Trabalho Prático 1 - Ordenador Universal

Nome: Miguel Bertussi Carneiro Moreira
Matrícula: 2024005483

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais

(UFMG) Belo Horizonte - MG - Brazil

1. Introdução

A empresa Zambs têm o objetivo de lançar uma estrutura de dados que vai revolucionar o mercado, o TAD Ordenador Universal, capaz de selecionar o algoritmo de ordenação ideal para quaisquer que sejam as características do vetor a ser ordenado, determinando durante a execução as condições ideais para cada um dos *sorts* a serem feitos. O objetivo é determinar com precisão estes limiares de determinação de seleção para minimizar os custos.

O problema envolve claramente múltiplas iterações e sub-iterações para atingir seu ideal, além dos *arrays* para armazenar os dados. Por esta razão, um vetor dinâmico chamado **vector** será implementado, permitindo um uso mais controlado da memória, além de ser expansível sobre demanda. Além disso, os diversos métodos auxiliares implementados serão de suma importância para ajudar com a clareza do programa.

Iremos utilizar dos algoritmos ***Quick Sort*** e ***Insertion Sort*** diversas vezes para achar os limites de quebras e de partições, reduzindo cada vez mais a faixa de busca iterativamente. Também faremos uso de uma função ***shuffleVector***, que será importante para que possamos reutilizar os *sorts* mantendo números de quebras estáveis.

A coordenação entre estes diversos loops e estruturas de dados vão permitir que os limiares ideias sejam encontrados pela empresa e o algoritmo alcance seu projeto de ordenar qualquer tipo de estrutura que nele seja utilizado.

2. Implementação

O código foi inteiramente projetado e implementado na linguagem C++, com exceção do uso da função ***fprintf*** do C ao invés de ***cout***, visando principalmente a modularização através das classes e o aspecto multi-uso do **template**. Seguindo esta linha do algoritmo versátil para diferentes tipos de dados, grande parte das funções foram implementadas fazendo o uso dos **templates**, portanto, a maioria dos arquivos foi declarada e implementada no próprio arquivo .hpp. Nesta seção, descreveremos as classes e seus respectivos métodos.

2.1. Estatísticas

A classe **estatisticas** é a responsável por registrar e calcular as estatísticas necessárias durante a execução do programa, possuindo como atributos informações de entrada da **main** (**a**, **b**, **c**) e informações a respeito da ordenação dos vetores (**comparacoes, movimentacoes, chamadas**), além do construtor e dos métodos getters, incremento, *reset* e cálculo. Como não depende diretamente do tipo de dado a ser ordenado, pôde ser dividida entre .cpp e .hpp.

Atributos

- **int comparacoes**, **movimentacoes**, **chamadas**.
- **double a**, **b**, **c**.

Construtor

- **estatisticas(double _a, double _b, double _c)**: Inicializa **a**, **b** e **c** com os valores **_a**, **_b** e **_c,** e **comparacoes**, **movimentacoes** e **chamadas** com zero.

Métodos

- **int getComparacoes()**: retorna o valor de **comparacoes**.
- **int getMovimentacoes()**: retorna o valor de **movimentacoes**.
- **int getChamadas()**: retorna o valor de **chamadas**.
- **void resetEstatisticas()**: zera o valor de **comparacoes**, **movimentacoes** e **chamadas**.
- **double calcularCusto()**: retorna o valor do custo, calculado usando **a*comparacoes** + **b*movimentacoes** + **c*chamadas**.
- **void incComparacoes()**: incrementa **comparacoes** em um.
- **void incComparacoes(int num)**: incrementa **comparacoes** em **num**.
- **void incMovimentacoes()**: incrementa **movimentacoes** em um.
- **void incMovimentacoes(int num)**: incrementa **movimentacoes** em **num**.
- **void incCalls()**: incrementa **chamadas** em um.
- **void incCalls(int num)**: incrementa **chamadas** em **num**.

Esta classe será invocada em quase todas as demais, sem ela, o acesso aos dados armazenados seria muito mais complexo, e os métodos aqui implementados facilitarão muito o uso destes dados para resolver o problema.

2.2 Vector

A classe **vector** é uma implementação própria de um vetor dinâmico genérico, que pode armazenar elementos de qualquer tipo **T**. Ela é semelhante a **std::vector**, permitindo inserção dinâmica de elementos, acesso por índice e cópia profunda entre vetores. A classe foi inteiramente implementada em um arquivo .hpp por ser do tipo **template**.

