Instituto Federal do Triângulo Mineiro - Campus ituiutaba
Disciplina: Análise de algoritmo
aluno: Viccenzo de oliveira nunes resende
Período: 6
Professora: Andrés chaves

Trabalho de Análise Empírica de Algoritmos de Ordenação
1. Objetivo

Realizar a análise empírica dos algoritmos de ordenação apresentados no Slide da Aula 06, implementando-os e comparando seus desempenhos por meio da execução em entradas geradas aleatoriamente.
2. Descrição da Atividade

O trabalho consiste em:
1. Implementar cada um dos algoritmos de ordenação estudados.
2. Executar os algoritmos com entradas de tamanhos crescentes, geradas de forma aleatória.
3. Registrar os tempos de processamento de cada execução.
4. Comparar os resultados obtidos entre os diferentes algoritmos.

Tamanhos de entrada obrigatórios:

· 10, 100, 1.000, 5.000, 10.000, 50.000, 100.000

Ampliar os testes até o máximo de elementos que sua máquina suportar até a data da entrega.
3. Entregáveis (Relatório)

O relatório deverá conter:
- Configurações da máquina utilizada (processador, memória, sistema operacional, etc.);
- Implementação de cada algoritmo testado;
- O número máximo de entradas alcançado em cada algoritmo;
- A complexidade teórica de cada algoritmo (melhor caso, caso médio e pior caso);
- Gráficos individuais: tamanho da entrada × tempo de execução de cada algoritmo;
- Gráfico comparativo: todos os algoritmos no mesmo gráfico;
- Referências bibliográficas utilizadas.
4. Critérios de Avaliação (10,0 pontos)

(2,0) Correção das implementações;
(2,0) Metodologia dos testes (organização, descrição e clareza dos experimentos);
(2,0) Apresentação dos resultados (tabelas e gráficos adequados);
(2,0) Discussão crítica dos resultados obtidos, comparando teoria e prática;
(1,0) Descrição correta das complexidades;
(1,0) Organização, redação e referências.
5. Formato de Entrega

• Relatório em PDF;

•link do código fonte disponibilizado no github;
• Nome do arquivo: Análise-Ordenação _<Turma >_<SeuNome > .pdf.
6. Data de Entrega
19/09/2025 – Moodle – até às 23h59

3)- Configurações da máquina utilizada (processador, memória, sistema operacional, etc.);
Geração instâncias : gerada aleatoriamente com random.randint(0, 10**9) para cada tamanho m gerar 1 instâncias diferentes e medir cada algoritmo sobre a cópia desta mesma instância - isso controla a variabilidade da entrada.

Repetições: medir cada par (algoritmo, n) r vezes e usar a mediana dos tempos para reduzir ruídos

Warm-up: execute uma ou duas execuções iniciais do algoritmo python timsort para aquecer cache/JIT (se usar PyPy) antes de coletar dados finais;

Ambientes: fechar outros processos intensivos(navegador com muitos abas, virtual machines etc.).

Validações sempre verificar se o resultado final está realmente ordenado(sorted(arr)).

Cuidado com O(n^2): para n grandes (>= 50k), pule ou limite execuções de bubble/insert/selection - registre no relatórios que foi publicado por tempo impraticável e qual tempo estimado/justificativa.

4) Tabela de complexidades (coloque no relatório)

Algoritmo	Melhor caso	Caso médio	Pior caso	Complexidade de espaço
Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Merge	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	O(n)
Quick (random pivot)	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$ (raro com pivo aleatório)	O(log n) recursion avg
Heap	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	O(1) (heap in-place) / O(n) with heapq variant
Shell	depende da sequência (ex.: Knuth)	entre n^(3/2) e n^(4/3) empiricamente	$O(n^2)$	O(1)
Timsort (Python sorted)	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	O(n)

5)O relatórios - estrutura sugerida(usar o texto como template)

O nome do arquivo: Análise-Ordenação
turma: 6 período
Meu nome: viccenzo de oliveira Nunes resende

Metodologia descrição da implementação (ambiente de linguagem google colab python)

Tamanhos testados: 10,1.00,1.000,5.000,10.000,100.000(e adicional até o máximo elemento que conseguiu executar no google colab).

Números de repetições e como foi calculada média(mediana)
Hardware e software(ver seção configurado na máquina).

