---

7. CONCLUSÃO FINAL

Este experimento demonstrou claramente a importância da escolha do algoritmo correto:

- **BubbleSort**: 88x mais lento que QuickSort para 10.000 elementos
- **QuickSort**: Excelente desempenho com implementação cuidadosa
- **TimSort**: Melhor opção para uso geral em produção

A análise de complexidade temporal é fundamental para prever o comportamento de algoritmos com o crescimento dos dados. Para 10.000 elementos, a diferença entre $O(n^2)$ e $O(n \log n)$ resultou em uma diferença de desempenho de quase 2 ordens de grandeza.

Aprendizado Principal: A escolha do algoritmo correto pode ser a diferença entre um sistema que responde em milissegundos versus segundos, impactando diretamente a experiência do usuário e a escalabilidade da aplicação. # RELATÓRIO DE ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

Disciplina: Estrutura de Dados Orientada a Objetos
Data: 10/11/2025
Objetivo: Comparar o desempenho de algoritmos de busca e ordenação

---

1. EXECUÇÃO DO PROGRAMA

1.1 Print do Debugger Mostrando a Lista Sendo Ordenada

Screenshot do IntelliJ IDEA - Debugger em Ação:

Durante a execução do método `particionar()`, o debugger mostra o estado das variáveis:

```
Variáveis no Debugger:
- lista: ArrayList@962 (size = 10000)
- inicio: 0
- fim: 9999
- pivo: 73081 (último elemento)
- i: 0
- temp: 33678
- j: 0 (iterando de inicio até fim-1)
```

Breakpoint no método particionar() - Linha 128:

```
lista.set(j, temp);  // Executando troca de elementos
```

Call Stack mostrando a recursão:

```
particionar:128, App (br.com.pucgo)
quickSortRecursivo:107, App (br.com.pucgo)
main:35, App (br.com.pucgo)
```

Este screenshot demonstra o algoritmo QuickSort em funcionamento, mostrando: - A lista sendo particionada - O pivô sendo utilizado para comparação - As trocas de elementos acontecendo (temp = 33678) - A recursão ativa no call stack

---

1.2 Print da Execução Completa - Saída do Console

```
========================================
COMPARAÇÃO DE ALGORITMOS DE ORDENAÇÃO
========================================
Tamanho da lista: 10000
Exemplo de lista gerada: [92172, 32820, 33763, 88235, 82744, 17332, 87521, 11672, 64966, 964

--- BUBBLE SORT ---
Tempo BubbleSort: 563.126 ms
Lista ordenada (primeiros 10): [0, 2, 13, 21, 33, 51, 59, 63, 82, 87]...

--- QUICK SORT ---
Tempo QuickSort: 6.399 ms
Lista ordenada (primeiros 10): [0, 2, 13, 21, 33, 51, 59, 63, 82, 87]...

--- COLLECTIONS.SORT (TimSort) ---
Tempo Collections.sort: 10.318 ms
Lista ordenada (primeiros 10): [0, 2, 13, 21, 33, 51, 59, 63, 82, 87]...
```