Atributos

- **T* dados**: *array* simples do tipo **T**.
- **int _capacidade**: capacidade atual alocada do vetor.
- **int _tamanho**: quantidade atual de elementos armazenados no vetor.

Construtores e Destrutor

- **vector():** inicializa o **vector** com **_capacidade** igual a um e **_tamanho** zero.
- **vector(int capacidade)**: inicializa o **vector** com a **capacidade** desejada e **_tamanho** zero.
- **vector(const vector& outro)**: construtor de cópia que aloca nova memória e copia todos os elementos do **vector** original.
- **~vector()**: destrutor responsável por desalocar a memória utilizada pelo **vector**.

Métodos

- **void push_back(const T& elemento):** insere elemento no final do **vector**. Caso o **vetor** esteja cheio, **expande** sua **_capacidade** antes da inserção.

- **void limpar():** reseta o **_tamanho** do **vector** para 0, mantendo a **_capacidade**. Os elementos são perdidos, mas a memória permanece alocada.
- **int tamanho() const:** retorna quanto elementos estão no **vector**, **_tamanho**.
- **int capacidade() const:** retorna a capacidade atual alocada do **vector**, **_capacidade**.
- **void expandir():** dobra a **_capacidade** do **vetor** alocando nova memória, copiando os elementos antigos e liberando a memória anterior.

Operadores Sobrecarregados

- **T& operator[](int index)**: retorna uma referência ao elemento da posição **index**, permite leitura e escrita.
- **const T& operator[](int index) const**: versão constante do operador acima, permite apenas leitura.
- **vector<T>& operator=(const vector<T>& outro)**: realiza cópia profunda dos elementos do **vector outro**, substituindo os dados do **vector** atual.

Esta classe será muito utilizada a seguir, e possui uma grande flexibilidade de usos, graças à implementação do **template**. Estes métodos que foram implementados permitem acesso, criação e modificação eficiente aos elementos armazenados.

2.3 Sorts

O arquivo **sorts.hpp** guarda apenas a implementação e definição dos algoritmos ***Quick Sort***, ***Insertion Sort***, a função ***shuffleVector*** e as implementações extras que eles precisam. Este arquivo herda **estatisticas.hpp** para registrar as estatísticas feitas dentro de cada *sort*, que recebem por referência uma instância de **estatisticas,** e de **vector.hpp**, já que iremos ordenar **vector's** . Todos os cabeçalhos das funções estão utilizando **template**, pois recebem um **vector** que pode ser de tipo variável.

Funções

- **void swap(T& a, T&b, estatisticas& estat)**: troca o valor das variáveis **a** e **b** e incrementa em três as **comparacoes**.
- **T mediana(T& a, T&b, T&c)**: recebe três valores **a**, **b** e **c** e retorna a mediana deles.
- **void insertionSort(vector<T>& vetor, int esquerda, int direita, estatisticas& estat)**: realiza o ***InsertionSort*** no **vetor** recebido, da posição **esquerda** até a posição **direita**, registrando as estatísticas em **estat**.
- **void partition(vector<T>& vetor, int esquerda, int direita, int* i, int *j, estatisticas& estat)**: cria a partição para o ***Quick Sort***, utilizando o pivô na posição da mediana para evitar o pior caso, e ordena os valores utilizando o swap, além de incrementar as estatísticas.
- **void quickSort(vector<T& vetor, int esquerda, int direita, estatisticas& estat, int minTamParticao)**: faz a chamada do **partition** e confere os valores obtidos, em conjunto com o **minTamParticao**, para fazer as chamadas recursivas para parte da **esquerda** do pivô e da **direita**. Caso o sub-vetor a ser ordenado seja pequeno, o ***insertionSort*** é chamado.

- **int shuffleVector(vector<T>& vetor, int numShuffle, int seed)**: induz **numShuffle** quebras no vetor usando uma chave pseudoaleatória obtida através da função **srand48(seed)**.

Com este arquivo, seremos capazes de analisar os diferentes custos dos *sorts* para determinar com precisão os limiares que são desejados. A implementação eficiente dos *sorts* também contribui para a amortização da complexidade, e.g. o ***quickSort*** inteligente, que sabe quando chamar o ***insertionSort***.

