TRABALHO 1

Gustavo Gabriel Ripka

Departamento de Informática

Universidade Federal do Paraná - UFPR

Curitiba, Brasil
ggr20@inf.ufpr.br

RESUMO

Trabalho desenvolvido na disciplina de Algoritmos e Estrutura de Dados 2, com o intuito de implementar os Algoritmos: Busca Binária, Busca Sequencial, *Selection Sort, Insertion Sort* ambas em suas versões iterativas e recursivas e *Merge Sort* exclusivamente recursiva. Foram feitas bancadas de testes para todos os algoritmos, com o intuito de analisar o custo computacional em parâmetros de tempo e comparações internas dos Algoritmos.

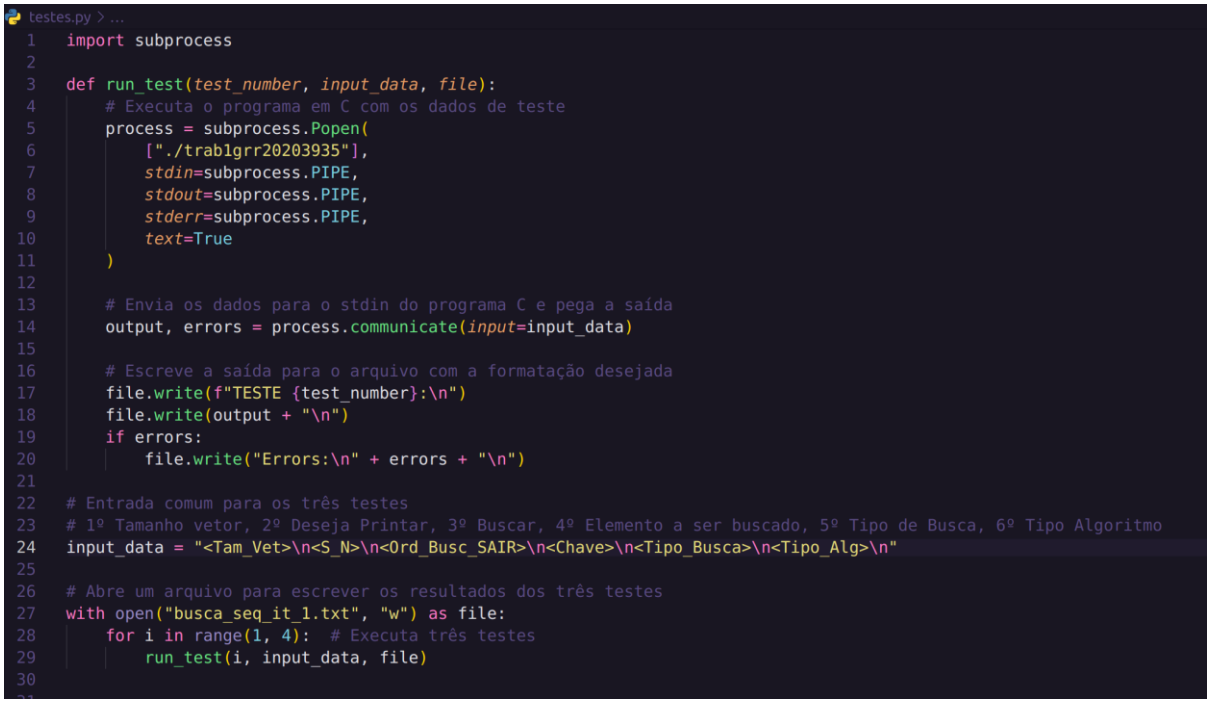
1. INTRODUÇÃO

A eficiência dos algoritmos é fundamental para maximizar o desempenho e minimizar o custo computacional, especialmente em aplicações onde o tempo de resposta e a utilização eficaz dos recursos são cruciais. Neste contexto, os algoritmos de ordenação e busca são peças centrais, pois estão frequentemente envolvidos na preparação de dados para análise, na facilitação do acesso rápido à informação e na otimização de processos decisórios. A ordenação, processo pelo qual elementos são sistematicamente arranjados em uma sequência específica (crescente ou decrescente), é a base para operações mais complexas em estruturas de dados. Por outro lado, a busca binária, exemplificando um método de busca eficiente, opera sobre um conjunto de dados previamente ordenado, dividindo-o repetidamente ao meio até encontrar o elemento desejado, destacando-se pela sua eficácia em comparação com buscas sequenciais, especialmente em grandes volumes de dados. A otimização desses algoritmos, portanto, não apenas melhora a performance de sistemas individuais, mas também eleva a eficiência global de operações computacionais, destacando a importância de algoritmos bem projetados e implementados na economia de tempo e recursos computacionais em uma variedade de aplicações práticas.

2. BENCHMARKS

Para medir a eficiência dos algoritmos de busca e ordenação, foi utilizado um *script* na linguagem *python* com intuito de automatizar a entrada do usuário.

Imagem 1: Visual Studio Code – Script.py para emular entrada do usuário.



```python
import subprocess

def run_test(test_number, input_data, file):
    # Executa o programa em C com os dados de teste
    process = subprocess.Popen(
        ["./trab1grr20203935"],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True
    )

    # Envia os dados para o stdin do programa C e pega a saída
