Trabalho Prático I - Análise de Algoritmos de Ordenação

Disciplina: Projeto e Análise de Algoritmos
Professor: Danilo Medeiros Eler
Instituição: FCT/Unesp – Presidente Prudente

Alunos:

- Alexandre Malavazi
- Ryan Rocha

Descrição do Projeto

Este projeto implementa e analisa nove algoritmos de ordenação clássicos através de benchmark experimental extensivo, comparando seu desempenho em diferentes cenários (melhor caso, caso médio e pior caso) e validando as complexidades assintóticas teóricas com dados práticos.

Algoritmos Implementados

1. **Bubble Sort** (versão original sem melhorias)
2. **Bubble Sort Melhorado** (com verificação de ordenação)
3. **Insertion Sort** (inserção direta)
4. **Selection Sort** (seleção direta)
5. **Shell Sort**
6. **Heap Sort**
7. **Merge Sort**
8. **Quick Sort** (pivô no primeiro elemento)
9. **Quick Sort** (pivô no elemento central)

Instalação e Configuração

Pré-requisitos

- Python 3.7 ou superior
- pip (gerenciador de pacotes Python)

Instalação das Dependências

```
pip install matplotlib pandas numpy
```

Nota: As bibliotecas são necessárias apenas para o script `gerador_graficos_completo.py`. O arquivo `main_benchmark.py` usa apenas bibliotecas padrão do Python.

Estrutura do Projeto

```
IPLPAAV4/
│
├── Algoritmos/
│   ├── bubbleSortBasico.py
│   ├── bubbleSortComplexo.py
│   ├── insertionSort.py
│   ├── selectionSort.py
│   ├── shellSort.py
│   ├── heapSort.py
│   ├── mergeSort.py
│   ├── quickSortPrimeiro.py
│   └── quickSortMedio.py
│
├── graficos_benchmark/
│   ├── grafico_individual_*.png
│   ├── grafico_comparativo_*.png
│   ├── grafico_rapidos_*.png
│   └── resumo_tempos_medios.png
│
├── executar_tudo.py
├── main_benchmark.py
├── gerador_graficos_completo.py
├── teste_algoritmos.py
│
├── analise_assintotica.md
├── DOCUMENTACAO.pdf
```

```
├── README.md
│
└── resultados_benchmark_completo.csv
```

Como Executar

Opção 1: Execução Completa (Recomendada)

Execute o script principal que gerencia todo o processo automaticamente:

```
python executar_tudo.py
```

Fluxo automático:

1. Verifica dependências
2. Executa benchmark completo (2.000+ testes)
3. Gera todos os gráficos de análise
4. Cria relatórios comparativos

Opção 2: Execução em Etapas Manuais

```
python teste_algoritmos.py
python main_benchmark.py
python gerador_graficos_completo.py
```

Monitoramento de Progresso

O sistema inclui sistema completo de monitoramento com:

Barras de progresso visuais:

```
text
[====================] 100.0% (50/50)
```

Status em tempo real:

```
text
Bubble Sort Basico (1/9)
------------------------------------------------
   Aleatorio - Tamanho 1000: 0.0234s (5/5 ok)
```

```
   Crescente - Tamanho 1000: 0.0012s (5/5 ok)
   Decrescente - Tamanho 1000: 0.0456s (5/5 ok)
```

Relatório final detalhado:

```
BENCHMARK CONCLUIDO!
Tempo total: 25.3 minutos
Testes executados: 1215
```

Configurações do Benchmark

Parâmetros de Teste

Tamanhos de vetor testados:
1000, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000 elementos

Tipos de entrada:

- Aleatorio: Caso médio representativo
- Crescente: Melhor caso para vários algoritmos
- Decrescente: Pior caso para a maioria dos algoritmos

Sistema de repetições adaptativas:

- n ≤ 5000: 5 repetições
- 5000 < n ≤ 20000: 3 repetições
- n > 20000: 2 repetições

Estatísticas do Benchmark

- Total de algoritmos: 9
- Total de tamanhos: 9
- Total de tipos de entrada: 3
- Total de testes executados: ~1.215
- Tempo estimado: 15-45 minutos (depende do hardware)