2.4 Ordenador Universal

Este TAD é o principal de todo o trabalho, é o responsável por implementar as funções solução do problema e, por ser o principal, herda **estatisticas.hpp**, **vector.hpp** e **sorts.hpp** (os dois primeiros indiretamente, pois estão inclusos em **sorts.hpp**). Os métodos aqui implementados foram parcialmente baseados naqueles apresentados no enunciado do trabalho, mantendo a mesma lógica mas com algumas atualizações. O TAD foi implementado como uma classe de métodos públicos, para facilitar a instanciação deste usando outros tipos de dados, o que se mostrou eficiente, portanto, não há a necessidade de construtores e destrutores, apenas os métodos principais.

Métodos

- **void ordenadorUniversal(vector<T>& vetor, int minTamParticao, estatisticas& estat)**: realiza a ordenação do **vector** com base no **minTamParticao**(**limiar de Partição**), se **vetor.tamanho()** > **minTamParticao**, usa ***QuickSort***, caso contrário, usa ***InsertionSort***.
- **int determinaLimiarParticao(vector<T>& vetor, double limiarCusto, estatisticas& estat)**: calcula o valor ideal de **minTamParticao**, ajustando iterativamente o custo, por meio do **ordenadorUniversal,** até que ele convirja para o **limiarCusto,** ou que o número de faixas de partição distintas seja maior ou igual a cinco. Para fazer este calibramento, a função **calculaNovaFaixa** também é chamada iterativamente até que a condição citada anteriormente seja satisfeita. O custo de cada iteração é armazenado em um **vector**, e ao fim das sub-iterações, procuramos o índice do menor custo neste para mandar para **calculaNovaFaixa**. Assim, o processo vai se repetindo.
- **int determinaLimiarQuebras(vector<T>& vetor, double limiarCusto, estatisticas& estat, int minTamParticao, int seed)**: faz o cálculo do limiar de quebras, **limQuebras**, que originalmente era usado em **ordenadorUniversal** para decidir qual é o melhor *sort*, mas seu uso acabou não sendo importante durante a execução do problema. Este método é similar ao **determinaLimiarParticao**, vai chamando o ***quickSort*** e o ***insertionSort*** iterativamente, calibrando o **limQuebras** e usando a função **calculaNovaFaixa** para ir reduzindo a faixa de busca das partições. Também procuraremos o índice do **vector** de custos onde a diferença entre o custo do ***quickSort*** e ***insertionSort*** é mínima, e mandaremos para **calculaNovaFaixa**.

- **void calculaNovaFaixa(vector<int>& vetorX, vector<double>& custos, int limIndex, int& minX, int& maxX, int& passoX, int numX, float& diffCusto)**: é uma função importantíssima para o calibramento dos valores, porque pega o índice desejado no vetor de custos, que será a posição do elemento de menor valor, e estabelece os novos **upper** e **lower bound** de partições para refazer o processo de busca. Este método é chamado iterativamente e, como seus parâmetros são passados por referência, não precisa retornar nenhum valor. Além disso, a função pode ser reutilizada em **determinaLimiarParticao** e **determinaLimiarQuebras**, porque os processos são feitos separadamente, então não há conflitos de ponteiros. Além disso, tendo em seu controle as novas posições, ela calcula o **diffCusto**, que será necessário no **print**.
- **void imprimeEstatisticas(std::string sort, int t, vector<double>& custos, int numX, estatisticas& estat)**: é um método que é utilizado em ambas funções principais, **determinaLimiarParticao** e **determinaLimiarQuebras**, recebe uma ***string*** como parâmetro para poder diferenciar qual das duas funções está sendo executada. Seu objetivo principal é facilitar os **prints** que são necessários.

Este TAD resolve o problema de forma eficiente, reutilizando o máximo de estruturas possível, além de ser aplicável a qualquer tipo de estrutura de dados necessária. Os tipos padrão como **int**, **char**, **double** etc. são todos funcionais sem nenhuma alteração além da instanciação na ***main,*** já tipos mais elaborados como **Structs** ou **Classes** precisam possuir a sobrecarga dos operadores de comparação (<, >, ==, !=) em suas definições para serem funcionais, caso contrário, o compilador não saberá como comparar duas instâncias do tipo de dado em questão.

3. Análise de Complexidade

Nesta seção, a complexidade de cada algoritmo será definida de forma precisa.

3.1 Algoritmos de Estatísticas

Os algoritmos implementados em **estatisticas.hpp** estão restritos a atribuições uni valoradas, operações de soma e de multiplicação e de chamadas de atributos, também não possuímos nenhum tipo de memória extra alocada nestes processos, portanto, a complexidade de tempo e de espaço são iguais, constantes, então temos O(1).

