SUMÁRIO

1 INTRODUÇÃO

Este trabalho está dividido em 7 capítulos. Neste primeiro capítulo introdutório, serão definidos o problema da ordenação e sua importância. No capítulo seguinte, serão apresentados alguns dos conceitos que serão importantes para o entendimento do conteúdo que se segue. Os próximos três capítulos (3, 4 e 5) apresentarão, ao todo, 14 algoritmos que resolvem o problema da ordenação, bem como suas implementações, complexidades temporais e espaciais, dentre outras características. O capítulo 6 realizará, então, uma análise empírica a partir dos resultados obtidos com a execução dos algoritmos, à luz de determinadas métricas. Por fim, o último capítulo trará uma conclusão.

1.1 O problema da ordenação

O problema da ordenação consiste em reorganizar uma sequência de dados para que seus elementos estejam dispostos em uma ordem específica, geralmente crescente ou decrescente. Em termos formais, o problema pode ser definido da seguinte maneira: dada uma sequência de n números $\langle a_1, a_2, ..., a_n \rangle$, produzir uma permutação $\langle a'_1, a'_2, ..., a'_n \rangle$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$ (CORMEN *et al.*, 2009, p. 5).

A ordenação é uma das operações fundamentais em computação, com aplicações diretas em diversas áreas, como pesquisa, organização de dados, algoritmos de busca e análise estatística. Além disso, é uma etapa essencial em muitos outros algoritmos, como aqueles utilizados em análise de grafos, aprendizado de máquina e simulação de sistemas.

A importância do problema da ordenação também está relacionada à sua presença em problemas cotidianos, como organização de listas de compras, ordenação de arquivos em sistemas operacionais, ordenação de registros em bancos de dados e priorização de tarefas em filas de execução. Além disso, a resolução desse problema proporciona uma base teórica para o estudo de outros problemas mais complexos em computação.

Compreender e resolver eficientemente o problema da ordenação é crucial para otimizar o desempenho de sistemas computacionais. A escolha de um algoritmo adequado para um determinado cenário pode impactar significativamente os requisitos de tempo e espaço, especialmente em aplicações que envolvem grandes volumes de dados. Assim, o estudo da ordenação não só permite avanços na própria área de computação, mas também contribui para soluções mais eficientes em diversas áreas do conhecimento.

2 PRELIMINARES

2.1 Ordenação interna e externa

A ordenação pode ser classificada em dois tipos principais: interna e externa, dependendo da forma como os dados são manipulados durante o processo de ordenação.

Ordenação interna ocorre quando todos os dados a serem ordenados cabem na memória principal do computador (RAM). Esse tipo de ordenação é amplamente utilizado em cenários onde o volume de dados é relativamente pequeno e o acesso rápido à memória permite um desempenho eficiente.

Ordenação externa, por outro lado, é aplicada quando o volume de dados é tão grande que excede a capacidade da memória principal, necessitando do uso de memória secundária, como discos rígidos ou SSDs. Nesse caso, os algoritmos devem ser projetados para minimizar o número de acessos à memória secundária, que é consideravelmente mais lenta do que a memória principal. Exemplos incluem algoritmos de ordenação por blocos e Mergesort externo.

Neste trabalho, o foco será exclusivamente na ordenação interna, abordando os principais algoritmos utilizados, suas características e desempenhos em diferentes cenários.

2.2 Ordenação estável

A ordenação é considerada estável quando preserva a ordem relativa dos elementos com chaves iguais na sequência de entrada. Em outras palavras, se dois elementos possuem a mesma chave e um deles aparece antes do outro na sequência original, um algoritmo de ordenação estável garantirá que essa ordem relativa seja mantida na sequência ordenada.

A estabilidade em algoritmos de ordenação é especialmente importante em cenários onde os elementos possuem atributos adicionais além da chave utilizada para a ordenação. Por exemplo, em um sistema de gerenciamento de registros, onde cada registro é identificado por um nome e uma data, a estabilidade permite ordenar os registros primeiro por nome e, em seguida, por data, sem perder a ordem relativa estabelecida na primeira ordenação.

A Tabela 1 apresenta os algoritmos que serão implementados neste trabalho e indica quais deles são estáveis:

Tabela 1 – Estabilidade dos algoritmos apresentados

Algoritmo	Estável
Bolha	✓
Coquetel	✓
Selecao	
Insercao	✓
Shellsort	
Mergesort	✓
Heapsort	
Quicksort	
Countingsort	✓
Bucketsort	✓
RadixsortC	✓
RadixsortB	✓

2.3 Análise de algoritmos

2.3.1 Notação assintótica

A Notação Assintótica é um conjunto de ferramentas matemáticas utilizada na análise de algoritmos para descrever o comportamento limitante ou de crescimento de uma função à medida que o tamanho da entrada se aproxima do infinito. Seu propósito principal é fornecer uma forma de classificar e comparar a eficiência de algoritmos, focando apenas na ordem de magnitude do tempo ou espaço consumido e ignorando constantes de proporcionalidade e termos de ordem inferior. Ao expressar a complexidade de tempo de um algoritmo, a notação assintótica permite prever como o algoritmo escalará em grandes volumes de dados, independentemente de detalhes de implementação ou da velocidade específica do hardware.

Definir a complexidade de um algoritmo exige o uso de notações assintóticas precisas para delimitar seu comportamento. A notação $\mathcal{O}(f(n))$ fornece um limite superior para complexidade de um algoritmo, seja ela temporal ou espacial, descrevendo o pior caso. Ela significa que, para valores grandes de entrada n, a complexidade nunca será pior do que uma função $c \cdot f(n)$, onde c é uma constante positiva e $f(n)$ é a função de crescimento. Já a notação $\Theta(f(n))$ fornece um limite assintótico apertado, definindo o comportamento

exato da complexidade de um algoritmo. Isso significa que, para n grande, a complexidade está limitado superior e inferiormente por múltiplos constantes de $f(n)$, indicando que $f(n)$ é a ordem de crescimento precisa, tanto no melhor quanto no pior caso.

2.3.2 Cota inferior

Cota inferior define o limite mínimo de desempenho necessário para resolver um problema. Isso significa que nenhum algoritmo pode resolver o problema em menos tempo do que a cota inferior definida, independentemente da abordagem utilizada. A notação usada para descrever a cota inferior é a notação $\Omega(f(n))$.

Para algoritmos de ordenação baseados em comparação, a cota inferior é $\Omega(n \log_2 n)$. Esse resultado demonstra que qualquer algoritmo de ordenação que utilize apenas comparações entre elementos, no pior caso, precisará de pelo menos $n \log_2 n$ comparações para ordenar n elementos. Este limite foi provado por (KNUTH, 1998), onde ele utiliza conceitos de árvores de decisão para demonstrar que a profundidade mínima da árvore que representa todas as permutações possíveis dos elementos é $\log_2(n!)$, o que se aproxima de $n \log_2 n$ pela fórmula de Stirling.

2.3.3 Divisão e conquista e Teorema mestre

O paradigma Divisão e conquista é uma das estratégias mais poderosas no projeto de algoritmos, na qual um problema complexo é resolvido de forma recursiva por meio de três etapas: dividir, conquistar e combinar. Na etapa de divisão, o problema original de tamanho n é quebrado em a subproblemas menores, cada um de tamanho n/b. Na etapa de conquista, os subproblemas são resolvidos recursivamente. Finalmente, na etapa de combinação, as soluções dos subproblemas são reunidas para formar a solução final do problema original.

A eficiência de um algoritmo que segue o paradigma Divisão e conquista é caracterizada por uma relação de recorrência da forma $T(n) = aT(n/b) + f(n)$, sendo que $T(n)$ representa o tempo total de execução. O Teorema Mestre é a ferramenta matemática que permite solucionar essas recorrências e determinar sua complexidade assintótica. Ele funciona comparando o custo do trabalho de divisão e combinação, $f(n)$, com o custo do trabalho realizado nas folhas da árvore de recursão, dado por $n^{\log_b a}$. O teorema afirma que a solução $T(n)$ cairá em um de três casos:

- Se $f(n)$ for assintoticamente menor que $n^{\log_b a}$, a complexidade será $O(n^{\log_b a})$;
- Se $f(n)$ for assintoticamente maior, a complexidade será $O(f(n))$;
- E se forem assintoticamente iguais, a complexidade será $O(n^{\log_b a} \log n)$.

2.3.4 Probabilidade e valor esperado

A probabilidade é a medida numérica da chance de um evento específico ocorrer, variando de zero (impossibilidade) a um (certeza), e é fundamental para modelar a incerteza. Já o valor esperado (ou esperança matemática) é uma medida que resume o resultado mais provável ou a média de longo prazo de um fenômeno aleatório. Ele é calculado ponderando-se cada resultado possível pelo seu respectivo valor de probabilidade, oferecendo uma estimativa útil para a tomada de decisões em cenários de risco, como jogos de azar ou análises financeiras.

Esses dois conceitos matemáticos serão utilizados no próximo capítulo para a análise de complexidade da versão probabilística do algoritmo Quicksort.

2.4 Arquitetura de computadores

A análise de complexidade $\mathcal{O}(f(n))$ é fundamental, mas não é o único fator que determina o tempo de execução real. Em sistemas de computação modernos, a hierarquia de memória e o conceito de localidade de referência também ajudam a entender o desempenho.

2.4.1 Hierarquia de memória

O processador (CPU) interage com diferentes níveis de memória que variam drasticamente em velocidade de acesso e capacidade de armazenamento. A Tabela 2, apresentada a seguir, relaciona os tipos de memória com suas capacidades e velocidades. Como os valores exatos de capacidade e latência podem variar entre fabricantes e modelos, esta é uma classificação em alto nível que se mantém consistente, independentemente desses fatores.

O processador é muito rápido. Uma operação básica, como uma comparação ou soma, pode levar apenas um ciclo de clock. No entanto, quando ocorre uma falha de cache — isto é, quando se busca por um dado que não está no cache e precisa ser carregado da RAM — o processo pode custar dezenas ou até centenas de ciclos de clock. Desse modo, a

Tabela 2 – Capacidade e velocidade de acesso por tipo de memória

Nível de memória	Capacidade	Velocidade de acesso
Registradores/L1 Cache	Muito Baixa	Extremamente rápido
L2/L3 Cache	Baixa	Muito rápido
Memória principal (RAM)	Alta	Lento
Memória secundária (SSD/HDD)	Muito Alta	Muito lento

velocidade de um algoritmo em um grande conjunto de dados é frequentemente limitada pelo tempo gasto esperando o acesso à memória, e não pelo número de operações da CPU

2.4.2 Localidade de referência

Para mitigar o alto custo de acesso à RAM, os processadores utilizam um princípio chamado localidade de referência. Este princípio se refere à tendência de um programa de acessar o mesmo conjunto de locais de memória ou locais próximos em um curto período de tempo.

- Localidade temporal: Se um item foi acessado, é provável que ele seja acessado novamente em breve.
- Localidade espacial: Se um item foi acessado, é provável que os itens armazenados em endereços de memória próximos sejam acessados em breve.

A eficiência de cache é aumentada porque o cache opera em "blocos" ou "linhas de cache", geralmente de 64 bytes. Ou seja, quando um dado $v[i]$ não está no cache e a CPU precisa carregá-lo da RAM, ela não carrega apenas $v[i]$, mas sim um bloco inteiro que contém $v[i]$ e vários elementos adjacentes, como $v[i+1]$, $v[i+2]$, etc.

Se um algoritmo seguir a localidade espacial, acessando sequencialmente $v[i]$, $v[i+1]$, $v[i+2]$, ele encontrará os próximos dados que precisa já carregados no cache. Assim, o custo de esperar a RAM é pago apenas uma vez para o bloco inteiro, o que torna o acesso subsequente, linear e sequencial quase instantâneo.

Algoritmos que varrem grandes vetores de forma sequencial maximizam a Localidade Espacial e, por isso, são considerados "cache-friendly". Em contraste, algoritmos que "pulam" em endereços de memória dispersos desperdiçam o bloco de cache carregado e incorrem em muitas falhas de cache.

3 MÉTODOS INFERIORES

Neste capítulo, são apresentados quatro algoritmos básicos para o problema da ordenação. Esses algoritmos são chamados de inferiores por serem simples, intuitivos e, principalmente, por apresentarem, no mínimo, complexidade temporal quadrática nos casos médio e pior.

3.1 Bolha

O algoritmo Bolha é um dos algoritmos mais simples para o problema da ordenação. Ele consistem em percorrer várias vezes o vetor, comparando elementos adjacentes e trocando-os de posição caso estejam fora da ordem desejada. Nesse processo, os elementos flutuam para suas posições finais. Em outras palavras, "[...] os elementos da lista são movidos para as posições adequadas de forma contínua, assim como uma bolha move-se num líquido. Se um elemento está inicialmente numa posição i e, para que a lista fique ordenada, ele deve ocupar a posição j, então ele terá que passar por todas as posições entre i e j." (VIANA; CINTRA, 2011, p.8).

```
1  void Bolha(int n, int *v) {
2      char trocou = 1;
3      for (int i = n - 1; i > 0 && trocou; i--) {
4          trocou = 0;
5          for (int j = 0; j < i; j++)
6              if (v[j] > v[j + 1]) {
7                  Troca(v + j, v + j + 1);
8                  trocou = 1;
9              }
10     }
11 }
```

Esta versão do algoritmo Bolha é também conhecida como "Bolha com Flag", por conta do uso da variável *trocou*. Perceba que, sem essa variável, o algoritmo continua funcionando da mesma forma; porém o laço externo executará sempre $n-1$ iterações, mesmo que o vetor fique ordenado antes.

Observe que o algoritmo Bolha oferece uma ordenação estável, pois, quando dois elementos adjacentes são iguais, eles não são trocados.

Corretude

Ao final de uma iteração do laço externo, o maior elemento de $v[0..i]$ estará na posição i. Dessa forma, ao final da primeira iteração, o maior elemento do vetor estará na posição $n-1$; ao final da segunda iteração, o segundo maior estará na posição $n-2$, e assim por diante. Portanto, o algoritmo Bolha é correto.

Desempenho

Em termos de consumo de memória adicional, o algoritmo declara apenas três variáveis escalares, sendo duas delas para controle de laço de repetição. Logo, sua complexidade espacial é $\Theta(1)$. Esta observação será omitida para os algoritmos apresentados a seguir que tenham a mesma complexidade espacial.