Análise Detalhada dos Algoritmos

1. Bubble Sort Básico

Complexidade: $O(n^2)$ em todos os casos
Desempenho Observado:

- Consistentemente o mais lento
- Tempo médio para n=10.000: ~0.400s
- Crescimento quadrático claramente visível
- **Uso Prático**: Não recomendado para produção

2. Bubble Sort Melhorado

Complexidade: O(n) melhor caso, $O(n^2)$ caso médio/pior
Desempenho Observado:

- 2x mais rápido que versão básica
- Excelente em arrays ordenados (O(n))
- Detecta precocemente a ordenação
- **Uso Prático**: Educacional, arrays pequenos quase ordenados

3. Insertion Sort

Complexidade: O(n) melhor caso, $O(n^2)$ caso médio/pior
Desempenho Observado:

- Mais rápido entre $O(n^2)$ para entradas aleatórias
- Excelente para arrays pequenos (<1000 elementos)
- Performance linear em arrays ordenados
- **Uso Prático**: Arrays pequenos, quase ordenados, como parte de algoritmos híbridos

4. Selection Sort

Complexidade: $O(n^2)$ em todos os casos
Desempenho Observado:

- Performance consistente entre diferentes entradas
- Sempre $O(n^2)$, sem melhor caso
- Menos trocas que Bubble Sort
- **Uso Prático**: Educacional, quando trocas são custosas

5. Shell Sort

Complexidade: O(n log n) a O(n^(3/2))
Desempenho Observado:

- Performance próxima dos O(n log n)
- Bom compromisso entre simplicidade e eficiência
- Tempo médio para n=10.000: ~0.012s
- **Uso Prático**: Boa alternativa geral, fácil implementação

6. Heap Sort

Complexidade: O(n log n) em todos os casos
Desempenho Observado:

- Performance O(n log n) consistente
- Tempo médio para n=10.000: ~0.018s
- Ordenação in-place, sem memória extra
- **Uso Prático**: Quando memória é limitada, necessidade de O(n log n) garantido

7. Merge Sort

Complexidade: O(n log n) em todos os casos
Desempenho Observado:

- Performance O(n log n) estável e previsível
- Tempo médio para n=10.000: ~0.015s
- Algoritmo estável, usa memória extra O(n)
- **Uso Prático**: Quando estabilidade é necessária, ordenação externa

8. Quick Sort (Primeiro)

Complexidade: O(n log n) médio, $O(n^2)$ pior caso
Desempenho Observado:

- Muito rápido em entradas aleatórias (~0.009s para n=10.000)
- Degradação severa em entradas ordenadas ($O(n^2)$)
- **Uso Prático**: Evitar em produção, vulnerável a ataques de complexidade

9. Quick Sort (Central)

Complexidade: O(n log n) em quase todos os casos
Desempenho Observado:

- Mais rápido em geral (~0.008s para n=10.000)
- Evita pior caso da versão com pivô primeiro
- Performance consistente
- **Uso Prático**: Algoritmo de escolha geral para arrays na memória

---

Resultados e Gráficos Gerados

Conjunto Completo de Visualizações

Gráficos Individuais (9 arquivos)

- Desempenho de cada algoritmo nos 3 tipos de entrada
- Visualização do comportamento em melhor/pior caso
- Identificação de padrões de crescimento

Gráficos Comparativos (3 arquivos)

- Aleatorio: Comparação no caso médio
- Crescente: Comparação no melhor caso
- Decrescente: Comparação no pior caso

Gráficos de Algoritmos Rápidos (3 arquivos)

- Foco nos algoritmos O(n log n)
- Quick Sort vs Merge Sort vs Heap Sort vs Shell Sort
- Análise de constantes ocultas

Resumo Executivo

- Ranking de eficiência geral
- Tempos médios comparativos
- Tabela resumo para relatório