3.2 Vector

Aqui, temos métodos mais custosos, que alocam memória dinamicamente, passam elementos de um **vector** ao outro etc. A complexidade dos métodos: **vector()**, **vector(const vector& outro)**, **expandir()**, **operator=(const vector<T>& outro)**, será sempre O(n) tanto de espaço, quanto de tempo, porque devem alocar ‘n’ espaços na memória (espaço) e atribuir ‘n’ elementos à memória (tempo). As demais operações são sempre O(1), tanto de tempo, quanto de espaço, pois retornam valores já armazenados e não precisam alocar memória. Já o **push_back()**, tem o pior caso com O(n), tanto de complexidade como de espaço, porque neste caso ele deve chamar o **expandir()**, mas como o crescimento da capacidade é

exponencial, este custo é amortizado e é, em média, O(1), pois a memória já estará alocada e apenas insere-se o elemento ao fim do vetor, o que é custo constante.

3.3 Sorts

Aqui, os algoritmos apresentados já têm seu custo conhecido, o ***insertionSort*** é O(1) em espaço, é completamente *in-place*, mas têm complexidade temporal O(n) no melhor caso (vetor ordenado) e O(n^2) no pior caso (vetor inversamente ordenado). Portanto, o caso médio é O(n^2). Já o ***quickSort*** não é *in-place*, o caso médio espacial é O(log(n)), pois vamos dividindo o vetor em dois pedaços, as chamadas recursivas são empilhadas e custosas e, no pior caso, temos O(n) chamadas na pilha, ou seja, o pivô sempre fica sozinho em uma partição. Já a complexidade de tempo, o pior caso é O(n^2), quando o pivô é sempre o maior ou menor elemento, já o caso médio e o melhor caso são ambos O(n.log(n))

3.4 Ordenador Universal

Iremos analisar cada um dos métodos de **ordenadoruniversal** a seguir:

- **3.4.1 ordenadorUniversal**

  Será dependente do *sort* escolhido, então poderá ser, em média, ou O(n.log(n)) ou O(n^2) temporalmente, e de espaço ou O(1) ou O(log(n)) em média, como visto na seção **3.3 Sorts**.

- **3.4.2 determinaLimiarParticao**

  Depende do **ordenadorUniversal**, mas vamos considerar que, em média, o ***quickSort*** será chamado, teremos em média O(log(n)) espacialmente e O(n.log(n)) temporalmente, no entanto, iterativamente estamos atribuindo um **vector** a outro, o que, como visto anteriormente, custa O(n) espacialmente e temporalmente. Portanto, temos, em média O(n + n.log(n)) temporalmente (= O(n.log(n))) e O(n + log(n)) espacialmente (= O(n)). No pior caso, teremos O(n^2 + n) (= O(n^2)) temporalmente e O(n) espacialmente, o pior caso do ***Quick*** (espacial e temporal) mais a alocação dos vetores. No entanto, também devemos considerar a complexidade da estrutura **do while**, que executa O(log(n)) vezes, então temos: O(log(n) . (n + n.log(n))) = O(n^2.log(n)) temporalmente e O(n) espacialmente.

- **3.4.3 determinaLimiarQuebras**

  Nesta função, faremos os dois *sorts* e faremos mais atribuições de vetores, então teremos, no mínimo, O(n^2) temporalmente e O(n) espacialmente, temporalmente por culpa do ***insertionSort*** e espacialmente por culpa do operador de atribuição dos **vector's,** já que o ***quickSort*** é no caso médio O(n) espacialmente e O(n^2) temporalmente. Similar ao **determinaLimiarParticao**, temos o *loop* **do while**, que aplica o ***quickSort*** log(n) vezes, além do ***shuffleVector***(O(n/2)), então temporalmente temos O(n.log(n) + log(n).(n + log(n) + n.log(n))) = O(n^2.log(n)) temporalmente e O(n) espacialmente.

- **3.4.4 calculaNovaFaixa**

  Esta função só faz operações simples de atribuição e não aloca nova memória dinâmica, portanto, é O(1) tanto espacial quanto temporalmente.

- **3.4.5 imprimeEstatisticas**

É uma função que só realiza **prints**, então é O(1) espacial e temporalmente.