1.3 Print do QuickSort Passo a Passo (QuickSortDebug)

Para melhor visualização do algoritmo em ação, foi criada uma versão com debug que mostra cada etapa:

```
     DEMONSTRAÇÃO DO QUICKSORT COM DEPURADOR


Lista Original: [64, 34, 25, 12, 22, 11, 90, 88, 45, 50]
Tamanho: 10 elementos

Iniciando ordenação...
```

```
Nível 0: Processando índices [0..9]
 Sublista: [64, 34, 25, 12, 22, 11, 90, 88, 45, 50]
 Pivô (último elemento): 50
 Particionando:
   → Troca: 64   34
   → Troca: 64   25
   → Troca: 64   12
   → Troca: 64   22
   → Troca: 64   11
   → Troca: 64   45
   → Troca pivô: 90   50
 Após particionamento: [34, 25, 12, 22, 11, 45, 50, 88, 64, 90]
 Pivô está na posição correta: índice 6 = 50
Dividindo em duas sublistas...

 ↓ Recursão à ESQUERDA do pivô
  Nível 1: Processando índices [0..5]
   Sublista: [34, 25, 12, 22, 11, 45]
   Pivô (último elemento): 45
   Particionando:
   Após particionamento: [34, 25, 12, 22, 11, 45]
   Pivô está na posição correta: índice 5 = 45
  Dividindo em duas sublistas...

    ↓ Recursão à ESQUERDA do pivô
     Nível 2: Processando índices [0..4]
      Sublista: [34, 25, 12, 22, 11]
      Pivô (último elemento): 11
      Particionando:
        → Troca pivô: 34   11
      Após particionamento: [11, 25, 12, 22, 34]
      Pivô está na posição correta: índice 0 = 11
     Dividindo em duas sublistas...

       ↓ Recursão à DIREITA do pivô
        Nível 3: Processando índices [1..4]
         Sublista: [25, 12, 22, 34]
         Pivô (último elemento): 34
         Particionando:
         Após particionamento: [25, 12, 22, 34]
         Pivô está na posição correta: índice 4 = 34
        Dividindo em duas sublistas...

          ↓ Recursão à ESQUERDA do pivô
           Nível 4: Processando índices [1..3]
```

```
        Sublista: [25, 12, 22]
        Pivô (último elemento): 22
        Particionando:
          → Troca: 25   12
          → Troca pivô: 25   22
        Após particionamento: [12, 22, 25]
        Pivô está na posição correta: índice 2 = 22
       Dividindo em duas sublistas...

↓ Recursão à DIREITA do pivô
 Nível 1: Processando índices [7..9]
  Sublista: [88, 64, 90]
  Pivô (último elemento): 90
  Particionando:
  Após particionamento: [88, 64, 90]
  Pivô está na posição correta: índice 9 = 90
 Dividindo em duas sublistas...

  ↓ Recursão à ESQUERDA do pivô
   Nível 2: Processando índices [7..8]
    Sublista: [88, 64]
    Pivô (último elemento): 64
    Particionando:
      → Troca pivô: 88   64
    Após particionamento: [64, 88]
    Pivô está na posição correta: índice 7 = 64
   Dividindo em duas sublistas...



  Ordenação Completa!
Lista Final: [11, 12, 22, 25, 34, 45, 50, 64, 88, 90]
```

Observações do Debug: 1. A cada nível de recursão, a lista é dividida em partições menores 2. O pivô é sempre o último elemento da sublista 3. Após o particionamento, elementos menores ficam à esquerda e maiores à direita 4. A profundidade máxima da recursão foi 4 níveis (para 10 elementos) 5. Cada pivô encontra sua posição final correta após o particionamento

---

2. TABELA DE COMPARAÇÃO DE DESEMPENHO

Algoritmo	Tempo (ms)	Complexidade Temporal	Complexidade Espacial
BubbleSort	563,126	$O(n^2)$	O(1)
QuickSort	6,399	O(n log n)*	O(log n)
Collections.sort	10,318	O(n log n)	O(n)

*QuickSort tem complexidade $O(n^2)$ no pior caso, mas O(n log n) no caso médio.

Análise dos Resultados:

Desempenho Relativo:

- **QuickSort** foi aproximadamente **88x mais rápido** que BubbleSort
- **Collections.sort (TimSort)** foi aproximadamente **54x mais rápido** que BubbleSort
- **QuickSort** foi aproximadamente **1,6x mais rápido** que Collections.sort neste teste

---

3. IMPLEMENTAÇÃO DO QUICKSORT RECURSIVO

Estratégia de Divisão e Conquista:

O QuickSort implementado utiliza a técnica de **divisão e conquista**:

1. **Divisão**: Escolhe um pivô (último elemento) e particiona a lista
2. **Conquista**: Ordena recursivamente as sublistas à esquerda e direita do pivô
3. **Combinação**: As sublistas ordenadas são naturalmente combinadas

Código Implementado:

```
private static void quickSort(List<Integer> lista) {
    quickSortRecursivo(lista, 0, lista.size() - 1);
}

private static void quickSortRecursivo(List<Integer> lista, int inicio, int fim) {
    if (inicio < fim) {
        int indicePivo = particionar(lista, inicio, fim);
        quickSortRecursivo(lista, inicio, indicePivo - 1);
        quickSortRecursivo(lista, indicePivo + 1, fim);
    }
}

private static int particionar(List<Integer> lista, int inicio, int fim) {
    int pivo = lista.get(fim);
```

```
    int i = inicio - 1;

    for (int j = inicio; j < fim; j++) {
        if (lista.get(j) <= pivo) {
            i++;
            int temp = lista.get(i);
            lista.set(i, lista.get(j));
            lista.set(j, temp);
        }
    }

    int temp = lista.get(i + 1);
    lista.set(i + 1, lista.get(fim));
    lista.set(fim, temp);

    return i + 1;
}
```

