1. A importância dos algoritmos de ordenação na computação.

Algoritmos de ordenação são fundamentais na ciência da computação: eles permitem reorganizar dados de modo que estejam em uma sequência bem definida (ex: crescente ou decrescente), o que facilita operações como busca, compressão, junção de dados e recuperação eficiente. Quando os dados estão ordenados, é possível acelerar algoritmos de pesquisa (como busca binária), otimizar estruturas de dados (árvores de busca, índices em bancos de dados) e melhorar a performance de sistemas em geral.

Além disso, diferentes algoritmos de ordenação possuem características distintas de desempenho em termos de tempo de execução, uso de memória e estabilidade (se preservam a ordem entre elementos iguais). A escolha de um algoritmo adequado depende do tamanho dos dados, da distribuição inicial e das restrições de recursos, tornando o estudo de algoritmos de ordenação um pilar nos cursos de estruturas de dados e algoritmos.

2. Descrição objetiva do funcionamento de cada método.

Foram escolhidos três métodos de ordenação para exemplo: Bubble Sort, Quick Sort e Shell Sort.

2.1. Bubble Sort

2.1.1. Descrição / Ideia principal

É um dos algoritmos de ordenação mais simples: percorre a lista repetidamente, comparando pares adjacentes e trocando-os se estiverem fora de ordem (“faz borbulhar” os maiores elementos para o final). Repete-se até que uma passada inteira ocorra sem nenhuma troca, indicando que a lista está ordenada.

Originalmente proposto por Edward H. Friend em 1956.

2.1.2. Vantagens

1- Extremamente simples de entender e implementar.
2- Adequado para listas muito pequenas ou quando a simplicidade tem prioridade sobre a eficiência.
3- Pode ser otimizado para detectar listas já ordenadas (parar antes).

2.1.3. Desvantagens / Limitações

1- Ineficiente para grandes volumes de dados devido ao crescimento quadrático no número de comparações e trocas.
2- Alto custo de movimentações de dados.
3- Geralmente não usado em sistemas de produção para ordenações intensivas — há algoritmos mais rápidos.

2.2. Quick Sort

2.2.1. Descrição / Ideia principal:

É um algoritmo de comparação que usa a técnica de “divisão e conquista”: escolhe-se um *pivô* (um elemento do array), e particiona-se o restante dos elementos em dois subarrays — aqueles menores que o pivô à esquerda, e aqueles maiores à direita. Isso é feito recursivamente até os subarrays ficarem triviais.

Originalmente proposto por C. A. R. Hoare na década de 1960.

2.2.2. Vantagens:

1- Muito eficiente na prática para muitos tipos de dados.

2- Boa localidade de referência (se implementado bem) — particionar em pedaços menores ajuda no cache.
3- Flexível: pode ser adaptado (ex: escolha de pivô aleatório, versão de três pivôs, otimizações como "block-quicksort"). Por exemplo, estudos mostram que branch mispredictions podem ser reduzidas com uma variante de blocos.

2.2.3. Desvantagens / Limitações:

1- No pior caso, tem desempenho quadrático se não for bem implementado.
2- Não é um algoritmo estável por padrão (a ordem de elementos iguais pode não ser preservada).
3- Pode haver sobrecarga de recursão para arrays muito grandes, dependendo da implementação.

2.3. Shell Sort

2.3.1. Descrição / Ideia principal:

É um refinamento do *insertion sort*: em vez de ordenar o array como um único segmento, ele divide-o em sub-vetores (com "gaps", ou intervalos) e aplica inserção em cada sub-vetor.

A sequência de espaçamentos ("h-sequência") diminui ao longo das passadas, até chegar em 1, quando se torna similar ao insertion sort padrão.

Foi criado por Donald Shell em 1959.

2.3.2. Vantagens:

1- Melhor desempenho que algoritmos quadráticos puros (como bubble sort) para muitos tamanhos de dados médios.
2- Ainda simples de implementar, comparado com algoritmos muito complexos.
3- Não requer memória adicional significativa (in-place).
4- Pode se adaptar bem quando os dados já têm alguma ordenação parcial.