    output, errors = process.communicate(input=input_data)

    # Escreve a saída para o arquivo com a formatação desejada
    file.write(f"TESTE {test_number}:\n")
    file.write(output + "\n")
    if errors:
        file.write("Errors:\n" + errors + "\n")

# Entrada comum para os três testes
# 1º Tamanho vetor, 2º Deseja Printar, 3º Buscar, 4º Elemento a ser buscado, 5º Tipo de Busca, 6º Tipo Algoritmo
input_data = "<Tam_Vet>\n<S_N>\n<Ord_Busc_SAIR>\n<Chave>\n<Tipo_Busca>\n<Tipo_Alg>\n"

# Abre um arquivo para escrever os resultados dos três testes
with open("busca_seq_it_1.txt", "w") as file:
    for i in range(1, 4):  # Executa três testes
        run_test(i, input_data, file)
```

Fonte: Autoria própria.

2.1. BUSCA SEQUENCIAL ITERATIVA

Alterando a linha 24 *"input_data"* para: input_data = "<100K, 1M, 1B>\n2\n2\n50\n2\n2\n, é obtido o custo computacional em números de comparação e tempo de execução, uma tentativa de deixar a bancada de testes mais padronizada possível, foi atribuída uma chave fixa, cujo valor inteiro é 50 e o algoritmo é passado 3 vezes para garantir aleatoriedade. Diante disso, segue saída do programa executado com os testes abaixo:

Tamanho Vetor = 1000000.

TESTE 1: Chave 50 encontrada na posição: 21, com 22 comparações, em 0.000001 segundos.

TESTE 2: Chave 50 encontrada na posição: 21, com 22 comparações, em 0.000001 segundos.

TESTE 3: Chave 50 encontrada na posição: 21, com 137960562919670 comparações, em 0.000001 segundos.

FIM DOS TESTES.

Tamanho Vetor = 1000000000.

TESTE 1: Chave 50 encontrada na posição: 56, com 57 comparações, em 0.000003 segundos.

TESTE 2: Chave 50 encontrada na posição: 52, com 53 comparações, em 0.000002 segundos.

TESTE 3: Chave 50 encontrada na posição: 22, com 23 comparações, em 0.000004 segundos.

FIM DOS TESTES.

Tamanho Vetor = 1000000000000

TODOS OS TESTES: Insira um tamanho de vetor de inteiros: Falha ao alocar memória.

Fica evidente com esses últimos testes, que o computador não possui memória o suficiente para alocar um vetor de tamanho 1000000000000.

2.2 BUSCA SEQUENCIAL RECURSIVA

Seguindo a mesma lógica para os testes, porém foi implementada uma modificação no último teste, onde antes era passado o tamanho de um vetor de 1 Bilhão de elementos agora foi passado um vetor de 3 Mil Milhões, alterando a linha 24 “input_data” para: input_data = "<100K, 1M, 3KM>\n2\n2\n50\n2\n1\n. Diante disso, segue saída do programa executado com os testes abaixo:

Tamanho Vetor = 1000000.