Análise de Complexidade: Teórica vs Prática

Confirmações Experimentais

1. **Hierarquia O(n log n) vs O(n^2)**
    - Diferença de 20-50x em n=20.000 elementos
    - Curvas de crescimento claramente distintas
    - Algoritmos O(n^2) tornam-se impraticáveis para n > 50.000
2. **Comportamentos de Melhor/Pior Caso**
    - Insertion Sort: O(n) em ordenados, O(n^2) em inversos
    - Quick Sort (Primeiro): O(n^2) em ordenados confirmado
    - Bubble Sort Melhorado: O(n) em ordenados verificado
3. **Constantes Ocultas Significativas**
    - Quick Sort ≈ 2x mais rápido que Merge Sort
    - Insertion Sort ≈ 2x mais rápido que Bubble Sort
    - Heap Sort mais lento devido a acesso não sequencial

Insights Práticos

Para Desenvolvedores:

Arrays Pequenos (n < 1000)

- Insertion Sort frequentemente mais rápido
- Overhead de algoritmos complexos não compensa

Arrays Médios/Grandes (n ≥ 1000)

- Quick Sort (Central) é a melhor escolha geral
- Merge Sort quando estabilidade é necessária
- Heap Sort para memória limitada

Casos Específicos

- Arrays quase ordenados: Insertion Sort
- Dados externos: Merge Sort
- Segurança: Merge Sort ou Heap Sort (evitar Quick Sort)

Metodologia de Validação

Verificação de Corretude

- Todos os algoritmos validados contra `sorted()` do Python
- Testes com arrays de diferentes características
- Verificação pós-execução em cada teste do benchmark

Precisão de Medição

- Uso de `time.perf_counter()` para alta precisão
- Múltiplas execuções para reduzir ruído
- Descarte automático de valores inconsistentes

Tratamento de Erros

- Captura de exceções e estouro de recursão
- Valores infinitos para algoritmos que falham
- Relatório detalhado de sucessos/falhas

Limitações e Considerações

Limitações do Estudo

- Execução em ambiente controlado (hardware específico)
- Vetores na memória principal
- Análise focada em complexidade temporal

Fatores que Influenciam Resultados

1. Hardware: Processador, cache, memória
2. Implementação: Linguagem, otimizações do compilador
3. Dados: Distribuição, tamanho, repetições

Complexidades Teóricas Comprovadas

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Estável
Bubble Sort Basico	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sim
Bubble Sort Melhorado	O(n)	$O(n^2)$	$O(n^2)$	Sim
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Não
Shell Sort	O(n log n)	O(n^(3/2))	$O(n^2)$	Não
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Não
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Sim
Quick Sort (Primeiro)	O(n log n)	O(n log n)	$O(n^2)$	Não
Quick Sort (Central)	O(n log n)	O(n log n)	$O(n^2)$*	Não

Conclusões Principais

Validação Científica

- Teoria confirmada: Comportamentos assintóticos observados experimentalmente
- Hierarquia mantida: O(n log n) significativamente superior a $O(n^2)$
- Casos específicos: Melhor/pior casos comportam-se como esperado

Recomendações Práticas

- Uso geral: Quick Sort (Central)
- Estabilidade: Merge Sort
- Arrays pequenos: Insertion Sort
- Memória limitada: Heap Sort
- Evitar: Bubble Sort básico, Quick Sort (Primeiro)

Contribuição Acadêmica

Este trabalho demonstra a importância da:

- Análise experimental complementar à teórica
- Consideração de constantes ocultas na prática
- Escolha de algoritmos baseada em cenários específicos

Referências Bibliográficas

1. CORMEN, T. H. et al. Algoritmos: Teoria e Prática. 3ª ed. Elsevier, 2012.
2. SEDGEWICK, R.; WAYNE, K. Algorithms. 4th ed. Addison-Wesley, 2011.
3. KNUTH, D. E. The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed. Addison-Wesley, 1998.
4. Material da disciplina - Prof. Danilo Medeiros Eler

Autores

Alexandre Malavazi
Ryan Rocha

FCT/Unesp – Presidente Prudente
Projeto e Análise de Algoritmos - 2025