4. Análise de Robustez

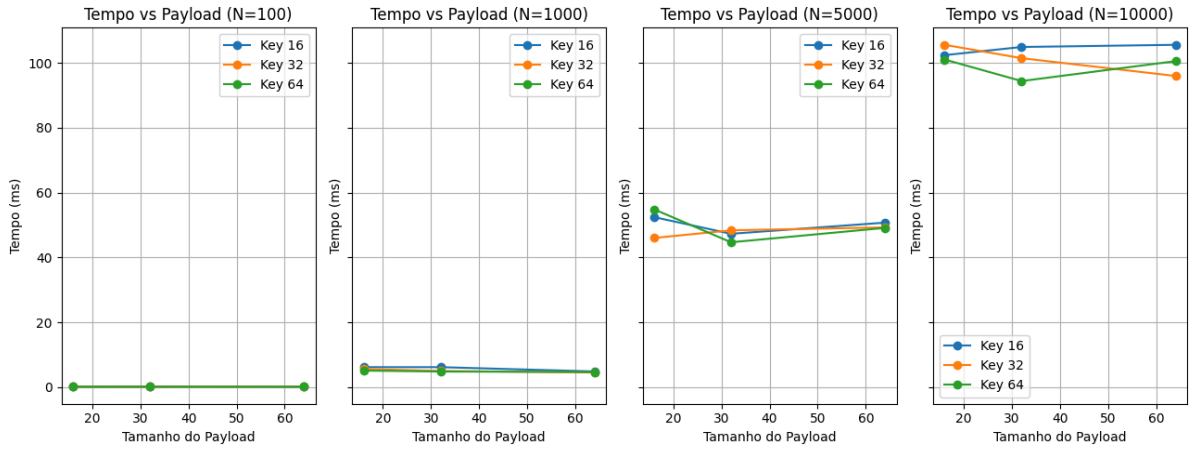
Todos os códigos que envolvem alocação de memória dinâmica foram reforçados com estruturas ***try-catch*** genéricas para lançar mensagens de erro durante a execução, mas não interrompê-la, porque no contexto deste código, a estabilidade é preferível, ainda mais que estamos trabalhando com limiares tão próximos e uma grande quantidade de pontos flutuantes, então precisamos manter os **custos** durante a execução mesmo que signifique vazar memória. O sistema implementado dos ***try-catch*** não irá tratar a exceção, apenas sinalizar onde ocorreu o erro e qual foi ele, por meio do ***fprintf(stderr, ...)***.Os blocos ***try-catch*** também foram inseridos em métodos para sinalizar erros distintos de *leaks*, como algum acesso a uma posição inexistente, por exemplo. Quanto à robustez dos **vectors**, na alocação de memória sua memória, no caso de não possuir tamanho inicial, é iniciado com **_capacidade** igual a um, o mínimo para ter memória alocada, para evitar *leaks*. Como pode ser visto na imagem, após rodar o teste do *Valgrind* para achar *leaks*, nenhum foi achado:

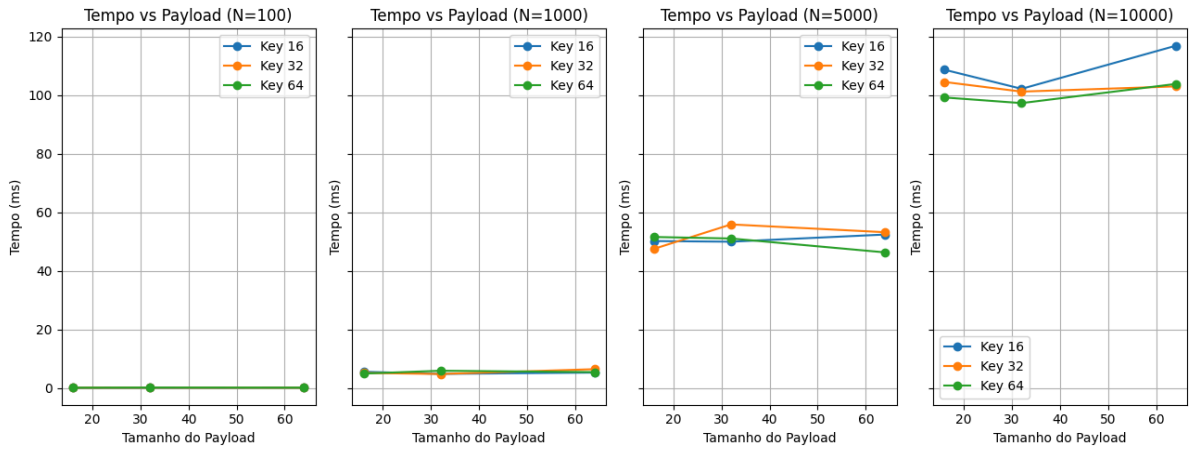
```
==421== HEAP SUMMARY:
==421==     in use at exit: 0 bytes in 0 blocks
==421==   total heap usage: 14 allocs, 14 frees, 83,472 bytes allocated
==421==
==421== All heap blocks were freed -- no leaks are possible
==421==
```