Já para a complexidade temporal, o melhor caso ocorre quando a entrada já está ordenada em ordem crescente, pois não é realizada nenhuma troca. Exige-se, assim, apenas uma iteração do laço externo, resultando em complexidade $\Theta(n)$.

Por outro lado, o pior caso ocorre quando o menor elemento está na última posição, sendo necessárias exatamente $n-1$ iterações do laço externo para que ele chegue à primeira posição. Com isso, a quantidade de iterações do laço interno é dada pela seguinte soma:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \tag{3.1}$$

Isso implica em uma complexidade quadrática $\Theta(n^2)$. Em média, a complexidade temporal do algoritmo Bolha é $\mathcal{O}(n^2)$.

3.2 Coquetel

O Coquetel é um algoritmo de ordenação estável que traz a mesma ideia do Bolha, mas com um laço interno adicional. O primeiro laço, assim como no Bolha, itera da esquerda para a direita, empurrando os maiores elementos para o final do vetor, enquanto o segundo itera no outro sentido, empurrando os menores elementos para o início.

```
1 void Coquetel(int n, int *v) {
2     char trocou = 1;
3     for (int i = 0, j = n - 1; i < j && trocou; i++, j--) {
4         trocou = 0;
5         for (int k = i; k < j; k++)
```

```
6               if (v[k] > v[k + 1])
7                   Troca(v + k, v + k + 1), trocou = 1;
8           if (trocou) trocou = 0;
9           else return;
10          for (int k = j - 2; k >= i; k--)
11              if (v[k] > v[k + 1])
12                  Troca(v + k, v + k + 1), trocou = 1;
13      }
14  }
```

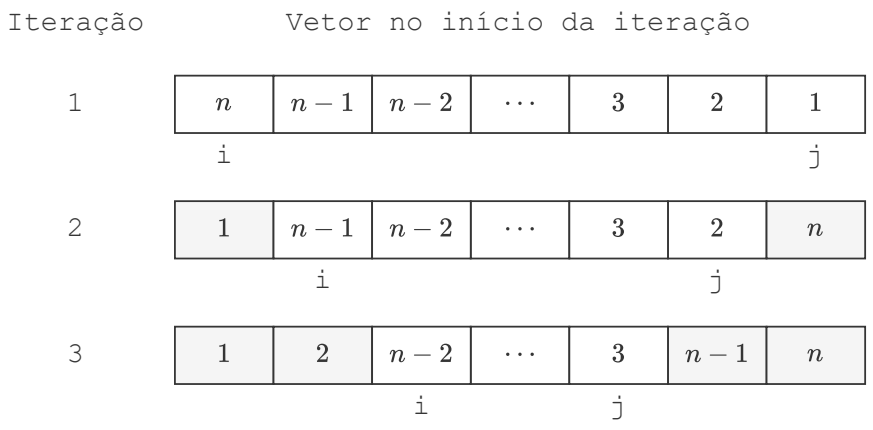
Corretude

Ao final de uma iteração do laço externo, o menor e o maior elemento de $v[i..j]$ estarão nas posições i e j, respectivamente. Dessa forma, ao final da primeira iteração, o menor elemento estará na posição 0 e o maior, na posição $n-1$; na segunda, o segundo menor, na posição 1, e o segundo maior, na posição $n-2$, e assim por diante. Esse processo garante que, quando $i \geq j$, o vetor estará ordenado; portanto, o algoritmo é correto.

Desempenho

Embora as complexidades dos casos melhor, pior e médio continuem sendo, respectivamente, $\Theta(n)$, $\Theta(n^2)$ e $\mathcal{O}(n^2)$, o pior caso do Coquetel é menos frequente, pois ocorre apenas quando a entrada está em ordem decrescente.

Figura 1 – Coquetel aplicado ao vetor $[n, n-1, ..., 2, 1]$.



Fonte: Elaborado pelo autor.

Considere então que $T(n)$ é a quantidade total de iterações dos dois laços internos

no pior caso. A figura 1 ajuda a notar que a seguinte identidade é válida:

$$T(n) = \begin{cases} 0, & n < 2 \\ T(n-2) + 2n - 3, & n \geq 2 \end{cases}$$

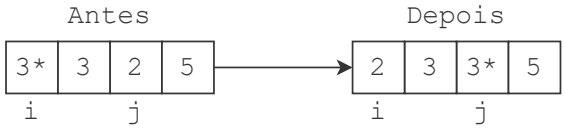
Ao resolver a recorrência mostrada acima, pode-se observar que $T(n)$ é igual a soma 3.1. Portanto, os algoritmos Bolha e Coquetel, em seus piores casos, executam exatamente a mesma quantidade de iterações.

3.3 Seleção

O algoritmo Seleção é um método de ordenação instável, como mostra a figura 2. A ideia é bastante simples: execute n iterações; na i-ésima iteração, selecione o i-ésimo menor elemento e coloque-o na posição $i-1$. Uma propriedade interessante desse algoritmo é que ele realiza o número mínimo possível de trocas para ordenar o vetor.

```
1 void Selecao(int n, int *v) {
2     for (int i = 0; i < n - 1; i++) {
3         int j = i;
4         for (int k = i + 1; k < n; k++)
5             if (v[j] > v[k])
6                 j = k;
7         if (j > i) Troca(v + i, v + j);
8     }
9 }
```

Figura 2 – Seleção aplicado ao vetor $[3,3,2,5]$.



Fonte: Elaborado pelo autor

Corretude

É invariante que, ao final de cada iteração do laço externo, os elementos de $v[0..i]$ já estejam em suas posições definitivas. Portanto, ao final da última iteração, quando i for igual a $n-1$, o vetor $v[0..n-1]$ estará ordenado.

Desempenho

O ponto crítico do algoritmo Seleção é o teste condicional *if* na linha 5. A quantidade de vezes que esse ponto é executado é sempre dada pela soma 3.1, ou seja, não há melhor nem pior caso. Portanto, esse algoritmo sempre requer tempo igual a $\Theta(n^2)$.

3.4 Inserção

O algoritmo Inserção é um método de ordenação estável com o melhor desempenho entre os métodos inferiores. A ideia por trás do algoritmo é simples: para i variando de 1 a $n-1$, nesta ordem, insira $v[i]$ em $v[0..i-1]$, de modo que $v[0..i]$ esteja em ordem.

Em especial, o algoritmo Inserção possui complexidade temporal linear para entradas quase ordenadas. Por conta disso, outros algoritmos fazem uso dele como sub-rotina. O Shellsort, por exemplo, utiliza o Inserção para tornar o vetor h-ordenado.

Definição 3.1 *Para um dado número natural $h > 0$, um vetor v de tamanho n é considerado h-ordenado quando $v[i] \leq v[i+h]$, para todo $0 \leq i < n-h$.*

```
1 void H_Ordena(int n, int h, int *v) {
2     int chave, j;
3     for (int i = h; i < n; i++) {
4         chave = v[i];
5         for (j = i; j >= h && v[j - h] > chave; j -= h)
6             v[j] = v[j - h];
7         v[j] = chave;
8     }
9 }
```

A implementação do Inserção pode ser idêntica à função H_Ordena mostrada acima, bastando fazer $h = 1$, pois, nesse caso em particular, um vetor h-ordenado é simplesmente um vetor ordenado.

```
1 void Insercao(int n, int *v) {
2     H_Ordena(n, 1, v);
3 }
```

Corretude

Ao final de cada iteração do laço externo, o segmento $v[0..i]$ está ordenado. Esse invariante permanece válido durante todas as iterações, inclusive na última. Portanto, ao final do processo, $v[0..n-1]$ estará ordenado.

Desempenho

No melhor caso, a complexidade é linear e ocorre quando o vetor já está ordenado, pois a condição do laço interno será sempre falsa. Já o pior caso acontece quando o vetor está em ordem decrescente; nesse cenário, $v[i]$ é sempre inserido na posição 0. O ponto crítico é a linha 6 e, mais uma vez, a soma 3.1 determina a quantidade de vezes que essa linha será executada. Portanto, a complexidade no pior caso é $\Theta(n^2)$. Em média, o algoritmo Inserção consome tempo proporcional a $\mathcal{O}(n^2)$.

4 MÉTODOS SUPERIORES

4.1 Shellsort

O Shellsort foi desenvolvido por Donald L. Shell e apresentado em 1959 no artigo *A High-Speed Sorting Procedure*. Trata-se de um método de ordenação não estável que se baseia na escolha de uma sequência de incrementos h e, para cada valor h_i, do maior para o menor, transforma o vetor de entrada em um vetor h_i-ordenado.

Definição 4.1 *No contexto do algoritmo Shellsort, uma sequência de incrementos é uma sequência de números inteiros positivos* $h = \{h_1, h_2, ..., h_r\}$*, tal que* $h_1 = 1$ *e* $h_i < h_{i+1}$*.*

A análise de complexidade do Shellsort não é uma tarefa simples e depende da sequência de incrementos escolhida. Diversos pesquisadores propuseram diferentes sequências ao longo dos anos; algumas das mais conhecidas são mostradas na Tabela 3.

Tabela 3 – Sequências de incrementos

Referência	Sequência	Pior caso
(SHELL, 1959)	$h_i = \lfloor n/2^i \rfloor$	$\Theta(n^2)$
(HIBBARD, 1963)	$h_i = 2^i - 1$	$\Theta(n^{3/2})$
(KNUTH, 1973)	$h_i = (3^i - 1)/2$	$\Theta(n^{3/2})$
(SEDGEWICK, 1986)	$h_i = 4^i + 3 \times 2^{i-1} + 1$	$\mathcal{O}(n^{4/3})$
(TOKUDA, 1992)	$h_i = \left\lceil \frac{9(9/4)^i - 4}{5} \right\rceil$	-
(CIURA, 2001)	$1, 4, 10, 23, 57, 132, 301, 701, 1750$	-

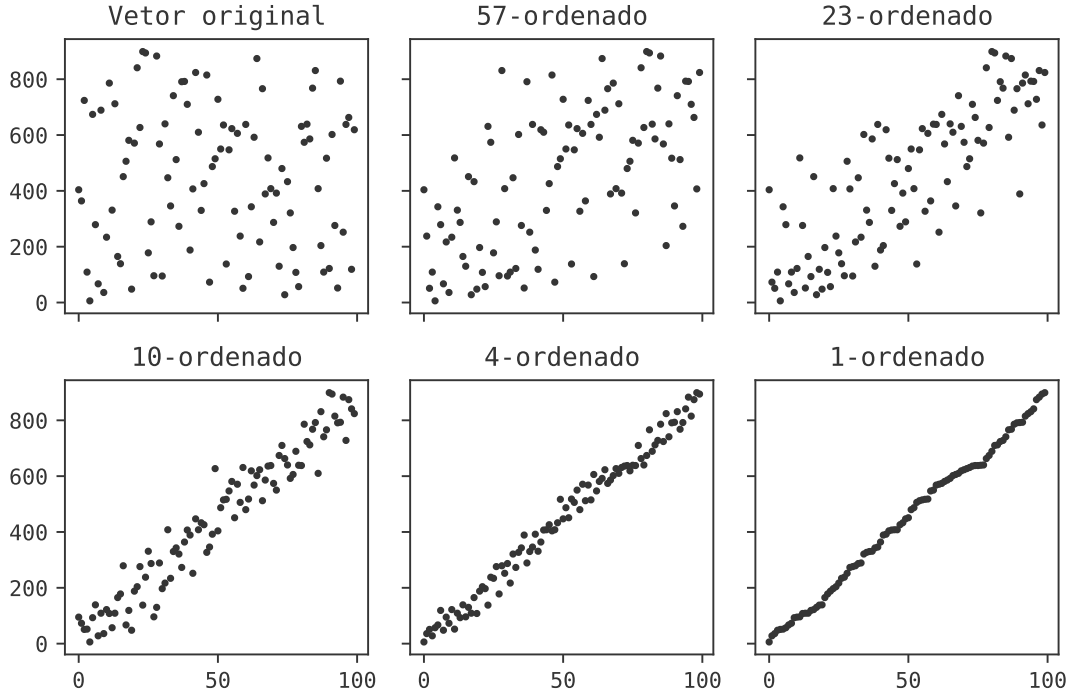
Neste trabalho é utilizada a sequência proposta por (CIURA, 2001). Tal sequência não possui uma análise precisa de complexidade de pior caso, mas demonstrou um bom desempenho experimentalmente, superando todas as outras. Os primeiros oito termos da sequência, escolhidos empiricamente pelo autor, são os mostrados na Tabela 3. Para expandir a sequência, uma abordagem comum é seguir a fórmula:

$$h_i = \lfloor 2{,}25 \times h_{i-1} \rfloor$$

Desta forma, para a implementação do Shellsort mostrada em linguagem C abaixo, considere que o vetor *CiuraSeq* foi criado de forma global e contém os primeiros 24 termos da sequência descrita acima, em ordem decrescente.

```
1 void Shellsort(int n, int *v) {
2     for (int i = 0; i < 24; i++)
3         H_Ordena(n, CiuraSeq[i], v);
4 }
```

Figura 3 – Estado de um vetor durante iterações do Shellsort.



Fonte: Elaborado pelo autor

4.2 Mergesort

O Mergesort é um algoritmo de ordenação estável baseado na técnica de divisão e conquista, proposto por John von Neumann em 1945. Essa técnica consiste em um processo recursivo de três etapas:

1. Divisão: divide-se o problema em instâncias menores do mesmo problema.
2. Conquista: resolve-se cada uma das instâncias definidas na etapa anterior.
3. Combinação: combinam-se as soluções para resolver o problema original.

Enquanto as etapas de divisão e conquista consistem apenas em algumas operações aritméticas e duas chamadas recursivas, a de combinação requer um pouco mais de atenção. Nesse momento, tem-se dois vetores ordenados, e é necessário juntá-los em apenas um vetor também ordenado, em um processo chamado intercalação, que aqui será referido pelo termo em inglês *merge*.

Intercalação de vetores ordenados

A implementação da operação *merge* é mostrada em linguagem C abaixo. Ela assume que $v[l..m-1]$ e $v[m..r-1]$ estão ordenados. O resultado é que $v[l..r-1]$ também ficará em ordem. A função CopiaVetor, na linha 2, copia os $r-l$ elementos de $v[l..r-1]$ para o vetor auxiliar $v\_aux[l..r-1]$. Doravante, essa função será usada de forma similar, sem mais explicações.

