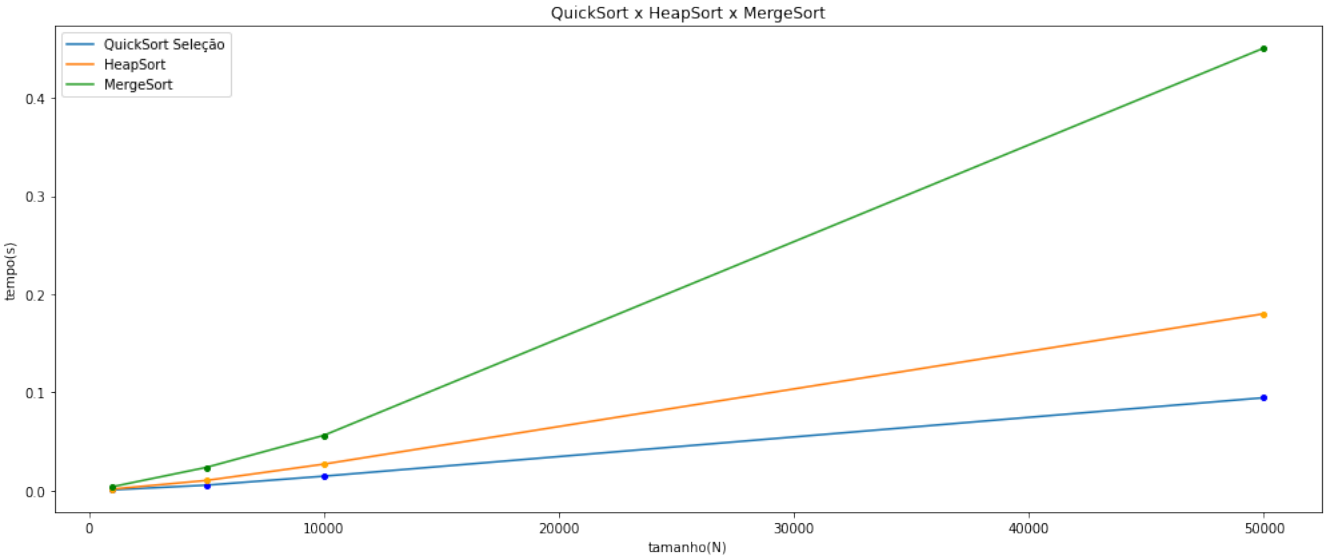
Esse primeiro gráfico mostra o tempo médio de execução de cada método, de acordo com o tamanho de cada vetor. Claramente, o QuickSort Seleção é o algortimo com o melhor desempenho, seguido do HeapSort e por fim do MergeSort.

É sabido que teoricamente os três algoritmos tem uma complexidade assintótica de tempo similar: no seu melhor caso, o QuickSort tem um desempenho de ordem $O(n \log(n))$, e o HeapSort e o MergeSort sempre tem um desempenho de ordem $O(n \log(n))$, independente de qual seja a entrada. O que explica então, que o QuickSort tenha mostrado a melhor velocidade dos três?

Podemos atribuir isso a vários fatores. O primeiro deles sendo o tipo de operação que cada algoritmo realiza. O HeapSort e o MergeSort fazem muito mais operações de cópia que o QuickSort. A operação de atribuir a um espaço de Registro na memória um novo valor, implica em atualizar não só a chave de cada Registro, como atualizar todos as quinze strings e os dez floats associados a ele. Trabalhando com vetores alocados dinamicamente ou seja, que estão alocados no heap, isso demora mais ainda.

O MergeSort em especial, é o que faz mais cópias. Isso porque ele precisar criar vetores novos para fazer a ordenação, e para vetores grandes, isso representa um custo alto de tempo. Dessa forma, esse método se mostra não só custoso espacialmente, por precisar de memória auxiliar proporcional a N, como temporalmente o menos eficiente. Dentre os três métodos ele é o que faz menos comparações de chaves, mas essa economia de comparações é contrabalanceada pelo grande número de movimentações. Para N=1.000.000 essa diferença fica brutal: o MergeSort levou 20 segundos para ordenar o vetor, já o HeapSort precisou de 5,8 segundos e o QuickSort Seleção somente 3 segundos.

Abaixo temos um gráfico somente dos vetores menores, de $N \leq 50000$, para mostrar mais detalhadamente a diferença no tempo de processamento nas execuções iniciais.



7 Conclusão

Concluo esse trabalho tendo tirado um grande aprendizado sobre os métodos de ordenação efieci-entes. Poder implementar, testar e debugar cada um deles foi um processo que exigiu bastante mão na massa e esforço, para um entendimento do algorimo para além do teórico.

Além disso, ter a opornidade de fazer uma tarefa em que o foco estava tão forte na análise experimental, quanto no código, foi muito estimulante e desafiador para mim. Gostei muito de poder comparar e analisar as métricas de cada algotimo, e aprender que entre teoria e prática, há uma grande distância a ser considerada. Me surpreendi com as otimizações do QuickSort. Não esperava que elas pudessem superar o algoritmo tradicional apresentado em sala. Isso me ensinou que sempre há espaço para tentar melhorar algo que já está estabelecido. Ou que é possível otimizar um algoritmo famoso para as demandas específicas que uma situação pedir.

Com certeza esse foi o trabalho mais díficil e detalhado que fiz até o momento no curso, porém também o que mais me fez aprender.

8 Bibliografia

- Material didático usado em sala(slides).
- https://www.cs.princeton.edu/courses/archive/fall12/cos226/lectures/23Quicksort.pdf
- https://cplusplus.com/doc/tutorial/arrays/

9 Instruções de Compilação e execução

Para compilar o programa, deve-se seguir os seguintes passos:

- Acessar o diretório TP;
- Utilizando um terminal, digitar make, de forma a gerar o arquivo executável **run.out**;
- Na pasta bin, chamar o executável ./run.out, junto aos parâmetros de execução.
- No caso de dúvida, deve-se rodar o executável sozinho, de forma a imprimir o menu de uso do programa na tela.

10 Apêndice

Na tabela abaixo estão as métricas obtidas para cada tamanho de vetor. Os dados estão na seguinte ordem por algortimo: QuickSort Seleção, HeapSort e MergeSort.

Tamanho do vetor	Comparações	Cópias	Tempo de execução
N=1000	42177	5789	0.0008516
N=1000	20909	14061	0.0016418
N=1000	8703	19952	0.004014
N=5000	224437	36822	0.005589
N=5000	133651	82096	0.0103962
N=5000	55236	123616	0.0237124
N=10000	454251	81570	0.0147666
N=10000	292282	174181	0.0270356
N=10000	120437	267232	0.0563588
N=50000	2446184	487744	0.0946812
N=50000	1752788	987436	0.180221
N=50000	718202	1568928	0.450895
N=100000	5153724	1035348	0.221888
N=100000	3755599	2075128	0.454004
N=100000	1536346	3337856	1.10647
N=500000	26781933	6035731	1.44439
N=500000	21673728	11524355	3.43744
N=500000	8837546	18951424	9.13655
N=1000000	55.273.237	12.694.226	3.09971
N=1000000	45846072	24.048.278	5.76244
N=1000000	18674532	39902848	20.0473

Tabela 9: QuickSort Seleção, MergeSort e HeapSort