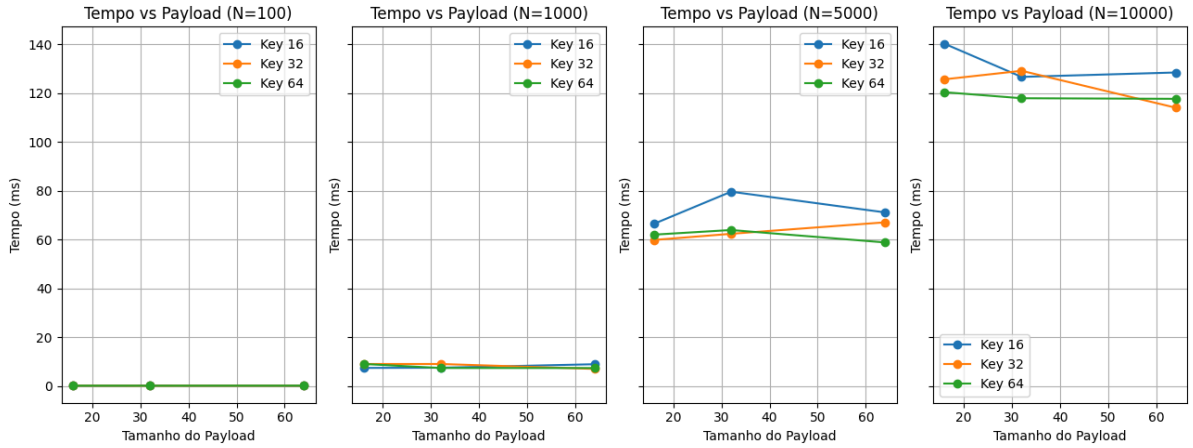
5. Análise Experimental

5.1 Elaboração

Para garantir a robustez do algoritmo, iremos variar as três dimensões com os seguintes valores, **vetores** de tamanhos = {100, 1000, 5000, 10000}, **KEYSZ** = {*int16_t, int32_t, int64_t*}, **PLSZ** = {*int16_t, int32_t, int64_t*}. As variáveis *int*_t* serão utilizadas a fim de reduzir a dificuldade de implementação. A *struct* **item_t** será formada por uma instância de **KEYSZ** e uma de **PLSZ**, enquanto o tamanho do **vector** será definido na main. A ideia principal é mostrar que, com dados e **payloads** uni-valorados e vetores de tamanhos variados, a maior influência no custo vem do tamanho base do **vector**. A partir da análise de tempo, temos os seguintes dados para vetores **ordenados**, **inversamente ordenados** e **desordenados com breaks induzidos por seed**, de tamanhos 100, 1000, 5000 e 10000, respectivamente:







Estes gráficos mostram o impacto do tamanho das chaves na aplicação do algortimo, sempre se mantendo próximas independentemente do tamanho. Também podemos perceber, analisando os dois gráficos mais à direita nas três instâncias, o quanto a inserção de um vetor já ordenado impacta na performance do algoritmo, o tempo de execução é visivelmente menor. Já o algoritmo inversamente ordenado não tem sua performance tão afetada porque o ***Quick Sort*** com mediana de três evita o pior caso e amortiza o ***Insertion Sort***, no entanto, continua mais rápido que o algoritmo desordenado.

O foco destes gráficos é mostrar o comportamento geral, e não exato, da performance do algoritmo com dados uni-valorados, além de mostrar a influência da inserção de um **vector** já **ordenado** na velocidade de execução, dado que é comum ocorrer a diferença de medição de tempo de execução de um mesmo código.