2.3.3. Desvantagens / Limitações:

1- A eficiência depende criticamente da escolha da sequência de h ("gap"), e não há uma sequência ótima universal.
2- Não é estável (elementos iguais podem trocar de ordem).
3- Pode não ser tão rápida quanto algoritmos O(n log n) altamente otimizados para grandes dados.

3. Pseudocódigo dos algoritmos

3.1. Bubble Sort

O algoritmo Bubble Sort é apresentado na literatura como um método de ordenação simples, baseado em comparações sucessivas entre elementos adjacentes de um vetor, realizando trocas sempre que eles estiverem fora de ordem. A seguir, apresenta-se o pseudocódigo do algoritmo, conforme Puga e Rissetti (2016).

```
Algoritmo BolhaIterativa(v: vetor[0..n] de inteiros): vetor de inteiros
Var
    i, j, aux: inteiro
Início
    Para j ← n até j >= 1 passo -1 faça
        Para i ← 0 até i < j passo 1 faça
```

```
            Se (v[i] > v[i+1]) então
                aux ← v[i]
                v[i] ← v[i+1]
                v[i+1] ← aux;
            Fim-Se
        Fim-Para
    Fim-Para
    Retornar v
Fim Algoritmo.
```

Fonte: PUGA, Sandra G.; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 3. ed. São Paulo: Pearson, 2016.

3.2. Quick Sort

O Quick Sort é um algoritmo de ordenação que utiliza a estratégia de divisão e conquista: a partir da escolha de um pivô, o vetor é particionado em duas partes, com elementos menores que o pivô à esquerda e maiores à direita, aplicando-se o processo recursivamente. O pseudocódigo a seguir está de acordo com a apresentação de Puga e Rissetti (2016).

```
Procedimento Quicksort(primeiro, ultimo: inteiro; v: vetor[0..n] de
inteiros)
Var
    x: inteiro
Início
    Se (primeiro < ultimo) então
        x ← Particao(primeiro, ultimo, v)
        Quicksort(primeiro, x - 1, v)
        Quicksort(x + 1, ultimo, v)
    Fim-Se
Fim Procedimento.
Função Particao(primeiro, ultimo: inteiro; v: vetor[0..n] de inteiros):
inteiro
Var
    pivo, aux, i, j: inteiro
Início
    j ← ultimo
    pivo ← v[primeiro]
    Para (i ← ultimo até primeiro passo -1) faça
        Se (v[i] >= pivo) então
            aux ← v[j]
            v[j] ← v[i]
            v[i] ← aux
            j ← j - 1
        Fim-Se
```

```
    Fim-Para
    Retornar j + 1
Fim Função.
```

Fonte: PUGA, Sandra G.; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 3. ed. São Paulo: Pearson, 2016.

3.3. Shell Sort

O algoritmo Shell Sort é um refinamento do método de ordenação por inserção, utilizando uma sequência de "gaps" (intervalos) para ordenar elementos que estão mais distantes no vetor, reduzindo gradualmente esse intervalo até realizar uma ordenação final com gap igual a 1. O pseudocódigo a seguir é baseado na descrição apresentada por Ascencio e Campos (2008).

Função ShellSort(v: vetor[0..n] de inteiros): vetor de inteiros

```
Var
    gap, i, j: inteiro
    aux: inteiro
Início
    gap ← n div 2
    Enquanto (gap > 0) faça
        Para i ← gap até n passo 1 faça
            aux ← v[i]
            j ← i
            Enquanto (j >= gap) e (v[j - gap] > aux) faça
                v[j] ← v[j - gap]
                j ← j - gap
            Fim-Enquanto
            v[j] ← aux
        Fim-Para
        gap ← gap div 2
    Fim-Enquanto
    Retornar v
Fim ShellSort.
```

Fonte: ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi. Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java. 3. ed. São Paulo: Pearson Prentice Hall, 2008.