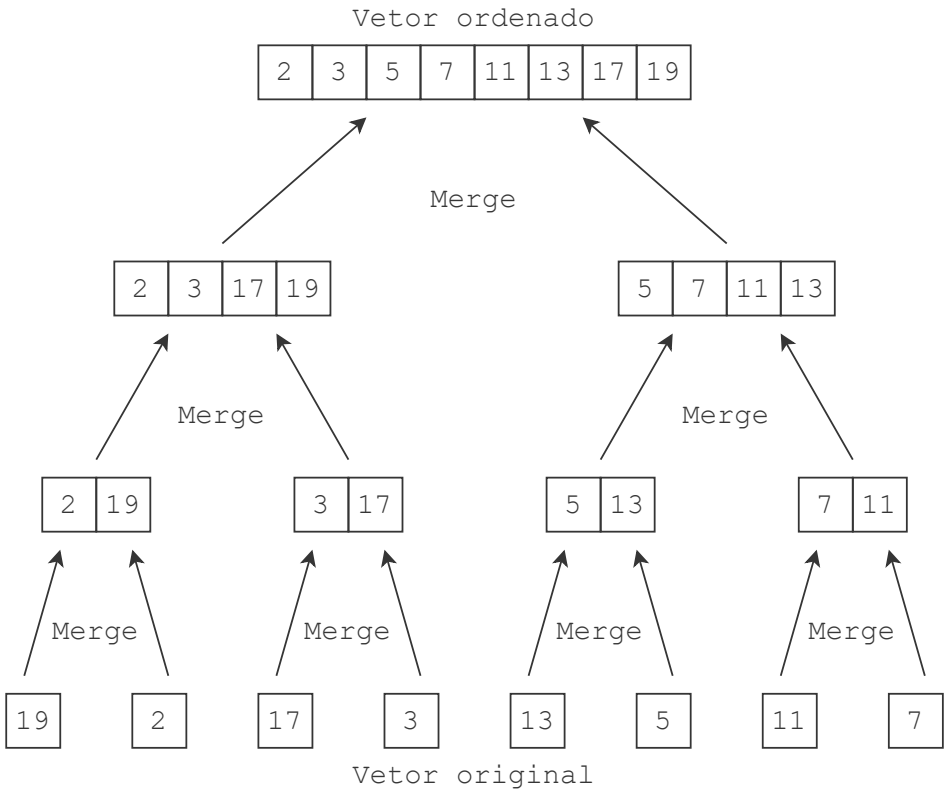
```
1  void Merge(int l, int m, int r, int *v, int *v_aux) {
2      CopiaVetor(r - l, v + l, v_aux + l);
3      int i = l, j = m, k = l;
4      while (i < m && j < r) {
5          if (v_aux[i] <= v_aux[j]) {
6              v[k++] = v_aux[i++];
7          } else {
8              v[k++] = v_aux[j++];
9          }
10     }
11     while (i < m) {
12         v[k++] = v_aux[i++];
13     }
14 }
```

O algoritmo Mergesort

Este trabalho traz uma implementação dividida em dois métodos. O primeiro é apenas uma interface que segue o mesmo padrão de assinatura dos métodos de ordenação mostrados anteriormente, e é responsável por alocar e liberar a memória auxiliar, além de invocar o segundo método. O segundo é uma implementação recursiva do algoritmo que executa as três etapas da técnica já mencionada. A função AlocaVetor usa malloc para alocar um vetor com o tamanho especificado e verifica se a alocação ocorreu com sucesso.

```
1  void Mergesort(int n, int *v) {
2      int *v_aux = AlocaVetor(n);
3      MergesortRec(0, n, v, v_aux);
4      free(v_aux);
5  }
6  void MergesortRec(int l, int r, int *v, int *v_aux) {
7      if (l < r - 1) {
8          int m = (l + r) / 2;
9          MergesortRec(l, m, v, v_aux);
10         MergesortRec(m, r, v, v_aux);
11         Merge(l, m, r, v, v_aux);
12     }
13 }
```

Figura 4 – Ilustração do Mergesort aplicado ao vetor $[19, 2, 17, 3, 13, 5, 11, 7]$.



Fonte: Elaborado pelo autor

Corretude

Observe que a base da recursão está bem definida, pois um vetor com menos de dois elementos já está ordenado. Note também que, em cada chamada, o vetor é dividido aproximadamente ao meio, e cada metade é ordenada recursivamente, sem deixar lacunas antes, entre ou depois de cada uma. Por fim, veja que a função Merge copia $v[l..r-1]$ para $v\_aux[l..r-1]$ e depois traz os elementos de volta para v, um por um, sempre escolhendo o menor que ainda não foi copiado. Como cada uma das etapas está correta, o Mergesort também está correto.

Desempenho

Seja $T(n)$ o tempo necessário para ordenar um vetor de n elementos utilizando o Mergesort. Temos então que:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ \underbrace{\Theta(1)}_{\text{Divisão}} + \underbrace{2T(n/2)}_{\text{Conquista}} + \underbrace{\Theta(n)}_{\text{Intercalação}}, & n > 1 \end{cases}$$

Desta forma, podemos simplesmente dizer que para $n > 1$ e alguma constante c,

$$T(n) \;=\; 2T(n/2) + cn \quad \Rightarrow \quad T(n) \;=\; n\log_2 n + cn$$

Portanto, o consumo de tempo do Mergesort é $\Theta(n \log_2 n)$. Já o de memória é $\Theta(n)$, devido ao vetor auxiliar utilizado.

4.3 Heapsort

O algoritmo Heapsort foi introduzido por J. W. J. Williams em 1964, no artigo *Algorithm 232: Heapsort*, publicado na revista *Communications of the ACM*. Como o próprio nome sugere, este algoritmo faz uso de uma estrutura de dados chamada *heap*, mais especificamente um *max-heap* binário.

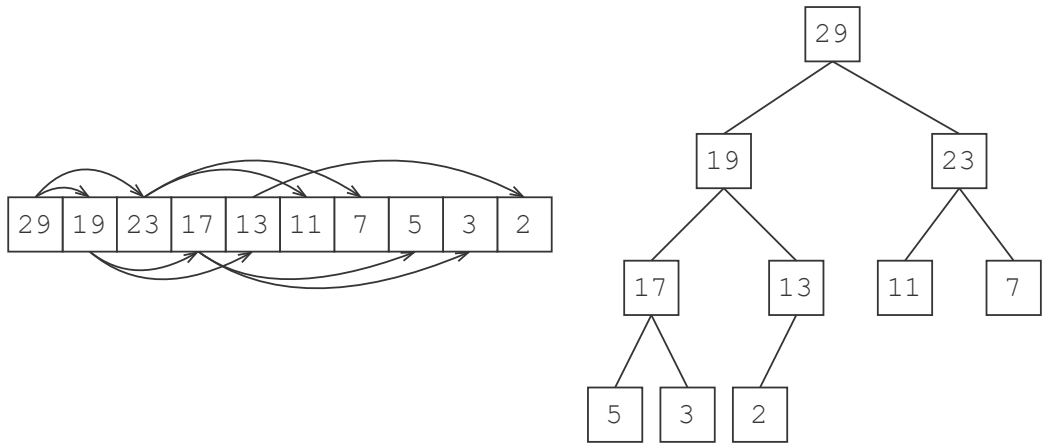
max-heap binário

Um *max-heap* binário – de agora em diante referido apenas como *heap* – é uma árvore binária quase completa que satisfaz a propriedade de que o valor armazenado em um nó pai é maior ou igual ao valor de seus filhos. Uma árvore binária é considerada

quase completa quando todos os seus níveis estão completamente preenchidos, exceto possivelmente o último, que deve ser preenchido da esquerda para a direita.

Convenientemente, essa árvore pode ser implementada em um vetor, de modo que cada posição do vetor represente um nó da árvore. Os filhos do nó na posição i estão localizados nas posições $2i+1$ e $2i+2$, desde que essas posições sejam válidas.

Figura 5 – Exemplo de max-heap binário.



Fonte: Elaborado pelo autor.

Função Heapifica

A função Heapifica será usada tanto para construir o *heap* quanto para implementar o Heapsort em si. Ela é responsável por restaurar a propriedade de *heap* em um "quase *heap*". Um "quase *heap*" é uma árvore em que apenas a raiz pode estar violando a propriedade de *heap*. A ideia é simples: começando pela raiz, compara-se o nó atual com seu filho de maior valor; se o filho for maior, os dois são trocados de posição, e o processo é repetido com o novo nó até que a condição seja falsa ou uma folha seja atingida.

```
1  void Heapifica(int i, int n, int *v) {
2      int chave = v[i], j = 2 * i + 1;
3      while (j < n) {
4          if (j < n - 1 && v[j] < v[j + 1]) j++;
5          if (chave >= v[j]) break;
6          v[i] = v[j];
7          i = j;
8          j = 2 * j + 1;
9      }
10     v[i] = chave;
11 }
```

Construção do heap

Considere o problema de transformar um vetor em um *heap*. Para resolvê-lo, basta aplicar a operação Heapifica a todos os nós que não são folhas, da direita para a esquerda.

```
1 void ConstroiHeap(int n, int *v) {
2     for (int i = n / 2 - 1; i >= 0; i--)
3         Heapifica(i, n, v);
4 }
```

O algoritmo Heapsort

O Heapsort começa transformando o vetor de entrada em um *heap*. Após isso, o maior elemento estará na primeira posição. Sabendo disso, $v[0]$ é trocado com $v[n-1]$. Neste momento, o maior elemento está na posição correta, e $v[0..n-2]$ forma um "quase *heap*". Em seguida, a função Heapifica transforma $v[0..n-2]$ em um *heap*, e esse processo é repetido até que o *heap* tenha apenas um elemento.

```
1 void Heapsort(int n, int *v) {
2     ConstroiHeap(n, v);
3     for (int i = n - 1; i > 0; i--) {
4         Troca(v, v + i);
5         Heapifica(0, i, v);
6     }
7 }
```

Desempenho

A quantidade de operações que a função Heapifica executa quando invocada em um nó de altura h é proporcional a h. Além disso, existem exatamente $2^M/2^h$ nós com altura $h > 1$, onde $M = \lfloor \log_2 n \rfloor + 1$. Logo, a quantidade total de operações na construção do *heap* é dada por:

$$\sum_{h=2}^{M} h\left(\frac{2^M}{2^h}\right) \;=\; 2^M \cdot \sum_{h=2}^{M} h\left(\frac{1}{2}\right)^h \;<\; 2^M \cdot \sum_{h=1}^{\infty} h\left(\frac{1}{2}\right)^h \;=\; 2^{M+1} \;\approx\; 2n$$

Portanto, o *heap* é construído em tempo linear, $\mathcal{O}(n)$. Já a segunda parte do algoritmo, a partir da linha 3, executa aproximadamente

$$\sum_{i=1}^{n-1} \log_2 i \;=\; \log_2(n-1)! \;\leq\; n \log_2 n$$

operações e, portanto, todo o algoritmo consome tempo proporcional a $\mathcal{O}(n \log_2 n)$.

4.4 Quicksort

O Quicksort é um algoritmo de ordenação que provavelmente é o mais amplamente utilizado. A versão básica foi desenvolvida em 1960 por C. A. R. Hoare e, desde então, tem sido estudado por muitas pessoas. O Quicksort é popular porque não é difícil de implementar, é um bom algoritmo de ordenação de uso geral e, em muitos casos, consome menos recursos do que qualquer outro método de ordenação (SEDGEWICK, 1990, p.115, tradução nossa).

O algoritmo Quicksort é um método de ordenação não estável que funciona da seguinte maneira: particiona-se o vetor em dois segmentos, de modo que os menores elementos fiquem no primeiro e os maiores no segundo. Em seguida, o processo é repetido recursivamente em cada um dos segmentos, enquanto eles tiverem tamanho maior que um.

A escolha de um elemento pivô define quais elementos são considerados menores e quais são considerados maiores. As estratégias para escolha do pivô, bem como o processo de particionamento, são apresentados a seguir.

Particionamento

Existem versões básicas do Quicksort que definem como pivô um elemento localizado em um índice fixo do intervalo, como, por exemplo, sempre escolher um dos extremos. Essa estratégia pode levar à complexidade $\mathcal{O}(n^2)$ para entradas que já estejam ordenadas, além de empilhar cerca de n chamadas recursivas, o que pode causar um estouro de pilha. Contudo, a versão do Quicksort apresentada neste trabalho já incorpora algumas otimizações. A função Particiona, apresentada a seguir, escolhe o pivô de forma pseudoaleatória.

```
1 int Particiona(int l, int r, int *v) {
2     Troca(v + r, v + GeraNumeroAleatorioNoIntervalo(l, r));
3     int pivo = v[r], j = l;
4     for (int i = l; i < r; i++)
5         if (v[i] <= pivo) {
```

```
6               Troca(v + j, v + i);
7               j++;
8           }
9       v[r] = v[j]; v[j] = pivo;
10      return j;
11  }
```

Na implementação da função Particiona mostrada acima, a função GeraNumeroAleatorioNoIntervalo utiliza a função *rand*, presente na biblioteca padrão da linguagem C. O retorno da função Particiona é um índice j tal que

$$l \leq j \leq r \quad \text{e} \quad \begin{cases} v[j] = pivô \\ v[k] \leq pivô, \quad l \leq k < j \\ v[k] > pivô, \quad j < k \leq r \end{cases}$$

O algoritmo Quicksort

A implementação do algoritmo Quicksort apresentada a seguir é idêntica àquela encontrada em (FEOFILOFF, 2009, p.90). Ela traz duas melhorias em relação à implementação básica. A primeira consiste em sempre tratar primeiro o menor dos segmentos gerados pela função Particiona, garantindo que a altura da pilha de recursão seja, no máximo, $\log_2 n$. A segunda é a eliminação de uma das chamadas recursivas, que é transformada em uma iteração, convertendo o algoritmo em uma recursão de cauda.

```
1  void Quicksort(int n, int *v) {
2      QuicksortRec(0, n - 1, v);
3  }
4  void QuicksortRec(int l, int r, int *v) {
5      while (l < r) {
6          int j = Particiona(l, r, v);
7          if (j - l < r - j) {
8              QuicksortRec(l, j - 1, v);
9              l = j + 1;
10         } else {
11             QuicksortRec(j + 1, r, v);
12             r = j - 1;
13         }
14     }
15 }
```

Desempenho

O pior caso do Quicksort tem complexidade $\mathcal{O}(n^2)$ e ocorre quando a função Particiona retorna sempre, ou quase sempre, um índice j muito próximo de l ou r. Isso acontece, por exemplo, quando todos os elementos do vetor são iguais. Já o melhor caso tem complexidade $\mathcal{O}(n \log_2 n)$ e ocorre quando j está próximo do ponto médio entre l e r em todas as chamadas recursivas.