TESTE 1: Chave 50 encontrada na posição: 52, com 53 comparações, em 0.000001 segundos.

TESTE 2: Chave 50 encontrada na posição: 52, com 53 comparações, em 0.000002 segundos.

TESTE 3: Chave 50 encontrada na posição: 54, com 51 comparações, em 0.000001 segundos.

FIM DOS TESTES.

Tamanho Vetor = 1000000000.

TESTE 1: Chave 50 encontrada na posição: 56, com 138778290776345 comparações, em 0.000003 segundos.

TESTE 2: Chave 50 encontrada na posição: 12, com 13 comparações, em 0.000001 segundos.

TESTE 3: Chave 50 encontrada na posição: 22, com 135335031539959 comparações, em 0.000002 segundos.

FIM DOS TESTES.

Tamanho Vetor = 3000000000

TESTE 1: Chave 50 encontrada na posição: 56, com 57 comparações, em 0.000003 segundos.

TESTE 2: Chave 50 encontrada na posição: 52, com 53 comparações, em 0.000002 segundos.

TESTE 3: Chave 50 encontrada na posição: 241, com 242 comparações, em 0.000007 segundos.

FIM DOS TESTES.

Nos últimos testes utilizando algoritmos recursivos tivemos problemas com overflow, onde ultrapassou e muito o número de comparações.

2.3 ALGORITMOS DE ORDENAÇÃO

A bancada de testes foi medida, usando o mesmo algoritmo de entrada, implementando mais argumentos, seguindo a ordem: *Selection Sort, Insertion Sort, Merge Sort*, primeiramente em suas versões iterativas logo após em suas versões recursivas, unicamente o *Merge Sort* foi testado de forma recursiva, pois não foi solicitado uma versão iterativa.

2.3.1 SELECTION SORT ITERATIVO

Alterando a linha 24 *"input_data"* para: input_data = "<100K, 200K, 300K >\n2\n1\n1\n1\n2\n0\n, é obtido o tempo que levou o algoritmo para ser ordenado, com seu respectivo número de comparações, o algoritmo é passado 3 vezes para garantir aleatoriedade. Diante disso segue saída do programa executado com os testes abaixo:

Tamanho Vetor = 100000.

TESTE 1: Operação concluída em 5.924760 segundos com 4999950000 comparações.

TESTE 2: Operação concluída em 5.866206 segundos com 4999950000 comparações.

TESTE 3: Operação concluída em 5.876386 segundos com 4999950000 comparações.

FIM DOS TESTES.

Tamanho Vetor = 200000.

TESTE 1: Operação concluída em 23.378015 segundos com 19999900000 comparações.

TESTE 2: Operação concluída em 23.570383 segundos com 19999900000 comparações.

TESTE 3: Operação concluída em 23.668086 segundos com 19999900000 comparações.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Operação concluída em 52.511089 segundos com 44999850000 comparações.

TESTE 2: Operação concluída em 53.583862 segundos com 44999850000 comparações.

TESTE 3: Operação concluída em 52.965184 segundos com 44999850000 comparações.

FIM DOS TESTES.

2.3.2 SELECTION SORT RECURSIVO

Houve novamente modificação na linha do "input_data" para: input_data = "<100K, 200K 300K>\n2\n1\n1\n1\n0\n. Gerando várias saídas do algoritmo, agora de forma recursiva. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Operação concluída em 7.046262 segundos com 4999950000 comparações.

TESTE 2: Operação concluída em 7.035019 segundos com 4999950000 comparações.

TESTE 3: Operação concluída em 6.983639 segundos com 4999950000 comparações.

FIM DOS TESTES.

Tamanho Vetor = 200000.

TODOS OS TESTES: Falha de segmentação (imagem do núcleo gravada).

FIM DOS TESTES.

Tamanho Vetor = 300000.