4. Exemplo PráticoPara cada algoritmo

4.1. Exemplo prático – Bubble Sort

Considerando o vetor inicial:

V = [34, 8, 50, 23, 8, 1, 42, 15, 4, 31],

a aplicação do algoritmo Bubble Sort gera as seguintes passagens do vetor, ao final de cada iteração do laço externo:

Passo	Situação	Vetor
0	Vetor inicial	[34, 8, 50, 23, 8, 1, 42, 15, 4, 31]
1	Após 1ª passada completa (j = 9)	[8, 34, 23, 8, 1, 42, 15, 4, 31, 50]
2	Após 2ª passada completa (j = 8)	[8, 23, 8, 1, 34, 15, 4, 31, 42, 50]
3	Após 3ª passada completa (j = 7)	[8, 8, 1, 23, 15, 4, 31, 34, 42, 50]
4	Após 4ª passada completa (j = 6)	[8, 1, 8, 15, 4, 23, 31, 34, 42, 50]
5	Após 5ª passada completa (j = 5)	[1, 8, 8, 4, 15, 23, 31, 34, 42, 50]
6	Após 6ª passada completa (j = 4)	[1, 8, 4, 8, 15, 23, 31, 34, 42, 50]
7	Após 7ª passada completa (j = 3)	[1, 4, 8, 8, 15, 23, 31, 34, 42, 50]

Observa-se que, a cada passada do algoritmo, o maior elemento dentre os ainda não ordenados é “empurrado” para o final da porção considerada do vetor. Na primeira passada, o valor 50 é deslocado para a última posição. Nas passadas seguintes, os maiores valores restantes (42, 34, etc.) vão sendo posicionados até que o vetor esteja totalmente ordenado em ordem crescente.

4.2. Exemplo prático – Quick Sort

A seguir, apresenta-se a aplicação do algoritmo Quick Sort sobre o mesmo vetor utilizado anteriormente:

V = [34, 8, 50, 23, 8, 1, 42, 15, 4, 31].

É utilizada a estratégia de escolher o último elemento do subvetor como pivô e particionar o vetor em torno dele, aplicando recursão às partições resultantes.

Etapa	Subvetor considerado ([início..fim])	Ação	Vetor completo após a etapa
0	[0..9]	Vetor inicial	[34, 8, 50, 23, 8, 1, 42, 15, 4, 31]
1	[0..9]	Pivô = 31 → particiona	[8, 23, 8, 1, 15, 4, 31, 50, 42, 34]
2	[0..5] (subvetor à esquerda do 31)	Pivô = 4 → particiona	[1, 4, 8, 23, 8, 15, 31, 50, 42, 34]
3	[0..0]	Subvetor de um elemento (parada)	[1, 4, 8, 23, 8, 15, 31, 50, 42, 34]
4	[2..5] (entre 4 e 31)	Pivô = 15 → particiona	[1, 4, 8, 8, 15, 23, 31, 50, 42, 34]
5	[2..3] (subvetor à esquerda do 15)	Pivô = 8 → particiona	[1, 4, 8, 8, 15, 23, 31, 50, 42, 34]
6	[5..5]	Subvetor de um elemento (parada)	[1, 4, 8, 8, 15, 23, 31, 50, 42, 34]
7	[7..9] (subvetor à direita do 31)	Pivô = 34 → particiona	[1, 4, 8, 8, 15, 23, 31, 34, 42, 50]
8	[7..7] e [9..9]	Subvetores de um elemento (parada)	[1, 4, 8, 8, 15, 23, 31, 34, 42, 50]

Na Etapa 1, o pivô escolhido é 31 (último elemento do vetor). Após a partição, todos os elementos menores ou iguais a 31 ficam à esquerda, e os maiores, à direita, com o pivô em sua posição definitiva. Em seguida, o algoritmo aplica recursivamente o mesmo processo aos subvetores [0..5] e [7..9].

O processo continua com novos pivôs (4, 15, 8, 34), refinando as partições até que todos os subvetores tenham tamanho 1, resultando no vetor totalmente ordenado:

[1, 4, 8, 8, 15, 23, 31, 34, 42, 50].