Em média, a complexidade desta versão aleatorizada do Quicksort é proporcional a $\mathcal{O}(n \log_2 n)$. Esse resultado pode ser demonstrado por meio do cálculo do valor esperado do número de comparações realizadas na linha 5 da função Particiona, conforme apresentado em (CORMEN *et al.*, 2009, p.180–184). Para isso, observe que a probabilidade de os elementos $v[i]$ e $v[j]$ serem comparados durante a execução do algoritmo é dada por:

$$P_{ij} = \frac{2}{j - i + 1}$$

Seja X a variável aleatória que representa a quantidade total de comparações feitas. Assim, sendo $k = j - i$, o valor esperado de X é:

$$\begin{aligned}
E[X] &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j - i + 1} \\
&= \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k+1} \\
&< \sum_{i=0}^{n-2} \sum_{k=1}^{n} \frac{2}{k} \\
&< 2(n-1) \ln n \\
&= \left(\frac{2(n-1)}{\log_2 \mathrm{e}} \right) \log_2 n
\end{aligned}$$

4.5 Métodos híbridos

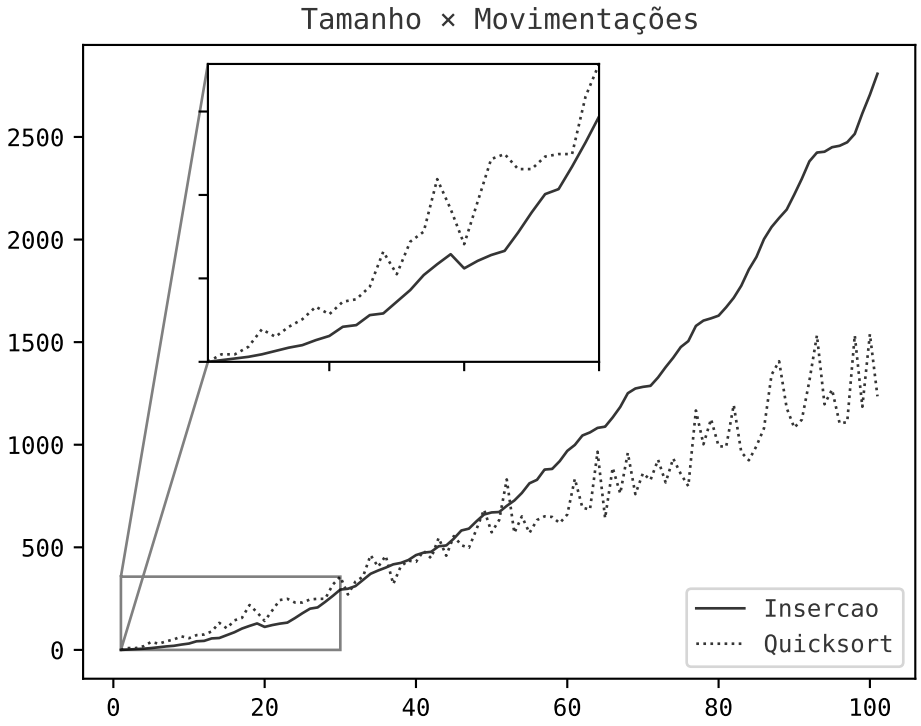
Esta seção apresenta dois algoritmos de ordenação híbridos, assim chamados por combinarem dois ou mais dos algoritmos discutidos anteriormente.

4.5.1 Quicksort com Inserção

Este algoritmo combina a eficiência do Quicksort para vetores grandes com a do Insercao para vetores pequenos e quase ordenados, como ilustra a Figura 6. Ele consiste

em duas etapas: a primeira é um Quicksort que ignora subvetores com tamanho menor ou igual a 16; a segunda é a aplicação do Insercao a todo o vetor. Observe que, ao final da primeira etapa, cada elemento estará, no máximo, a 16 posições de sua posição correta, de modo que o Insercao executará em tempo linear para finalizar a ordenação.

Figura 6 – Número médio de movimentações realizadas pelos algoritmos Quicksort e Insercao em vetores gerados de forma pseudoaleatória.



Fonte: Elaborado pelo autor

```
1  void QuicksortI(int n, int *v) {
2      QuicksortIRec(0, n - 1, v);
3      Insercao(n, v);
4  }
5  void QuicksortIRec(int l, int r, int *v) {
6      while (r - l > QS_LIMITE) {
7          int j = Particiona(l, r, v);
8          if (j - l < r - j) {
9              QuicksortIRec(l, j, v);
10             l = j + 1;
11         } else {
12             QuicksortIRec(j, r, v);
13             r = j - 1;
14         }
```

```
15      }
16  }
```

4.5.2 Introsort

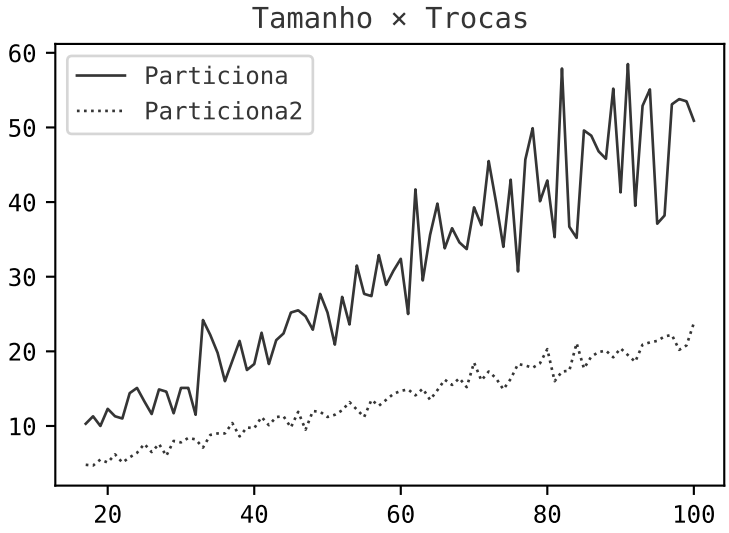
O algoritmo Introsort foi introduzido em (MUSSER, 1997) e é atualmente utilizado na implementação do método *sort* da biblioteca padrão da linguagem C++.

Outro método de particionamento

O Introsort controla a altura da pilha de recursão de maneira diferente daquela utilizada no Quicksort, o que possibilita o uso de outro método de particionamento que, como mostra a Figura 7, realiza menos trocas. Além disso, o elemento pivô é escolhido de forma distinta. A função MoveMedianaFim posiciona, na última posição do vetor, a mediana entre os elementos $v[l]$, $v[(l+r)/2]$ e $v[r]$. Essa estratégia de escolha do pivô é conhecida como "mediana de três".

```
1  int Particiona2(int l, int r, int *v) {
2      MoveMedianaFim(l, r, v);
3      int pivot = v[r], i = l - 1, j = r + 1;
4      while (1) {
5          do --j; while (v[j] > pivot);
6          do ++i; while (v[i] < pivot);
7          if (i >= j) break;
8          Troca(v + i, v + j);
9      }
10     return j;
11 }
```

Figura 7 – Número médio de trocas realizadas pelos métodos Particiona e Particiona2 em vetores gerados de forma pseudoaleatória.



Fonte: Elaborado pelo autor

O algoritmo Introsort

A ideia consiste em evitar o pior caso do Quicksort utilizando o Heapsort quando o tamanho da pilha de recursão atinge o valor $\lfloor 2\log_2 n \rfloor$. Dessa forma, como o Heapsort possui complexidade no pior caso linearítmica, o pior caso do Introsort é $\mathcal{O}(n \log_2 n)$. Além disso, assim como no algoritmo anterior, o Insercao é utilizado para concluir a ordenação.

```
1  void Introsort(int n, int *v) {
2      IntrosortRec(0, n - 1, log2(n) * 2, v);
3      Insercao(n, v);
4  }
5  void IntrosortRec(int l, int r, int d, int *v) {
6      while (r - l > QS_LIMITE) {
7          if (d == 0) {
8              Heapsort(r - l + 1, v + l);
9              return;
10         }
11         --d;
12         int j = Particiona2(l, r, v);
13         IntrosortRec(l, j, d, v);
14         l = j + 1;
15     }
16 }
```

5 ORDENAÇÃO EM TEMPO LINEAR

Este capítulo apresenta três métodos de ordenação que não se baseiam diretamente em comparações e que, sob determinados critérios, realizam a ordenação em tempo linear.

5.1 Countingsort

O algoritmo Countingsort baseia-se na contagem do número de ocorrências de cada elemento na entrada para determinar suas posições na saída. Foi introduzido por Harold H. Seward em 1954 e é particularmente eficiente quando os valores dos elementos a serem ordenados pertencem a um intervalo pequeno e conhecido. A implementação apresentada a seguir assume que o vetor de entrada não contém elementos negativos e que o maior elemento é suficientemente pequeno para permitir a criação de um vetor com esse tamanho.

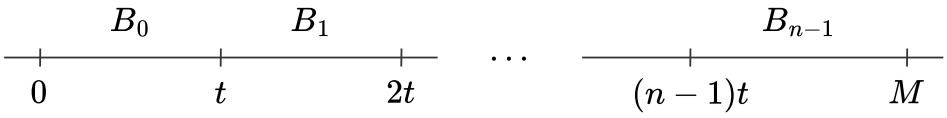
```
1  void Countingsort(int n, int *v) {
2      int *cont = AlocaVetorLimpo(MAX_EL + 1);
3      int *v_aux = AlocaVetor(n);
4      for (int i = 0; i < n; i++) {
5          cont[v[i]]++;
6      }
7      for (int i = 1; i <= MAX_EL; i++) {
8          cont[i] += cont[i - 1];
9      }
10     for (int i = n - 1; i >= 0; i--) {
11         v_aux[cont[v[i]] - 1] = v[i];
12         cont[v[i]]--;
13     }
14     for (int i = 0; i < n; i++) {
15         v[i] = v_aux[i];
16     }
17     free(cont);
18     free(v_aux);
19 }
```

5.2 Bucketsort

O Bucketsort é um algoritmo de ordenação descrito por John W. Tukey em 1944. Ele se baseia na distribuição dos elementos em diferentes grupos, chamados “buckets”,

cujos conteúdos são ordenados individualmente. Por fim, os buckets são concatenados, resultando em um vetor ordenado.

Figura 8 – Ilustração do intervalo correspondente a cada bucket, em que B_i representa o i-ésimo bucket e M é o maior elemento que pode existir no vetor.



$$t = \left\lceil \frac{M}{n} \right\rceil$$

$$x \in B_i \iff x \in [it,\ \ (i+1)t)$$

Fonte: Elaborado pelo autor.

Caso os elementos do vetor de entrada estejam distribuídos de forma uniforme no intervalo $[0, M]$, é esperado que cada bucket contenha uma quantidade constante de elementos, de modo que o algoritmo Bucketsort execute em tempo linear. No entanto, não se sabe previamente quantos elementos cada bucket armazenará durante a execução. Por isso, cada bucket foi implementado como uma estrutura de crescimento dinâmico, utilizando listas ligadas. Além disso, o método InsereOrdenado foi desenvolvido de forma semelhante ao algoritmo Insercao.

```
1  void Bucketsort(int n, int *v) {
2      Celula **bucket = (Celula **) malloc(n * sizeof(Celula*));
3      double t = ceil((MAX_EL + 1) / (double) n);
4      for (int i = 0; i < n; i++) {
5          bucket[i] = NULL;
6      }
7      for (int i = n - 1; i >= 0; i--) {
8          int j = (int) floor(v[i] / t);
9          InsereOrdenado(v[i], &bucket[j]);
10     }
11     for (int i = 0, k = 0; i < n; i++) {
12         Celula *atual = bucket[i];
13         while (atual != NULL) {
14             v[k++] = atual->valor;
15             Celula *temp = atual;
```

```
16              atual = atual->prox;
17              free(temp);
18          }
19      }
20      free(bucket);
21  }
```

5.3 Radixsort

O algoritmo Radixsort é um método de ordenação estável para números naturais, que funciona da seguinte maneira: dado um vetor de n números naturais com d dígitos cada, utiliza-se um algoritmo de ordenação estável para ordená-los pelo dígito menos significativo, depois pelo segundo menos significativo e assim por diante. O Radixsort depende da estabilidade do algoritmo auxiliar para garantir a correção do processo. Se existirem k valores possíveis para cada dígito, e o algoritmo auxiliar ordenar em tempo proporcional a $\mathcal{O}(n+k)$, então o Radixsort terá complexidade $\mathcal{O}(d(n+k))$, ou seja, complexidade linear.

A seguir, são apresentadas as duas versões do Radixsort que foram implementadas: uma utilizando uma adaptação do algoritmo Countingsort e outra, uma adaptação do Bucketsort. Como o objetivo deste trabalho é simplesmente ordenar números naturais em base decimal, a função Digito(x, p), utilizada em ambas as versões, assume que $p = 10^r$ e retorna $\lfloor x/p \rfloor \mod 10$, ou seja, o dígito de x que está na posição $r+1$, da direita para a esquerda.