TODOS OS TESTES: Falha de segmentação (imagem do núcleo gravada).

FIM DOS TESTES.

Possivelmente estourou a pilha de recursão para valores maiores deve existir alguma outra maneira de implementar.

2.3.3 INSERTION SORT ITERATIVO

Houve novamente modificação na linha do "input_data" para: input_data = "<100K, 200K 300K>\n2\n1\n2\n1\n0\n. Gerando várias saídas do algoritmo, agora de forma iterativa. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Operação concluída em 4.767158 segundos com 2473141641 comparações.

TESTE 2: Operação concluída em 4.811568 segundos com 2471533100 comparações.

TESTE 3: Operação concluída em 4.804681 segundos com 2474715442 comparações.

FIM DOS TESTES.

Tamanho Vetor = 200000.

TESTE 1: Operação concluída em 19.093698 segundos com 9894276789 comparações.

TESTE 2: Operação concluída em 19.263171 segundos com 9886727788 comparações.

TESTE 3: Operação concluída em 19.058298 segundos com 9894438273 comparações.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Operação concluída em 42.890296 segundos com 22237782511 comparações.

TESTE 2: Operação concluída em 43.014658 segundos com 22237886124 comparações.

TESTE 3: Operação concluída em 43.400206 segundos com 22276195985 comparações.

FIM DOS TESTES.

2.3.4 INSERTION SORT RECURSIVO

Houve novamente modificação na linha do “input_data” para: input_data = "<100K, 200K 300K>\n2\n1\n2\n2\n0\n. Gerando várias saídas do algoritmo, agora de forma recursiva. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Operação concluída em 4.793109 segundos com 2471792114 comparações.

TESTE 2: Operação concluída em 4.780683 segundos com 2475359048 comparações.

TESTE 3: Operação concluída em 4.773647 segundos com 2477117486 comparações.

FIM DOS TESTES.

Tamanho Vetor = 200000.

TESTE 1: Operação concluída em 19.178798 segundos com 9881625137 comparações.

TESTE 2: Operação concluída em 19.144527 segundos com 9865569529 comparações.

TESTE 3: Operação concluída em 19.072683 segundos com 9898625814 comparações.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Operação concluída em 42.929360 segundos com 22261266356 comparações.

TESTE 2: Operação concluída em 43.071594 segundos com 22279369043 comparações.

TESTE 3: Operação concluída em 42.868933 segundos com 22263534478 comparações.

FIM DOS TESTES.

A versão iterativa e recursiva do insertion sort praticamente empatam em termos de custo computacional, levando como parâmetro, tempo de execução e número de comparações.

2.3.4 MERGE SORT (APENAS RECURSIVO)

Por fim, dentre todos os algoritmos que foram solicitados para implementar, o Merge Sort tem apenas sua versão recursiva. Implementando uma modificação na linha 24 do script encurtando os argumentos da entrada do "input_data" para: input_data = "<100K, 200K 300K>\n2\n1\n3\n0\n. Gerando várias saídas do algoritmo, de forma recursiva. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Operação concluída em 0.009871 segundos com 1532831 comparações.

TESTE 2: Operação concluída em 0.009907 segundos com 1532831 comparações.

TESTE 3: Operação concluída em 0.010072 segundos com 1532831 comparações.

FIM DOS TESTES.

Tamanho Vetor = 200000.

TESTE 1: Operação concluída em 0.020585 segundos com 3264600 comparações.

TESTE 2: Operação concluída em 0.020723 segundos com 3264600 comparações.

TESTE 3: Operação concluída em 0.020631 segundos com 3264600 comparações.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Operação concluída em 0.031617 segundos com 5071466 comparações.

TESTE 2: Operação concluída em 0.033336 segundos com 5071466 comparações.

TESTE 3: Operação concluída em 0.031654 segundos com 5071466 comparações.

FIM DOS TESTES.

A diferença do *Merge Sort* para os outros algoritmos é surpreendente, porém, está sendo feito a análise de custo de outros componentes computacionais, se estivesse no parâmetro o custo de memória, possivelmente não teria um resultado tão satisfatório quanto para os outros algoritmos.