4.3. Exemplo prático – Shell Sort

O algoritmo Shell Sort é uma generalização do Insertion Sort que utiliza intervalos (gaps) para comparar e deslocar elementos que estão distantes entre si. A ideia é ir reduzindo gradualmente o valor do gap até que ele se torne igual a 1, momento em que o algoritmo se comporta como um Insertion Sort tradicional, porém sobre um vetor previamente “pré-ordenado”, o que tende a reduzir o número total de movimentações.

Etapa	gap	Descrição resumida	Vetor após a etapa
0	–	Vetor inicial	[34, 8, 50, 23, 8, 1, 42, 15, 4, 31]
1	5	Ordenação considerando elementos separados por gap = 5	[1, 8, 4, 15, 8, 34, 42, 23, 50, 31]
2	2	Nova ordenação com gap = 2 (subsequências de índice par/ímpar)	[1, 4, 8, 15, 8, 23, 42, 31, 50, 34]
3	1	Ordenação final com gap = 1 (Insertion Sort tradicional)	[1, 4, 8, 8, 15, 23, 31, 34, 42, 50]

Na Etapa 1, com gap = 5, o vetor é dividido, implicitamente, em subsequências formadas por elementos distantes cinco posições entre si, e cada subsequência é ordenada por inserção. Em seguida, com gap = 2, são formadas novas subsequências (índices pares e ímpares), que também são ordenadas. Por fim, com gap = 1, o algoritmo realiza uma última ordenação por inserção em um vetor já parcialmente organizado, resultando no vetor totalmente ordenado:

[1, 4, 8, 8, 15, 23, 31, 34, 42, 50].

5. Implementação em Java dos algoritmos

Nesta seção são apresentados os códigos em Java implementados pela equipe para os três algoritmos de ordenação estudados: Bubble Sort, Quick Sort e Shell Sort. As implementações foram realizadas utilizando vetores representados por ArrayList<Integer> e integram um programa principal com menu para carregamento de arquivos de dados e execução das ordenações.

5.1. Implementação em Java – Bubble Sort:

```
private static void bubbleSort(ArrayList<Integer> vetor) {
    BenchMark bench = new BenchMark();
    MemoryUsage mem = new MemoryUsage();
    int comparisons = 0;
    mem.start();
    bench.start();

    int n = vetor.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            comparisons++;
            if (vetor.get(j) > vetor.get(j + 1)) {
                int temp = vetor.get(j);
                vetor.set(j, vetor.get(j + 1));
                vetor.set(j + 1, temp);
            }
        }
    }

    bench.stop();
    mem.stop();
    show(vetor);
    time(bench);
    memory(mem);
    showComparisons(comparisons);
}
```

5.2. Implementação em Java – Quick Sort:

```
private static void quickSort(ArrayList<Integer> vetor) {
    BenchMark bench = new BenchMark();
    MemoryUsage mem = new MemoryUsage();
    mem.start();
    bench.start();
    int comparisons = quickSort(vetor, 0, vetor.size() - 1);
    bench.stop();
    mem.stop();
    show(vetor);
    time(bench);
    memory(mem);
    showComparisons(comparisons);
}
private static int quickSort(ArrayList<Integer> vetor, int inicio, int fim)
{
    int comparisons = 0;
    if (inicio < fim) {
        int pivo = vetor.get(fim); // pivô no final
        int i = (inicio - 1);

        for (int j = inicio; j < fim; j++) {
            comparisons++;
            if (vetor.get(j) <= pivo) {
                i++;
                int temp = vetor.get(i);
                vetor.set(i, vetor.get(j));
                vetor.set(j, temp);
            }
        }
        int temp = vetor.get(i + 1);
        vetor.set(i + 1, vetor.get(fim));
        vetor.set(fim, temp);
        int indicePivo = i + 1;
        comparisons += quickSort(vetor, inicio, indicePivo - 1);
        comparisons += quickSort(vetor, indicePivo + 1, fim);
    }
    return comparisons;
}
```