Radixsort com Countingsort

A principal diferença em relação ao algoritmo Countingsort apresentado anteriormente é que o vetor de *cont* possui apenas 10 posições e armazena as ocorrências dos dígitos de 0 a 9, extraídos das posições indicadas por p nos elementos do vetor v.

```
1  void RadixsortC(int n, int *v) {
2      int *cont = AlocaVetor(10);
3      int *v_aux = AlocaVetor(n);
4      for (int p = 1; MAX_EL / p > 0; p *= 10) {
5          for (int i = 0; i < 10; i++) {
6              cont[i] = 0;
7          }
```

```
8           for (int i = 0; i < n; i++) {
9               cont[(v[i] / p) % 10]++;
10          }
11          for (int i = 1; i < 10; i++) {
12              cont[i] += cont[i - 1];
13          }
14          for (int i = n - 1; i >= 0; i--) {
15              int j = (v[i] / p) % 10;
16              v_aux[cont[j] - 1] = v[i];
17              cont[j]--;
18          }
19          for (int i = 0; i < n; i++) {
20              v[i] = v_aux[i];
21          }
22      }
23      free(cont);
24      free(v_aux);
25  }
```

Radixsort com Bucketsort

Em comparação com o algoritmo Bucketsort, a lista de buckets contém apenas 10 posições, uma para cada valor possível de dígito. Por exemplo, se o dígito indicado por p em $v[i]$ for igual a 5, então $v[i]$ é colocado no bucket 5. Observe que é suficiente inserir cada elemento diretamente no início de cada bucket, ou seja, como primeiro elemento da lista ligada, sem a necessidade de manter a lista ordenada.

```
1  void RadixsortB(int n, int *v) {
2      Celula **celula = (Celula**) malloc( n * sizeof(Celula*));
3      Celula **bucket = (Celula**) malloc(10 * sizeof(Celula*));
4      for (int i = 0; i < 10; i++) {
5          bucket[i] = NULL;
6      }
7      for (int i = 0; i < n; i++) {
8          celula[i] = (Celula *) malloc(sizeof(Celula));
9          celula[i]->prox = NULL;
10     }
11     for (int p = 1; MAX_EL / p > 0; p *= 10) {
12         for (int i = n - 1; i >= 0; i--) {
```

```
13              int j = (v[i] / p) % 10;
14              celula[i]->valor = v[i];
15              celula[i]->prox = bucket[j];
16              bucket[j] = celula[i];
17          }
18          for (int i = 0, k = 0; i < 10; i++) {
19              while (bucket[i] != NULL) {
20                  v[k++] = bucket[i]->valor;
21                  bucket[i] = bucket[i]->prox;
22              }
23          }
24      }
25      for (int i = 0; i < n; i++) {
26          free(celula[i]);
27      }
28      free(celula);
29      free(bucket);
30  }
```

6 ANÁLISE EMPÍRICA

Este capítulo tem como objetivo analisar empiricamente todos os algoritmos de ordenação apesentados anteriormente, comparando a análise teórica com os resultados obtidos na prática, no que diz respeito ao tempo de execução, número de comparações e número de movimentações.

6.1 Preliminares

Nesta seção, descreve-se o ambiente de desenvolvimento, indicando o hardware e os softwares utilizados na implementação dos algoritmos e na execução dos testes. Também é apresentada a organização do projeto em diretórios e arquivos. Além disso, são feitas definições necessárias para o melhor entendimento das seções seguintes. Por fim, descreve-se o que foi coletado e como essa coleta foi realizada.

6.1.1 Ambiente de desenvolvimento

Para a implementação dos algoritmos e execução dos testes foi utilizado o notebook descrito na Tabela 4. Todos os algoritmos e funções auxiliares foram implementados com a linguagem de programação C. As linguagens de programação Bash e Python foram utilizadas de forma secundária para automatização dos testes e geração de gráficos, respectivamente. O hardware, softwares e versões utilizados são especificados na tabela 4.

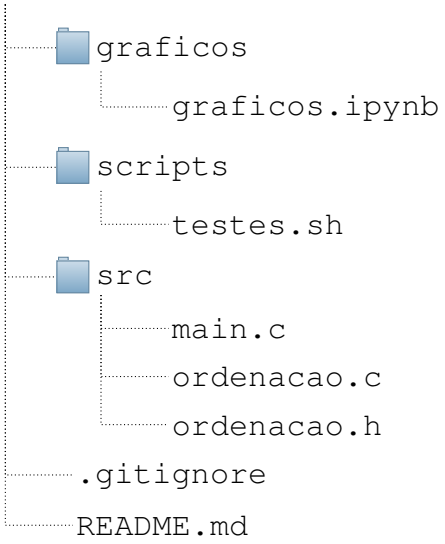
Tabela 4 – Hardware e softwares utilizados

Tipo	Componente	Produto
Hardware	Modelo	Avell A65i
	Processador	13th Gen Intel® Core™ i9-13900HX
	Cache	36 MB Intel® Smart Cache
	Memória principal	64.0 GiB
Software	Sistema operacional	Linux Ubuntu 24.04.3 LTS
	Editor de texto	Microsoft Visual Studio Code
	Compilador C	GNU Compiler Collection GCC
	Linguagens de programação	C, Bash e Python

6.1.2 Organização do código fonte

A figura 9 ilustra a forma como código fonte do projeto foi organizado, enquanto que a tabela 5 dá uma breve descrição de cada arquivo.

Figura 9 – Árvore de diretórios e arquivos do projeto



Fonte: Elaborado pelo autor. Acesso público em https://github.com/jbrenorv/ordenacao

Tabela 5 – Descrição dos arquivos do projeto

Arquivo	Descrição
graficos.ipynb	Arquivo exportado do notebook criado no Google Colab
testes.sh	Implementa script em Bash para execução automatizada dos testes
main.c	Implementa a função principal
ordenacao.c	Implementa os algoritmos de ordenação e funções utilitárias
ordenacao.h	Declara os protótipos dos algoritmos e funções utilitárias
.gitignore	Declara diretórios e arquivos que o Git deve ignorar
README.md	Documenta o projeto com instruções de execução dos testes

6.1.3 Tipos de vetores

Além do tamanho do vetor, a forma como os elementos estão dispostos inicialmente pode interferir no tempo e na quantidade de operações que cada algoritmo executa. Pensando nisso, com o objetivo de analisar como cada algoritmo se comporta para diferentes tipos de entrada, foram considerados três tipos de vetores nos testes, como mostra a tabela 6.

Tabela 6 – Tipos de vetores

Tipo	Descrição
1	Vetores já ordenados em ordem crescente
2	Vetores ordenados em ordem decrescente
3	Vetores gerados de forma pseudoaleatória

6.1.4 Dados coletados

Esta seção visa apenas apresentar e definir quais dados foram coletados. A próxima seção descreve com mais detalhes como a coleta foi realizada. Para cada combinação de algoritmo, tamanho e tipo de vetor, foi registrado o número de comparações, o número de movimentações e o tempo de execução. Essas três métricas são definidas na tabela 7.

Obter essas informações de cada algoritmo é importante para sabermos na prática como eles se comportam para diferentes configurações de entrada. Por exemplo, um algoritmo que executa muitas movimentações pode não ser adequado em um cenário em que mudar os registros de posição seja uma operação muito lenta.

Tabela 7 – Dados coletados e suas definições

Dado	Definição
Número de comparações	Uma comparação ocorre quando um elemento do vetor é comparado com outro elemento do vetor ou com outra variável do programa
Número de movimentações	Uma movimentação ocorre sempre que um valor é atribuído a um elemento do vetor ou um elemento do vetor é atribuído a uma outra variável
Tempo de execução	Tempo decorrido entre o início e o fim da execução do algoritmo em segundos

6.1.5 Metodologia de coleta

6.1.5.1 Entrada

A função principal é o ponto de entrada para qualquer programa escrito em linguagem de C. De uma forma mais técnica, ao compilar e linkar o código fonte do projeto, obtém-se um arquivo binário executável que, ao ser executado, invoca a função principal para iniciar a execução do processo. Essa função recebe dois argumentos quando invocada. O primeiro é um número inteiro comumente chamado de argc, que indica a quantidade

de parâmetros recebidos. Já o segundo é uma lista de vetores de caracteres comumente chamado de argv, que são os parâmetros em si. O primeiro parâmetro é sempre o nome do arquivo executável ou o caminho absoluto até ele. A tabela 8 indica os parâmetros adicionais esperados.

Tabela 8 – Parâmetros da função principal

Posição (argv)	Parâmetro esperado
0	Nome ou caminho para o arquivo executável
1	Nome do arquivo onde deve ser escrito os resultados
2	Número representando o tamanho do vetor
3	Número igual a 1, 2 ou 3, representando o tipo do vetor
4	Número representando número da execução

O último parâmetro, que indica o número da execução, é explicado na subseção 6.1.5.4, que apresenta como os testes foram executados.

6.1.5.2 Processamento

O processamento consiste na execução sequencial das etapas necessárias para realizar os experimentos definidos neste trabalho. O pseudocódigo apresentado a seguir oferece uma visão de alto nível do fluxo executado pela função principal, desde a leitura dos parâmetros de entrada até o registro dos resultados obtidos. Optou-se por essa forma de apresentação para destacar a lógica geral do procedimento e evitar a exibição detalhada de trechos de código que não contribuem significativamente para a compreensão do funcionamento do sistema.

Algoritmo 1: Função principal

Entrada: A tupla (S, N, T, E) representado os parâmetros listados na tabela 8

início

$F \leftarrow$ Abre arquivo S

$O \leftarrow$ Aloca vetor de tamanho N com valores de acordo com o tipo T

$V \leftarrow$ Aloca vetor de tamanho N

$L \leftarrow$ Cria lista de pares de algoritmos[a]

para *cada par* $(A, A') \in L$ **faça**

$V \leftarrow$ Cópia de O

$t \leftarrow$ Executa A em V[b]

$V \leftarrow$ Cópia de O[c]

$(c, m) \leftarrow$ Executa A' em V[d]

Adiciona em F uma linha com os resultados[e]

fim

Libera a memória alocada e fecha o arquivo F

fim

[a] O primeiro elemento é o algoritmo original e o segundo é a versão modificada para obter o número de comparações e movimentações.

[b] Obtendo o tempo de execução t em segundos.

[c] Neste ponto V está ordenado, então é necessário resetar para o vetor original.

[d] Obtendo o número de comparações c e movimentações m.

[e] Precisamente, a linha contêm o nome do algoritmo, o tamanho do vetor, o tipo do vetor, o número da execução, o número de comparações, o número de movimentações e o tempo de execução.

6.1.5.3 Exemplo de saída

Se após compilar e linkar o código C, o arquivo executável resultante se chamar a.out, o comando a ser executado a partir de um Terminal Linux, aberto no mesmo diretório do executável, poderia ser, por exemplo:

```
1 ./a.out saida.csv 1000 3 1
```

Neste caso, seria gerado um vetor de forma pseudoaleatória (tipo 3) de tamanho 1000 e os resultados seriam escritos no arquivo saida.csv. Com exceção do cabeçalho, o conteúdo do arquivo saida.csv é representado na tabela 9. Além disso, a coluna do tempo pode conter valores diferentes.

Tabela 9 – Exemplo de saída

Algo.	Tam.	Tipo	Exec.	Comp.	Movi.	Tempo (s)
Bolha	1000	2	1	499497	770538	0.004037
Coquetel	1000	2	1	388815	770538	0.003501
Selecao	1000	2	1	499500	2958	0.000543
Insercao	1000	2	1	257838	258844	0.000163
Shellsort	1000	2	1	13061	20296	0.000111
Mergesort	1000	2	1	8717	19302	0.000098
Heapsort	1000	2	1	8772	14040	0.000104
Quicksort	1000	2	1	10345	20325	0.000103
QuicksortI	1000	2	1	12107	17646	0.000072
Introsort	1000	2	1	14047	9620	0.000078
Countingsort	1000	2	1	0	2000	0.155960
Bucketsort	1000	2	1	1050	4050	0.000037
RadixsortC	1000	2	1	0	18000	0.000022
RadixsortB	1000	2	1	0	45000	0.000034

6.1.5.4 Testes

Neste ponto o código implementado em linguagem C está pronto para ser invocado com diferentes parâmetros de entrada. Para automatizar este processo, foi criado um script em linguagem Bash. Este script segue os seguintes passos:

1. Compilação do código C e inicialização do arquivo CSV de saída.
2. Geração dos tamanhos.
3. Três execuções para cada tamanho e tipo de vetor.

Optou-se por realizar três execuções por tamanho e tipo de vetor para reduzir o impacto de variações ocasionais no tempo de execução, como pequenas flutuações no uso da CPU. Esse número foi escolhido por ser suficiente para obter uma média representativa sem tornar a coleta de dados muito demorada.

Os tamanhos gerados são melhores descritos na tabela 10. Foram considerados ao todo 37 tamanhos entre 10^4 e 10^8, inclusive. Além disso, cada tamanho foi usado três vezes para cada um dos três tipos, totalizando 333 chamadas aos algoritmos.

Tabela 10 – Geração de tamanhos

Intervalo		Incremento	Subtotal	Total
Início	Fim			
10^4	$10^5 - 1$	10^4	9	37
10^5	$10^6 - 1$	10^5	9	
10^6	$10^7 - 1$	10^6	9	
10^7	10^8	10^7	10	

6.2 Métodos inferiores

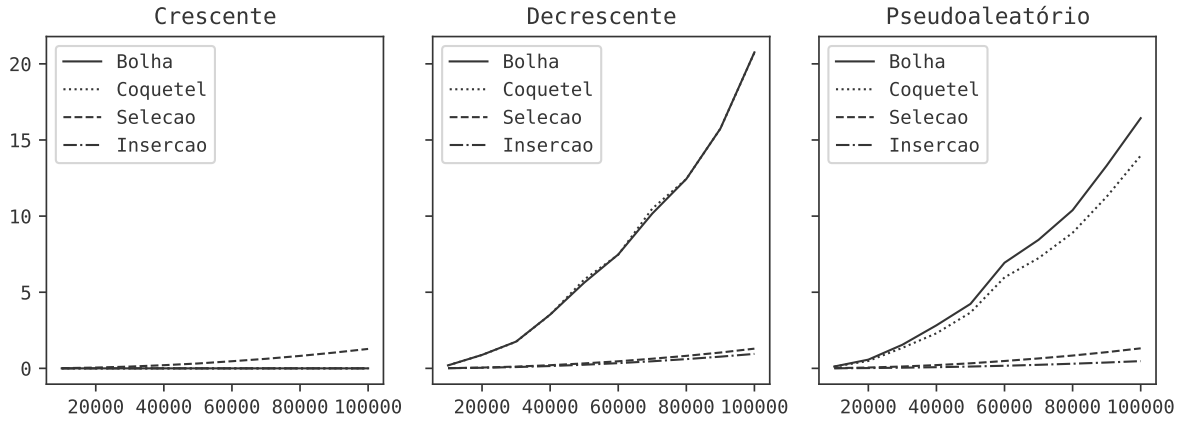
Os quatro algoritmos classificados neste trabalho como inferiores possuem, como já detalhado em capítulos anteriores, complexidade temporal quadrática no tamanho do vetor de entrada. Eles recebem essa denominação por apresentarem, de acordo com a análise de complexidade apresentada no Capítulo 3, essa ordem de crescimento. Entretanto, embora todos apresentem a mesma complexidade assintótica, os testes práticos demonstraram que eles possuem desempenhos distintos.