5.2

- **5.2.1**

Utilizando os dados gerados na seção **5.1**, aplicando a regressão temos os seguintes resultados para cada tipo de vetor como entrada:

- Ordenado: **a** = 3.e-6, **b** = -1.e-6 e **c** = -1.e-5;
- Inversamente Ordenado: **a** = 4.e-6, **b** = -1.e-6 e **c** = -15.e-7;

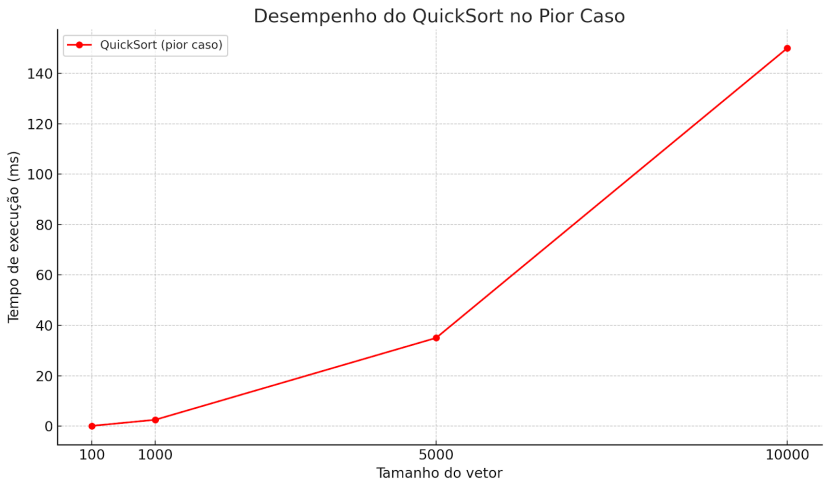
- Desordenado usando ***seed's*** aleatórias: **a** = 4.21.e-6, **b** = -1.34.e-6 e **c** = -1.1172e-5;

- **5.2.2 e 5.2.3**

Aplicando o algoritmo a diferentes tamanhos de vetos e utilizando os coeficientes calculados, temos:

- Ordenado:
  Tamanho = 100 -> **limParticao** = 2; **limQuebras** = 2.
  Tamanho = 1000 -> **limParticao** = 2; **limQuebras** = 8.
  Tamanho = 5000 -> **limParticao** = 2; **limQuebras** = 14.
  Tamanho = 10000 -> **limParticao** = 2; **limQuebras** = 16.
- Inversamente Ordenado:
  Tamanho = 100 -> **limParticao** = 2; **limQuebras** = 1.
  Tamanho = 1000 -> **limParticao** = 2; **limQuebras** = 5.
  Tamanho = 5000 -> **limParticao** = 2; **limQuebras** = 14.
  Tamanho = 10000 -> **limParticao** = 2; **limQuebras** = 16.
- Desordenado por ***seed***:
  Tamanho = 100 -> **limParticao** = 2; **limQuebras** = 1.
  Tamanho = 1000 -> **limParticao** = 2; **limQuebras** = 8.
  Tamanho = 5000 -> **limParticao** = 2; **limQuebras** = 16.
  Tamanho = 10000 -> **limParticao** = 2; **limQuebras** = 24.

O TAD apresentou desempenho satisfatório, sendo o **limParticao** limitado apenas reflexo da escolha por variáveis uni-valoradas para a comparação. Em relação às versões não otimizadas dos algoritmos, o ***Quick Sort*** seria o principal risco, como mostrado pelo gráfico:



O custo quadrático, quando o pivô escolhido é sempre o maior ou menor elemento nos casos do vetor estar **Ordenado** ou **Inversamente Ordenado**, é um fator que atrapalha muito o tempo de execução. Outro fator de risco seria usar o ***Insertion Sort*** sem a condição de **minTamParticao**, pois geraria um número massivo de **comparacoes** e **movimentacoes**, o que é custoso computacionalmente.

- **5.3**

O **limParticao** será 2 sempre, pois como os coeficientes são pequenos, o ***Insertion Sort*** acaba tendo custo comparável, ou menor, do que o ***Quick Sort***, já que chamadas recursivas também são contabilizadas. Outra razão para isto é o fato de que os testes estão sendo feitos com variáveis que têm diferenças de tamanho relativamente baixas, já que o intuito é exatamente provar que nestes casos o **tamanho do vector** importa mais, além de que, sem um **limiarCusto** bem definido, o algoritmo pode convergir de maneira errada.