2.4 BUSCA BINÁRIA ITERATIVA

Consiste em algoritmo que depende exclusivamente de o estado do vetor estar ordenado, porque ele utiliza da premissa "dividir para conquistar". Dentre os algoritmos de busca é o mais eficiente, porém, precisa ser garantido uma entrada do vetor ordenado. Foi utilizado o mesmo

script de entrada, uma tentativa de deixar a bancada de testes mais padronizada possível, foi atribuída uma chave fixa, cujo valor inteiro é 7500. Implementando uma modificação na linha 24 do script encurtando os argumentos da entrada do "input_data" para: input_data = "<100K, 200K 300K>\n2\n1\n3\n2\n1\n7500\n1\n1\n". Gerando várias saídas do algoritmo, de forma iterativa. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Chave 7500 encontrada na posição: 74632, com 12 comparações, em 0.000000 segundos.

TESTE 2: Chave 7500 encontrada na posição: 74632, com 12 comparações, em 0.000001 segundos.

TESTE 3: Chave 7500 encontrada na posição: 74632, com 12 comparações, em 0.000000 segundos.

Tamanho Vetor = 200000.

TESTE 1: Chave 7500 encontrada na posição: 150218, com 13 comparações, em 0.000001 segundos.

TESTE 2: Chave 7500 encontrada na posição: 150218, com 13 comparações, em 0.000000 segundos.

TESTE 3: Chave 7500 encontrada na posição: 150218, com 13 comparações, em 0.000001 segundos.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Chave 7500 encontrada na posição: 224962, com 13 comparações, em 0.000001 segundos.

TESTE 2: Chave 7500 encontrada na posição: 224962, com 13 comparações, em 0.000001 segundos.

TESTE 3: Chave 7500 encontrada na posição: 224962, com 13 comparações, em 0.000000 segundos.

FIM DOS TESTES.

2.5 BUSCA BINÁRIA RECURSIVA

Houve novamente modificação na linha do "input_data" para: input_data = "<100K, 200K 300K>\n2\n1\n3\n2\n1\n7500\n1\n2\n". Gerando várias saídas do algoritmo, agora de forma recursiva. Diante disso, segue abaixo os testes gerados pelo algoritmo:

Tamanho Vetor = 100000.

TESTE 1: Chave 7500 encontrada na posição: 75218, com 12 comparações, em 0.000001 segundos.

TESTE 2: Chave 7500 encontrada na posição: 75218, com 12 comparações, em 0.000001 segundos.

TESTE 3: Chave 7500 encontrada na posição: 75218, com 12 comparações, em 0.000000 segundos.

Tamanho Vetor = 200000.

TESTE 1: Chave 7500 encontrada na posição: 149999, com 2 comparações, em 0.000000 segundos.

TESTE 2: Chave 7500 encontrada na posição: 149950, com 12 comparações, em 0.000001 segundos.

TESTE 3: Chave 7500 encontrada na posição: 150218, com 13 comparações, em 0.000001 segundos.

FIM DOS TESTES.

Tamanho Vetor = 300000.

TESTE 1: Chave 7500 encontrada na posição: 224742, com 13 comparações, em 0.000001 segundos.

TESTE 2: Chave 7500 encontrada na posição: 224742, com 13 comparações, em 0.000001 segundos.

TESTE 3: Chave 7500 encontrada na posição: 224742, com 13 comparações, em 0.000001 segundos.

FIM DOS TESTES.

É surpreendente a eficiência desse algoritmo para buscar e encontrar um elemento no vetor, mas da mesma forma que o *Merge Sort* é excelente por ser rápido, o algoritmo de Busca Binária apenas funciona com vetores que estiverem ordenados.

2.6 SÍNTESE DOS ALGORITMOS DE ORDENAÇÃO

2.6.1 SELECTION SORT ITERATIVO E RECURSIVO:

Iterativo: Mostrou um alto custo computacional, especialmente em conjuntos de dados maiores, com um número de comparações que cresce de forma quadrática com o aumento do tamanho do vetor.