Como o tempo de execução desses algoritmos apresenta crescimento quadrático à medida que o tamanho da entrada aumenta, o tamanho máximo do vetor utilizado foi $n = 10^5$. Este tamanho foi considerado razoável para a observação dos resultados esperados, mantendo um tempo de execução viável para os testes.

6.2.1 Tempo de execução

O gráfico a seguir apresenta os resultados de tempo de execução obtidos ao se executar algoritmos de ordenação inferiores.

Figura 10 – Métodos inferiores – Tamanho × Tempo (em segundos).



Fonte: Elaborado pelo autor

Para vetores já ordenados em ordem crescente, como esperado, apenas o algoritmo Selecao chegou a consumir mais de $1s$. Todos os demais algoritmos executaram apenas uma iteração e concluíram suas execuções, o que resultou em um tempo de consumo praticamente nulo.

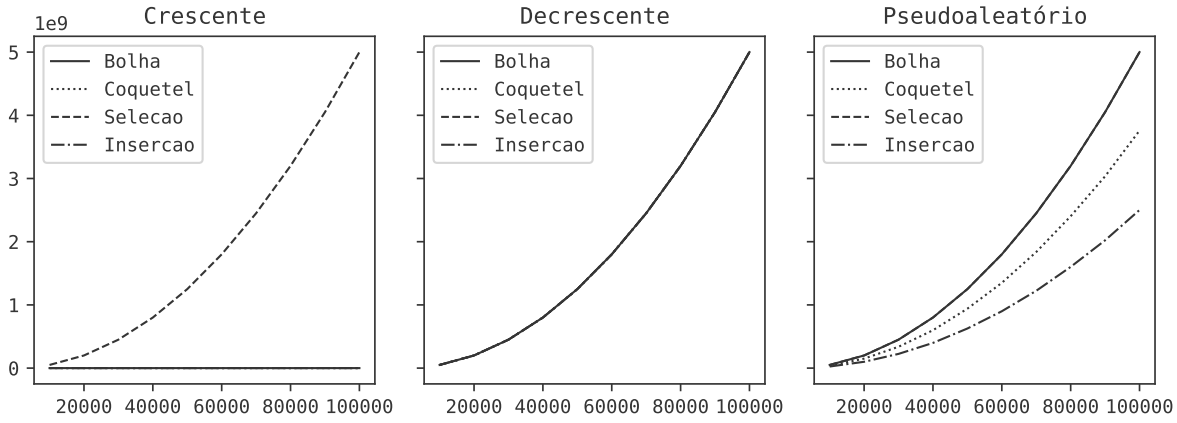
O pior caso de todos os algoritmos fica evidente para vetores que, inicialmente, estão em ordem decrescente. Curiosamente, no entanto, os algoritmos Bolha e Coquetel se mostraram significativamente mais lentos que o Insercao e o Selecao. O motivo para esse desempenho será detalhado nas próximas seções, onde comparamos o número de comparações e movimentações que cada algoritmo realizou para este tipo de vetor.

Por fim, buscando observar o comportamento geral dos algoritmos, analisamos os vetores gerados de forma pseudoaleatória. Os algoritmos Bolha e Coquetel apresentaram uma leve melhora de desempenho, sendo o Coquetel ligeiramente mais rápido. Isso ocorre porque a iteração de retorno do Coquetel permite que elementos menores atinjam suas posições finais mais rapidamente do que no Bolha. Já os algoritmos Selecao e Insercao foram significativamente mais rápidos, com o Insercao apresentando o melhor resultado geral. Vale notar que o Selecao manteve praticamente o mesmo tempo de execução, independentemente do tipo de vetor. Tal fato se deve à sua característica de sempre executar uma quantidade fixa e quadrática de comparações, enquanto realiza um número mínimo de trocas. Dessa forma, o número de movimentações do Selecao é desprezível no tempo final de execução, o que o torna uma boa escolha para casos em que o vetor possui poucos elementos e a operação de movimentação tem um custo computacional elevado.

6.2.2 Comparações

O gráfico da Figura 11 a seguir, exibe a relação entre o número de comparações e o tamanho do vetor para cada tipo de vetor analisado.

Figura 11 – Métodos inferiores – Tamanho × Comparações.



Fonte: Elaborado pelo autor

Para vetores em ordem crescente, como esperado, apenas o algoritmo Selecao executou uma quantidade quadrática de comparações. Os demais algoritmos, por outro lado, executaram uma quantidade linear de comparações, pois levaram apenas uma iteração externa para identificar que o vetor já se encontrava ordenado.

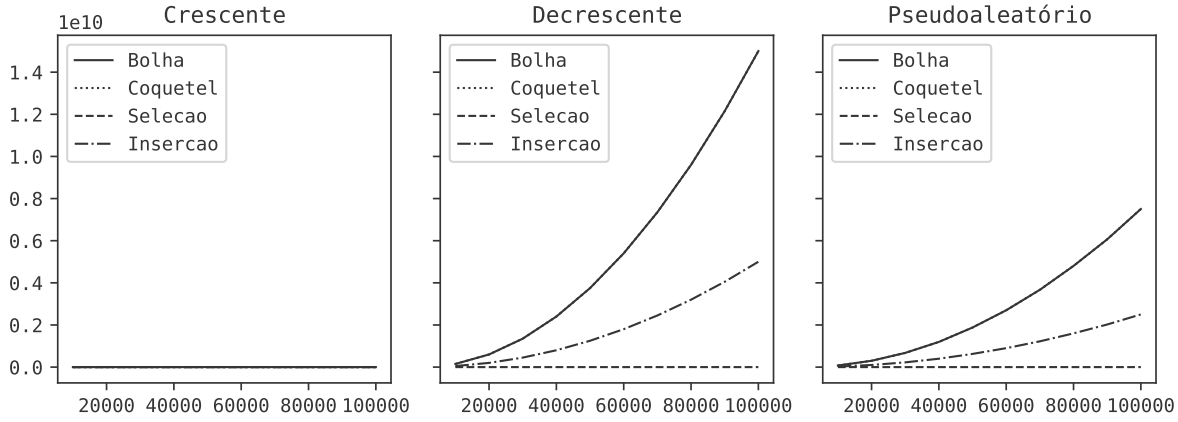
A análise teórica indicava que os algoritmos de ordenação inferiores apresentavam seu pior caso em comum: o vetor ordenado em ordem decrescente. Além disso, ela mostrava que esses algoritmos executam exatamente a mesma quantidade de comparações para esse tipo de vetor. Os testes práticos confirmaram essa conclusão, como é claramente demonstrado no gráfico.

No caso geral, com os vetores gerados de forma pseudoaleatória, pode-se observar que os algoritmos Bolha e Selecao permaneceram praticamente empatados em termos de número de comparações. O algoritmo Insercao demonstrou o melhor resultado, sendo aquele que necessitou do menor número de comparações para finalizar sua execução e garantir um vetor ordenado. Já o Coquetel ficou em um meio-termo, mostrando-se razoavelmente superior ao Bolha e ao Selecao nesse quesito. Sua vantagem reside no fato de que cada iteração externa coloca dois elementos em suas posições finais, além de aproximar os demais elementos de suas posições corretas em pelo menos uma posição.

6.2.3 Movimentações

O gráfico da Figura 12 a seguir, exibe a relação entre o número de movimentações e o tamanho do vetor para cada tipo de vetor analisado.

Figura 12 – Métodos inferiores – Tamanho × Movimentações.



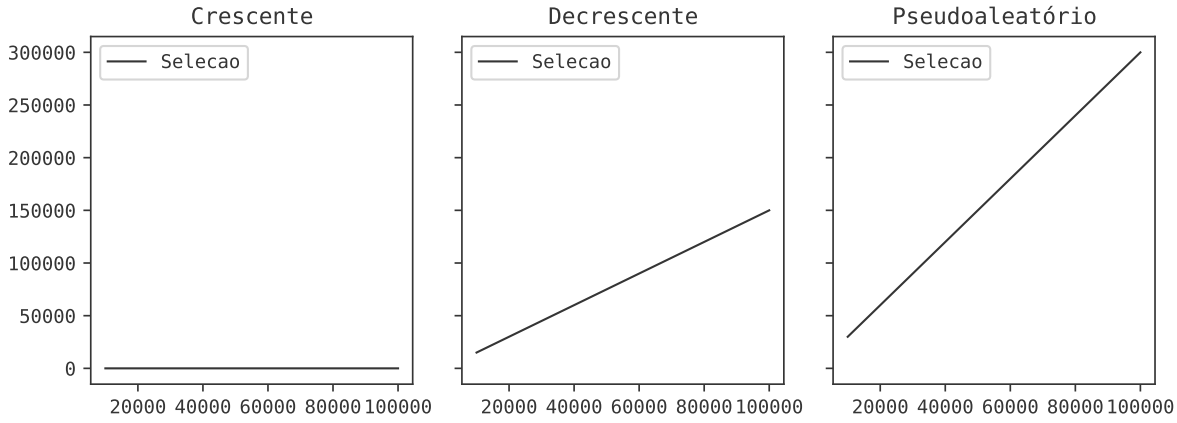
Fonte: Elaborado pelo autor

Observe que, na prática, independentemente do tipo do vetor, os algoritmos Bolha e Coquetel possuem o mesmo desempenho com relação ao número de movimentações.

Neste ponto, pode-se estabelecer uma boa explicação para o Insercao ter apresentado o menor tempo de execução. Dentre todos os algoritmos inferiores, ele é o que melhor consegue equilibrar o número de comparações e movimentações. Ou seja, ele não possui extremos; enquanto o Bolha e o Coquetel geralmente executam muitas comparações e movimentações, e o Selecao executa muitas comparações e poucas movimentações, o Insercao permanece em um meio-termo. Essa característica o favorece, resultando em um tempo de execução menor que os demais para todos os tipos de vetores.

Quando o assunto é número de movimentações, o Selecao é o algoritmo mais eficiente. Isso ocorre porque ele executa sempre o número mínimo de trocas, garantindo que cada troca posicione pelo menos um elemento em sua posição definitiva. Dessa forma, ele realiza $\mathcal{O}(n)$ movimentações, como fica evidente nos gráficos da Figura 13 para todos os tipos de vetores. É importante notar que cada troca executa 3 movimentações. Observe também que, para vetores decrescentes, cada troca posiciona dois elementos em suas posições finais.

Figura 13 – Selecao – Tamanho × Movimentações.



Fonte: Elaborado pelo autor

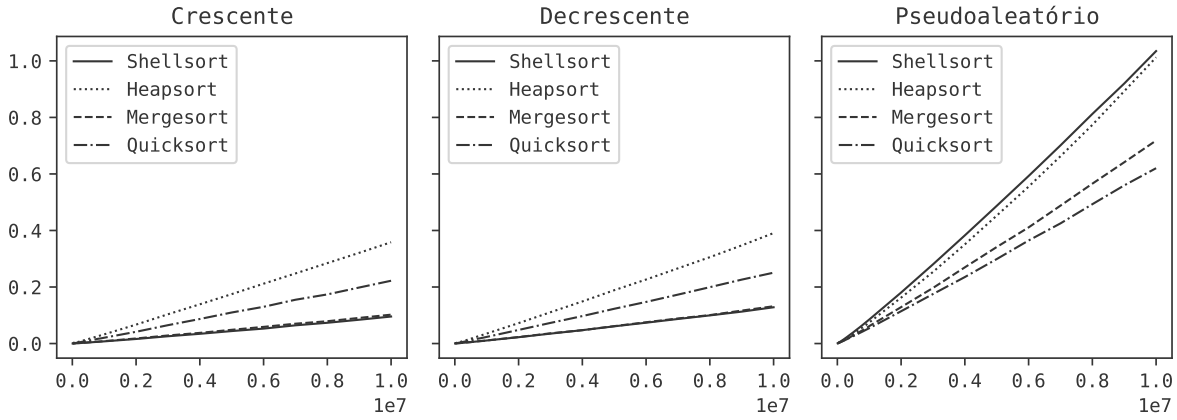
6.3 Métodos superiores

Os quatro algoritmos classificados como superiores são aqueles que possuem complexidade de tempo de ordem $\mathcal{O}(n \log n)$. Tal complexidade permite a realização de testes com vetores de tamanho consideravelmente maiores do que os utilizados para os algoritmos inferiores, razão pela qual foram testados com vetores cujo tamanho varia de 10^4 a 10^8.

6.3.1 Tempo de execução

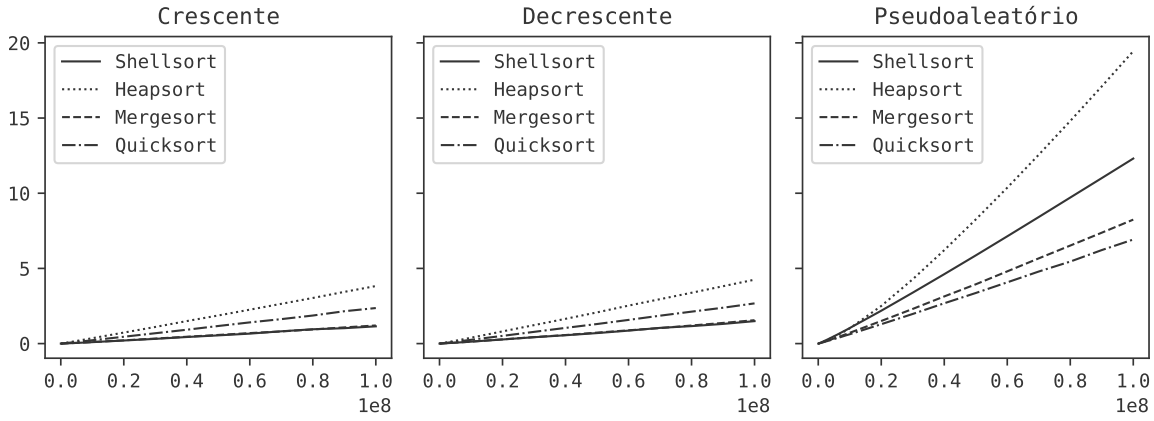
Foram gerados dois gráficos para apresentar os resultados de tempo de execução. O primeiro, na Figura 14, mostra os resultados com o tamanho dos vetores limitado a 10^7, e o segundo, na Figura 15, apresenta os tamanhos que atingem 10^8.