O **limQuebras** continuará funcionando quase normalmente (apenas a questão do **limiarCusto** não fica muito bem definida), porque sua lógica não é baseada na escolha entre algum dos dois algoritmos de ordenação, já que ele obrigatoriamente invoca ambos, então, como o ***Quick Sort*** é negativamente impactado por muitas quebras, o limiar aumenta.

6. Conclusão

O problema tratado neste trabalho foi o de montar um **Ordenador Universal**, que escolhe o melhor momento para se utilizar cada *sort* e é flexível a tipos distintos de dados, para a empresa Zambs. O uso de uma estrutura de dados como o **vector** agregou muito ao código em conjunto ao uso dos **templates**, porque permitiu armazenamento eficiente, atribuições entre vetores de forma mais simples, expansão do limite de forma automática e, principalmente, a adaptabilidade para diferentes tipos de dados, basta mudar a instância na *main*. Caso o trabalho fosse feito utilizando de *arrays* convencionais, provavelmente sua complexidade de implementação seria bem maior.

Em geral, o trabalho proveu uma noção boa de como os custos de execução são importantes e em qual situação cada *sort* é superior ao outro. Outro ponto a se destacar é como os diferentes algoritmos de ordenação são facilmente adaptáveis a tipos de dados diferentes ao **int** convencional.

7. Bibliografia

Lacerda, A. and Meira JR, W.(2024). Slides da disciplina de estruturas de dados, Aula 07 - QuickSort, Aula 05 - Ordenação: Bolha, Inserção e Seleção.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Documentação Extra

8.1 TAD ordenadoruniversal

- **void ordenadorUniversal(vector<T>& vetor, int minTamParticao, estatisticas& estat, int limQuebras)**: o método foi alterado para testar se o **limQuebras** é menor que o número de quebras atual do **vetor**, pois em casos de muitas quebras, o ***Quick Sort*** é ineficiente. Já o ***Merge Sort*** é sempre eficiente, embora tenha que alocar memória extra. Em casos ideais, o ***Quick*** é preferível ao ***Merge*** pela sua velocidade de execução. O ***Insertion*** continua restrito a casos em que o tamanho do **vetor** a ser ordenado é inferior ou igual ao **minTamParticao.**

8.2 Sorts

- **void merge(vector<T>& vetor, int esquerda, int meio, int direita, estatisticas& estat)**: é a função responsável por fazer com que o ***Merge Sort*** funcione, estabelece quais são os os **subvetores**, comparando seus elementos e povoando o **vector** principal com estes ordenadamente, além de contabilizar as estatísticas necessárias.
- **void mergeSort(vector<T>& vetor, int esquerda, int direita, estatisticas& estat)**: é a função responsável por chamar a **merge** e dividir o vetor na metade para fazer as chamadas recursivas para cada uma destas.

9. Complexidade Extras

A função de ordenação adicionada ***Merge Sort*** funciona dividindo o vetor em subvetores à direita e à esquerda do meio até que estes possam ser comparados em pequenas partições, ao fim deste processo, o ***merge*** é chamado para juntar essas partes e montá-las ordenadamente no **vector** principal. Por esta característica de ser dividido na metade até que sejam partições mínimas e por precisar chamar o ***merge*** para cada subdivisão, temos que a complexidade temporal é O(n.log(n)), já a espacial, é necessário alocar uma quantidade de memória proporcional ao **vector** original, portanto, é O(n).

Como o custo temporal e espacial do ***Merge Sort*** é sempre constante, o custo geral do código, no caso médio, continua $O(n^2.log(n))$ temporalmente, chama o ***Quick*** ou o ***Merge*** e, em pequenas partições, o ***Insertion***, e espacialmente temos O(n) da alocação do **vector** extra pelo **operador** de atribuição ou do pior caso do ***Quick Sort***. O pior caso temporalmente seria $O(n^2)$, quando o ***Insertion*** é chamado para um **vetor** inversamente ordenado ou quando o ***Quick*** é chamado para um vetor ordenado ou inversamente ordenado e o pivô é sempre o maior ou menor elemento.

10. Considerações Finais

O ***Merge Sort*** não foi implementado em **determinaLimiarQuebras**, porque é um algoritmo muito eficiente em termos de **comparações**, **movimentações** e **chamadas**, portanto, a influência dele no custo seria prejudicial à determinação deste limiar para o funcionamento ideal do ***Quick*** e do ***Insertion***, pois estaria ativamente diminuindo-o. Portanto, as implementações relativas ao **Merge** em **determinaLimiarQuebras** ficaram apenas comentadas.