3 Implementação
Mostrar o trechos importantes do código no google colab ou no link do github

4)Complexidade teóricas do algoritmo de ordenação no colab
tabela acima representa o resultado e depois mostrar a tabela do algoritmo de ordenação

5)Resultados
O resultado foi inserido na tabela com tempos

Gráficos separada e individual para cada algoritmo e gráfico comparativo(todos juntos)

O máximo que o n foi alcançado por algoritmo de ordenação(Exemplo do algoritmo bubble: 10.000 - parou em 160s; é o quick sort parou em 100.000 segundos; etc).

6)Discussão
Compare comportamento observado vs teoria:
Onde a prática seguiu a teoria?
Onde houve surpresas (por exemplo, timsort sendo mais rápido em instâncias parcialmente ordenadas)?
Por que os algoritmos $O(n^2)$ escalam mal (crescimento quadrático), mostrando números concretos.
Overheads de implementação em Python (recursão, custos de chamada de função).
Efeito de constantes e otimizações (por ex. heapq implementado em C é rápido).

7. Conclusões
Resumo das principais descobertas e recomendações (quando usar cada algoritmo).

8. Referências
Livros / slides usados (ex.: Cormen et al., slides da aula 06, documentação Python).

Apêndices
Código-fonte completo (ou link do GitHub).
CSV raw com tempos.

6) Como coletar as configurações da máquina (incluir no relatório)

Execute estes comandos e copie a saída no relatório:
Linux
lscpu
free -h
uname -a
python --version (e pip show matplotlib)
Windows (PowerShell)
systeminfo | findstr /B /C:"OS Name" /C:"OS Version" /C:"Total Physical Memory"
wmic cpu get name,numberofcores,maxclockspeed
python --version

No relatório inclua:
Processador (modelo e freq), número de núcleos/threads
Memória RAM total
Sistema Operacional (nome + versão)
Versão do Python e bibliotecas usadas
Se usou SSD/HDD (opcional)

7) Como relatar o n máximo alcançado por algoritmo
Execute o script com os tamanhos obrigatórios.
Se quiser ampliar, aumente a lista --sizes até sua máquina travar/ficar muito lenta.
Para cada algoritmo, registre:
maior n que concluiu sem truncar (ou tempo limite que você impôs).
tempo gasto.
No relatório mostre uma tabela tipo:
Algoritmo n máximo testado tempo (s) Observação
bubble 10000 152.3 pulado acima de 10000
merge 100000 0.98 completou sem problemas
quick 200000 1.10 ...
Dica: se quiser um critério rigoroso, pode impor um time limit por execução (ex.: 300 s) — se ultrapassar, interrompe e registra como "tempo excedido".
8) Observações + sugestões para discussão no relatório
Explique por que timsort (Python sorted) geralmente vence: otimizações em C + detecção de runs já ordenados.
Discuta overheads de chamadas recursivas em Python (merge/quick).
Argumente sobre uso real: para dados pequenos, insertion/shell podem ser melhores devido à baixa constante.
Mostre a diferença entre medidas em escala linear vs log-log (faça ambos os gráficos).
9) Preparar o GitHub e o link
Crie repositório público analise-ordenacao.
Coloque:
analise_ordenacao.py
README.md (instruções de execução)
results/ (opcional: coloque CSV e PNG gerados)
Analise-Ordenacao_<Turma>_<SeuNome>.pdf (arquivo final do relatório)
No Moodle entregue o link do repositório e o PDF.

10) Texto curto já pronto para a seção "Metodologia" do relatório (copiar/colar)
Metodologia — Para avaliar empiricamente os algoritmos de ordenação, implementamos as versões em Python puro para: Bubble, Selection, Insertion, Merge, Quick (pivô aleatório), Heap (com heapq), Shell (sequência simples) e Timsort (função sorted do Python). Testes foram realizados com entradas geradas aleatoriamente (inteiros na faixa [0, 10^9]). Para cada tamanho n utilizou-se a mesma instância para todos os algoritmos, executando cada experimento 3 vezes e tomando a mediana como valor representativo do tempo de execução. Medimos tempos com time.perf_counter() em ambiente controlado e registramos os resultados em CSV, gerando gráficos individuais e comparativos.
https://github.com/oviccenzo/trabalho-de-analise-de-algoritmo.git