Figura 14 – Métodos superiores – Tamanho (até 10^7) × Tempo.



Fonte: Elaborado pelo autor

Figura 15 – Métodos superiores – Tamanho (até 10^8) × Tempo.



Fonte: Elaborado pelo autor

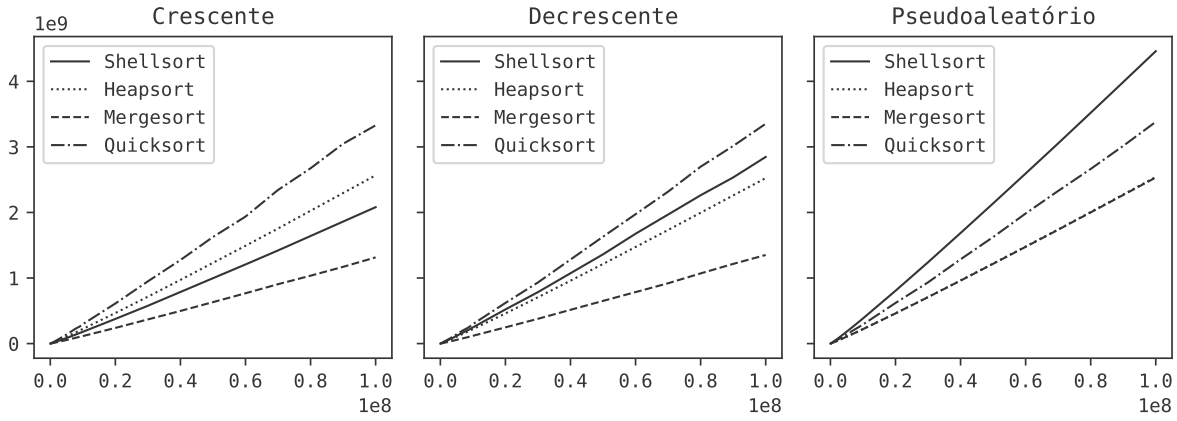
Para vetores inicialmente ordenados em ordem crescente ou decrescente, não há diferença significativa no comportamento entre a Figura 14 e a Figura 15. Os algoritmos mantêm a mesma relação de desempenho entre si. Nesses casos, o Heapsort e o Quicksort apresentam seus piores cenários de execução, enquanto o Shellsort e o Mergesort se beneficiam da presença de ordem nos dados.

Já no caso geral, em ambos os gráficos, o Quicksort apresenta o melhor resultado, sendo seguido pelo Mergesort. No entanto, observa-se um comportamento atípico no Heapsort. Para vetores com tamanhos até 10^7, como esperado pela análise teórica de complexidade, o Heapsort supera o Shellsort. Porém, para tamanhos maiores que 10^7, o Heapsort se torna ligeiramente mais lento que todos os outros. Este fato será explicado na subseção 6.3.4.

6.3.2 Comparações

O gráfico da Figura 16 a seguir, exibe a relação entre o número de comparações e o tamanho do vetor para cada tipo de vetor analisado.

Figura 16 – Métodos superiores – Tamanho × Comparações.



Fonte: Elaborado pelo autor

Em contraste com os demais, o Quicksort e o Heapsort demonstraram estabilidade em seu desempenho. Em todos os tipos de vetor, os gráficos de ambos os algoritmos são praticamente idênticos, o que indica que a ordem inicial dos dados não os beneficia nem os prejudica. No entanto, o Heapsort executa um número ligeiramente menor de comparações do que o Quicksort.

Já o Shellsort apresentou um desempenho variável conforme o tipo de vetor. Apesar de se beneficiar quando há presença de ordem nos dados, ele foi, no caso geral, o algoritmo superior que mais executou comparações.

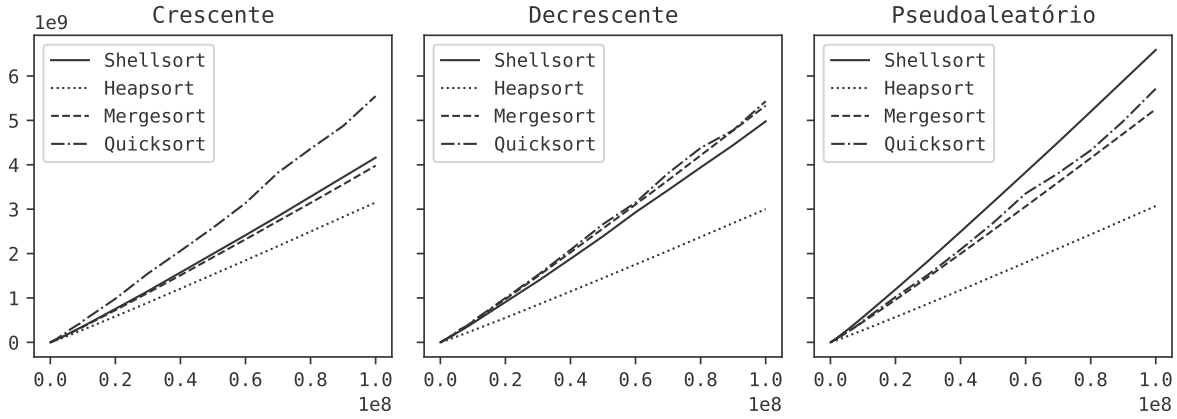
O Mergesort executa o menor número de comparações para todos os três tipos de vetor analisados, destacando-se em vetores já ordenados. Há uma explicação interessante para este fato: o algoritmo Mergesort só executa comparações durante o processo de Merge, e este processo se torna determinístico para vetores ordenados. Esse trecho do algoritmo é mostrado abaixo.

```
1  // Merge dos subvetores v[l..m-1] e v[m..r-1]
2  i = l, j = m, k = l;
3  while (i < m && j < r) {
4      // Se originalmente v[l..r] era crescente ou decrescente
5      // a escolha entre 'if' e 'else' se torna determinística
6      if (v_aux[i] <= v_aux[j]) {
7          v[k++] = v_aux[i++];
8      } else {
9          v[k++] = v_aux[j++];
10     }
11 }
```

6.3.3 Movimentações

O gráfico da Figura 17 a seguir, exibe a relação entre o número de movimentações e o tamanho do vetor para cada tipo de vetor analisado.

Figura 17 – Métodos superiores – Tamanho × Movimentações.



Fonte: Elaborado pelo autor

Com relação à quantidade de movimentações, o algoritmo Heapsort se destacou. Ele manteve o mesmo desempenho em todos os tipos de vetor, mas executou significativamente menos movimentações que os demais algoritmos superiores. O Quicksort também teve desempenho estável para todos os tipos de vetores, sendo ele o que mais executa movimentações para entradas já ordenadas de alguma forma. Já o Mergesort apresentou o mesmo desempenho em vetores decrescentes e no caso geral, mas executou menos movimentações para vetores crescentes, o que ocorre pelo mesmo motivo explicado na subseção anterior. De modo geral, o Shellsort é o que mais realiza a operação de movimentação de elementos do vetor de entrada.

6.3.4 Eficiência de cache

Conforme introduzido na seção 2.4, o processador tenta realizar a leitura e a escrita primeiramente no cache. Somente quando a informação de interesse não está presente em nenhum nível de cache, é que ele a busca na memória principal. Devido aos tamanhos de vetores utilizados neste trabalho, não é necessário mencionar a memória secundária, uma vez que toda a informação cabe na memória principal. A eficiência de cache de um algoritmo é, portanto, a taxa de sucesso na busca por elementos no cache, visto que esse

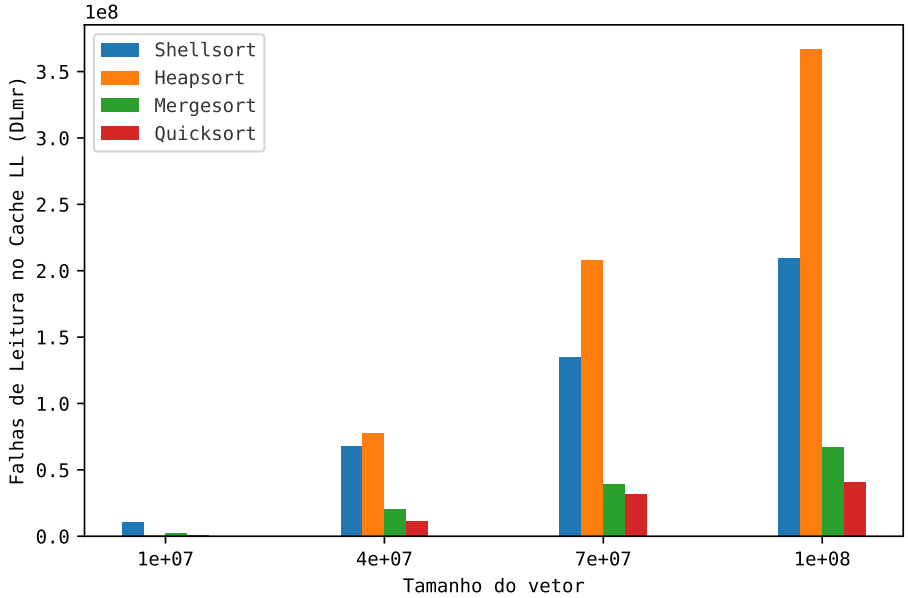
acesso é muito mais rápido do que na RAM.

6.3.4.1 Falhas de leitura

O gráfico da Figura 18 mostra a frequência em que o processador falhou ao tentar ler um dado no último nível de cache para cada algoritmo. Observe que, para vetores de tamanhos até 10^7, e dadas as propriedades do computador utilizado nos testes, praticamente todas as leituras resultavam em sucesso. No entanto, para vetores maiores que isso, as falhas se tornaram visíveis, com o Heapsort sendo o algoritmo que mais apresentou falhas de leitura.

A explicação para a ineficiência do Heapsort reside no princípio da localidade de referência espacial, conforme detalhado na subseção 2.4.2. Isso ocorre porque, à medida que o algoritmo desce pelos nós do Heap, o índice de cada posição acessada é pelo menos o dobro do índice da posição acessada anteriormente. Essa característica força o carregamento de elementos novos e distantes no cache, os quais se tornam inúteis já na próxima iteração.

Figura 18 – Métodos superiores – Falhas de leitura no último nível de cache.



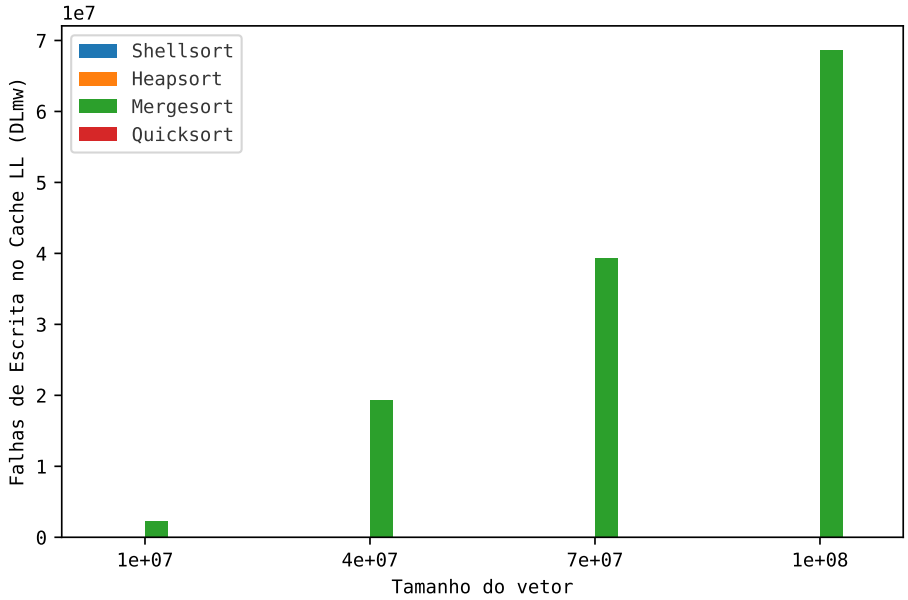
Fonte: Elaborado pelo autor

6.3.4.2 Falhas de escrita

O gráfico da Figura 19 mostra a frequência em que o processador falhou ao tentar escrever um dado no último nível de cache para cada algoritmo. Percebe-se que, mais uma vez, a quantidade de falhas é insignificante para vetores menores que 10^7. Constata-se

também que apenas o Mergesort causa falhas desse tipo. Este fato é explicado pela necessidade deste algoritmo de usar memória auxiliar.

Figura 19 – Métodos superiores – Falhas de escrita no último nível de cache.



Fonte: Elaborado pelo autor

6.3.5 Síntese dos resultados

A Table 11 a seguir classifica os algoritmos superiores por tempo, comparações, movimentações e uso do cache, no caso geral.

Tabela 11 – Métodos superiores – Classificação de desempenho por métrica

Ranking	Tempo	Comp.	Movi.	Leitura	Escrita
1º	Quicksort	Mergesort	Heapsort	Quicksort	Shellsort
2º	Mergesort	Heapsort	Mergesort	Mergesort	Heapsort
3º	Shellsort	Quicksort	Quicksort	Shellsort	Quicksort
4º	Heapsort	Shellsort	Shellsort	Heapsort	Mergesort

O Quicksort, apesar de não ser o algoritmo que realiza menos comparações ou movimentações, é o mais eficiente em termos de cache. Essa eficiência o estabelece como o melhor algoritmo superior para a principal métrica, o tempo. Já o Shellsort é o que mais executa comparações e movimentações, mas ainda assim acaba sendo mais rápido que o Heapsort, também devido à sua eficiência de cache. Por fim, o Mergesort é quase tão bom quanto o Quicksort, sendo penalizado pelo uso de memória extra.