---

4. CONCLUSÕES

4.1 Qual algoritmo é mais rápido?

Resposta: O **QuickSort** foi o algoritmo mais rápido nos testes realizados, com tempo de **6,399 ms**, seguido pelo Collections.sort (TimSort) com **10,318 ms** e por último o BubbleSort com **563,126 ms**.

Justificativa: - QuickSort utiliza divisão e conquista, reduzindo significativamente o número de comparações - BubbleSort compara todos os elementos múltiplas vezes (quadrático) - TimSort (Collections.sort) é otimizado mas teve overhead ligeiramente maior neste teste específico

---

4.2 Qual possui menor complexidade?

Resposta: QuickSort e Collections.sort (TimSort) possuem a menor complexidade temporal no caso médio: **O(n log n)**.

Comparação de Complexidades:

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço
BubbleSort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
QuickSort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
TimSort	O(n)	O(n log n)	O(n log n)	O(n)

Observações: - BubbleSort tem complexidade quadrática, ineficiente para grandes volumes - QuickSort tem melhor desempenho médio, mas pode degradar para $O(n^2)$ em listas já ordenadas - TimSort garante O(n log n) no pior caso e é otimizado para dados parcialmente ordenados

---

4.3 Qual seria mais adequado para grandes volumes de dados?

Resposta: Collections.sort (TimSort) é o mais adequado para grandes volumes de dados em produção.

Justificativas:

1. **TimSort (Collections.sort):**
   - Complexidade garantida O(n log n) no pior caso
   - Altamente otimizado e testado pela Oracle/OpenJDK
   - Excelente desempenho em dados parcialmente ordenados
   - Estável (mantém ordem relativa de elementos iguais)
   - Requer espaço adicional O(n)
2. **QuickSort:**
   - Excelente desempenho médio
   - Baixo uso de memória O(log n)
   - Pode degradar para $O(n^2)$ sem otimizações (pivô aleatório, mediana de três)
   - Não é estável
3. **BubbleSort:**
   - Ineficiente para grandes volumes ($O(n^2)$)
   - Simples de implementar
   - Útil apenas para ensino ou listas muito pequenas

---

5. COMPARAÇÃO DE BUSCA

Busca Linear vs Binária

```
Número a ser buscado: 6387
Busca Linear  -> Encontrado: true | Tempo: 244100 ns (0,244 ms)
Busca Binária -> Encontrado: true | Tempo: 25600 ns  (0,026 ms)
```

Resultado: A busca binária foi aproximadamente **9,5x mais rápida** que a busca linear.

Complexidades: - **Busca Linear**: O(n) - precisa verificar cada elemento - **Busca Binária**: O(log n) - divide o espaço de busca pela metade a cada iteração - **Requisito**: A busca binária exige que a lista esteja ordenada

---

6. RECOMENDAÇÕES PRÁTICAS

Para Projetos Reais:

1. **Use Collections.sort()** para ordenação geral em Java
   - É otimizado, testado e mantido
   - Desempenho excelente na maioria dos casos
2. **Use Arrays.parallelSort()** para arrays muito grandes (> 100.000 elementos)
   - Aproveita múltiplos núcleos de CPU
3. **Evite BubbleSort** em produção
   - Use apenas para fins educacionais
4. **Implemente QuickSort customizado** apenas se:
   - Precisar de ordenação in-place com baixo uso de memória
   - Tiver controle sobre os dados de entrada
   - Puder implementar otimizações (pivô aleatório, insertion sort para sublistas pequenas)

Para Grandes Volumes:

- **< 100 elementos**: Qualquer algoritmo serve
- **100 - 10.000 elementos**: Collections.sort é ideal
- **> 10.000 elementos**: Collections.sort ou Arrays.parallelSort
- **Dados em disco**: Considerar algoritmos externos (Merge Sort externo)