Recursivo: Apresentou problemas de falha de segmentação em tamanhos de vetor maiores, indicando um possível estouro da pilha de recursão devido ao grande número de chamadas recursivas.

2.6.2 INSERTION SORT ITERATIVO E RECURSIVO:

Iterativo: Eficiente para conjuntos de dados menores, mas seu desempenho cai significativamente com o aumento do tamanho do vetor. O número de comparações e o tempo de execução também aumentam de forma quadrática.

Recursivo: Similar ao iterativo em termos de custo computacional, mas sofre com os mesmos problemas de eficiência em conjuntos de dados grandes.

2.6.3 MERGE SORT (RECURSIVO):

Recursivo: Destacou-se pela eficiência em todos os tamanhos de conjuntos de dados testados, com um número de comparações significativamente menor e um tempo de execução muito mais rápido em comparação com os outros algoritmos de ordenação.

2.7 SÍNTESE DOS ALGORITMOS DE BUSCA

2.7.1 BUSCA SEQUENCIAL ITERATIVA E RECURSIVA:

Iterativa: Efetiva, mas ineficiente para vetores grandes, conforme mostrado pelo número de comparações necessárias para encontrar o elemento desejado.

Recursiva: Encontrou problemas similares à versão iterativa, com um custo computacional alto devido ao número de comparações.

2.7.2 BUSCA BINÁRIA ITERATIVA E RECURSIVA:

Iterativa e Recursiva: Extremamente eficientes em termos de tempo de execução e número de comparações, mas dependem de vetores previamente ordenados. Demonstraram um desempenho excepcional com um número mínimo de comparações, mesmo em vetores de tamanho considerável.

3. GLOSSÁRIO

Benchmark: ato de comparar de forma eficiente a performance entre dispositivos utilizando um ou mais programas, no caso desse relatório algoritmos.

Algoritmo: sequência finita de regras, raciocínios ou operações que, aplicada a um número finito de dados, permite solucionar classes semelhantes de problemas.

Recursividade: Método em programação onde a função chama a si mesma diretamente ou indiretamente, permitindo que problemas sejam resolvidos de maneira mais elegante e com menos código.

Overflow: Condição em sistemas computacionais onde o valor calculado excede o espaço alocado para armazená-lo, levando a erros ou comportamento inesperado.

Segmentação: Erro que ocorre quando um programa tenta acessar áreas de memória restritas ou não alocadas, frequentemente resultando em falha do programa.

Merge: Intercalar.

4. CONCLUSÃO

Com base no que foi estudado e feito prática de diversos algoritmos de busca e ordenação, destacando suas características em termos de eficiência e custo computacional. Através de extensivos testes, foram identificados que enquanto algoritmos como o Merge Sort e a Busca Binária se destacam pela sua rapidez e eficiência em conjuntos de dados ordenados, eles podem exigir considerações adicionais quanto ao uso de recursos, como memória. Por outro lado, algoritmos como *Selection Sort* e *Insertion Sort*, apesar de menos eficientes em conjuntos de dados grandes, provaram ser robustos e confiáveis em cenários de uso variado. Esta análise reforça a importância de escolher o algoritmo adequado baseado nas características específicas dos dados e dos requisitos de cada aplicação, um equilíbrio essencial entre eficiência computacional e consumo de recursos.

REFERÊNCIAS

[1] CORMEN, Thomas H. Desmistificando algoritmos. Tradução de Arlete Simille Marques. 1. ed. Rio de Janeiro: Elsevier, 2014. Título original: *Algorithms Unlocked.*

[2] CORMEN, Thomas H. et al. Algoritmos: teoria e prática. Tradução de Daniel Vieira. Revisão técnica de João Araujo Ribeiro. 4. ed. Rio de Janeiro: LTC, 2024. Título original: *Introduction to algorithms.*

[3] PIVA JR, Dilermando et al. *Algoritmos e programação de computadores*. 2. ed. Rio de Janeiro: Elsevier, 2019.