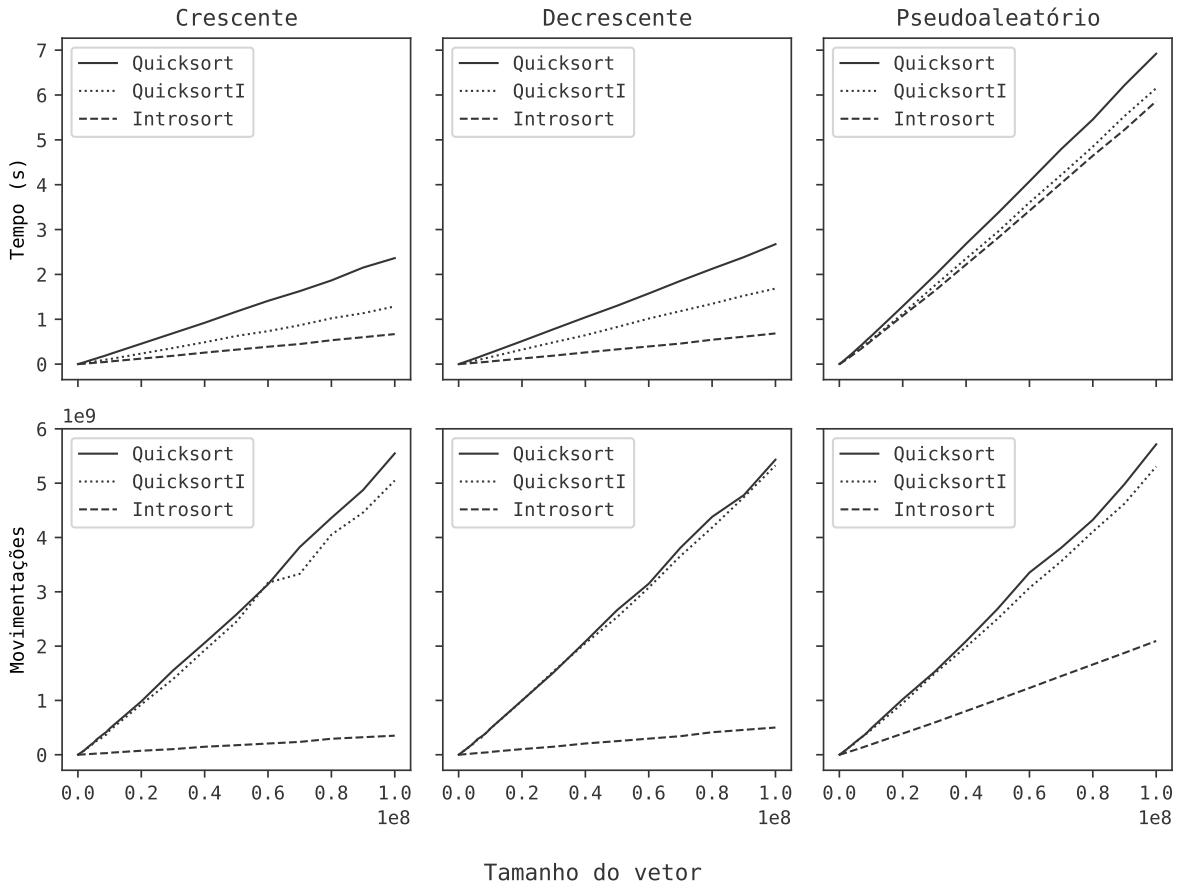
6.4 Quicksort e métodos híbridos

Os algoritmos de ordenação mais utilizados na prática são híbridos, baseados no Quicksort. Tais algoritmos se beneficiam dos pontos positivos do Quicksort ao mesmo tempo em que tentam diminuir o número de movimentações realizadas e evitar o pior caso.

No capítulo anterior, foram apresentados dois algoritmos desse tipo: o QuicksortI, que utiliza a Inserção para finalizar a ordenação, reduzindo o número de movimentações; e o Introsort, algoritmo utilizado na biblioteca padrão da linguagem de programação C++.

O Introsort também utiliza a Inserção para finalizar a ordenação, mas usa o Heapsort quando a árvore de recursão atinge altura igual a $2 \log n$, contornando assim os casos em que o Quicksort tende ao pior cenário. O gráfico da Figura 20 a seguir mostra os resultados de tempo e movimentações do Quicksort em comparação com o QuicksortI e o Introsort. Os resultados com relação ao número de comparações foram equivalentes e, por isso, não serão mostrados.

Figura 20 – Métodos híbridos – Tempo e movimentações.

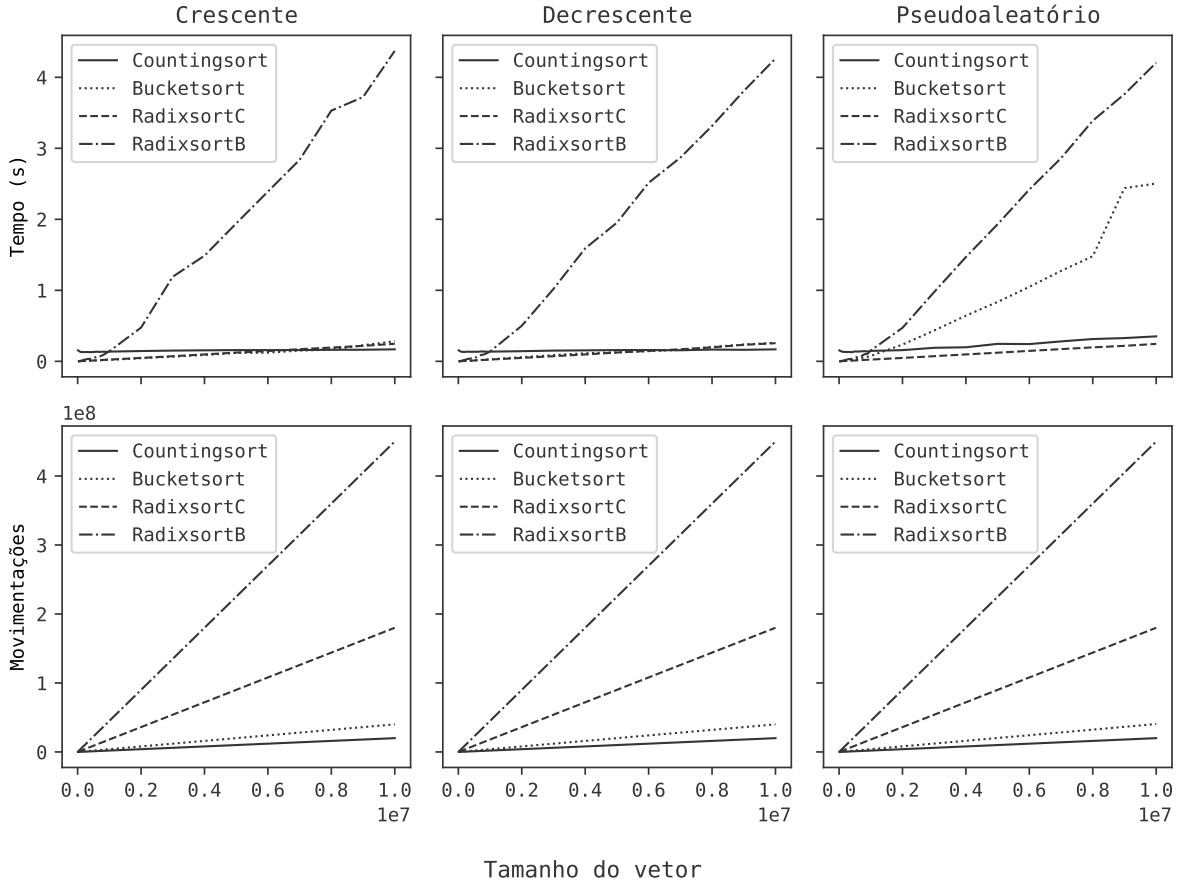


Fonte: Elaborado pelo autor

6.5 Métodos lineares

A subseção 2.3.2 mostrou que a cota inferior para algoritmos baseados em comparação é $\Omega(n \log_2 n)$. Isso significa que, para um algoritmo ser linear, ele não pode se basear em comparação. Além disso, os algoritmos lineares de ordenação apresentados neste trabalho só são lineares para entradas que obedecem a certas propriedades. Por exemplo, o Countingsort assume que os elementos da entrada podem ser indexados e que a diferença entre o maior e o menor elemento é limitada por uma constante suficientemente pequena para caber na memória. Já o Bucketsort requer que os elementos estejam uniformemente distribuídos em um intervalo. Por fim, o Radixsort assume que o algoritmo auxiliar é estável e linear e que cada elemento da entrada pode ser decomposto em dígitos pertencentes a um conjunto limitado e ordenável. Todos esses pré-requisitos foram levados em consideração nos testes, e todos os vetores foram gerados de modo a manter a característica linear de cada algoritmo.

Figura 21 – Métodos lineares – Tempo e movimentações.



Fonte: Elaborado pelo autor

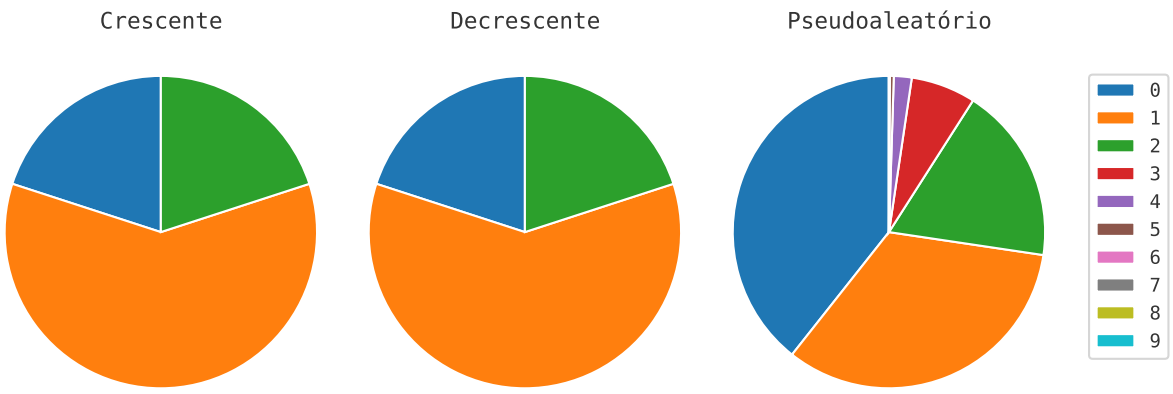
O primeiro ponto a se observar é que cada algoritmo apresenta consistência no número de movimentações. Isso ocorre porque cada um deles, independentemente do tipo de vetor, move os dados para a memória auxiliar e, em seguida, os traz de volta para o vetor de entrada, de modo a garantir a ordenação.

No entanto, ambas as versões do Radixsort — tanto a que usa o Countingsort como sub-rotina quanto a que usa o Bucketsort — executam mais movimentações. Isso se deve ao fato de que elas têm que realizar a movimentação para a memória auxiliar nove vezes, pois essa é a quantidade máxima de dígitos que cada número da entrada pode ter.

Por fim, observamos outros dois comportamentos que o gráfico não explica. O primeiro é que o RadixsortC foi mais rápido que o Countingsort, mesmo executando mais movimentações. Isso ocorre porque o RadixsortC utiliza menos memória que o Countingsort, uma vez que o vetor que armazenaria a frequência de cada elemento da entrada no Countingsort aloca 10^8 posições, enquanto que no RadixsortC aloca apenas 10.

O segundo é que o Bucketsort se tornou consideravelmente mais lento para vetores gerados de forma pseudoaleatória. Isso ocorre porque, mesmo que os elementos tendam a ser igualmente prováveis na geração do vetor, não é o que ocorre exatamente na prática. Para vetores crescentes ou decrescentes, devido à forma como os elementos foram escolhidos na geração dos vetores, cada bucket recebeu entre 0 e 2 elementos, o que não sobrecarrega a versão do Insercao utilizada. Já para vetores gerados de forma pseudoaleatória, os buckets tiveram entre 0 e 9 elementos, como mostra a Figura 22. Esse desequilíbrio fez com que o Insercao precisasse executar mais operações na lista ligada utilizada, o que tornou o algoritmo mais lento nesses casos.

Figura 22 – Bucketsort – Frequência de tamanho de bucket (10^7 elementos)



Fonte: Elaborado pelo autor

7 CONCLUSÃO

- Falar sobre o que FOI feito neste trabalho

REFERÊNCIAS

CIURA, M. Best increments for the average case of shellsort. In: SPRINGER. **International Symposium on Fundamentals of Computation Theory**. [S.l.], 2001. p. 106–117.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3rd. ed. Cambridge, MA: MIT Press, 2009. ISBN 978-0-262-03384-8.

FEOFILOFF, P. **Algoritmos em linguagem C**. [S.l.]: Elsevier/Campus, 2009.

HIBBARD, T. N. An empirical study of minimal storage sorting. **Communications of the ACM**, ACM New York, NY, USA, v. 6, n. 5, p. 206–213, 1963.

KNUTH, D. E. The art of computer programming, volume 3: Searching and sorting. **Addison-Westley Publishing Company: Reading, MA**, 1973.

KNUTH, D. E. **The Art of Computer Programming, Volume 3: Sorting and Searching**. 2nd. ed. Reading, Massachusetts: Addison-Wesley, 1998. ISBN 978-0201896855.

MUSSER, D. R. Introspective sorting and selection algorithms. **Software: Practice and Experience**, Wiley Online Library, v. 27, n. 8, p. 983–993, 1997.

SEDGEWICK, R. A new upper bound for shellsort. **Journal of Algorithms**, Elsevier, v. 7, n. 2, p. 159–173, 1986.

SEDGEWICK, R. **Algorithms in C**. 1. ed. [S.l.]: Addison-Wesley Professional, 1990. (Computer Science Series). ISBN 0201514257,9780201514254.

SHELL, D. L. A high-speed sorting procedure. **Communications of the ACM**, ACM New York, NY, USA, v. 2, n. 7, p. 30–32, 1959.

TOKUDA, N. An improved shellsort. In: **Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I**. NLD: North-Holland Publishing Co., 1992. p. 449–457. ISBN 044489747X.

VIANA, G. V. R.; CINTRA, G. F. Pesquisa e ordenação de dados. **Fortaleza: Publicação do Sistema UAB/UECE**, 2011.