5.3. Implementação em Java – Shell Sort:

```
private static void shellSort(ArrayList<Integer> vetor) {
    BenchMark bench = new BenchMark();
    MemoryUsage mem = new MemoryUsage();
    int comparisons = 0;

    mem.start();
    bench.start();

    int n = vetor.size();
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = vetor.get(i);
            int j;

            for (j = i; j >= gap; j -= gap) {
                comparisons++;
                if (vetor.get(j - gap) > temp) {
                    vetor.set(j, vetor.get(j - gap));
                } else {
                    break;
                }
            }
            vetor.set(j, temp);
        }
    }

    bench.stop();

    mem.stop();

    show(vetor);

    time(bench);

    memory(mem);

    showComparisons(comparisons);
}
```

6. Resultados experimentais

Para avaliar o desempenho prático dos algoritmos, foram gerados arquivos contendo conjuntos de números inteiros com diferentes tamanhos (1.000, 5.000, 10.000 e 50.000 elementos). Em seguida, cada um dos três algoritmos (Bubble Sort, Quick Sort e Shell Sort) foi executado para ordenar esses conjuntos, e o tempo de execução foi medido em milissegundos. Adicionalmente, foram feitas medições aproximadas do uso de memória e da quantidade de comparações realizadas, com o objetivo de comparar o comportamento real com a análise teórica.

6.1. Arquivos de teste

Foram utilizados quatro arquivos de teste, contendo números inteiros gerados aleatoriamente:

1.txt – 1.000 elementos

2.txt – 5.000 elementos

3.txt – 10.000 elementos

4.txt – 50.000 elementos

6.2. Tabela de tempos de execução

Tabela de tempo de execução (ms) por algoritmo e tamanho de entrada:

Arquivo	Tamanho	Bubble (ms)	Quick (ms)	Shell (ms)
1.txt	1000	22,742	0,946	1,457
2.txt	5000	95,731	1,499	6,747
3.txt	10000	261,313	1,676	2,138
4.txt	50000	8083,979	8,925	11,123

6.3. Gráficos

A partir da Tabela de tempo de execusão (ms), foram elaborados gráficos de linha relacionando o tamanho da entrada (eixo x) com o tempo de execução (eixo y) para cada algoritmo. Observa-se que o tempo do Bubble Sort cresce de forma muito mais acentuada, enquanto Quick Sort e Shell Sort mantêm tempos bem menores conforme o tamanho do vetor aumenta.

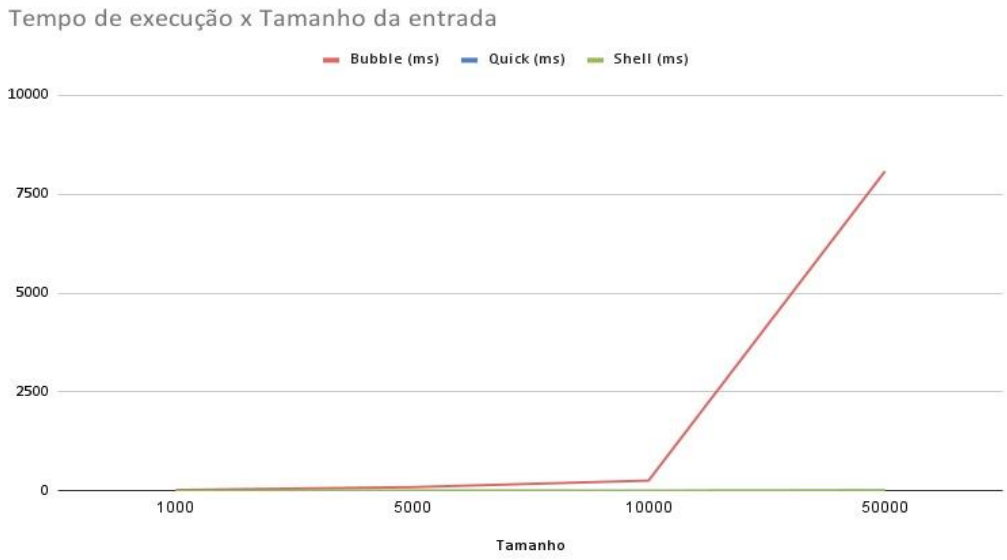

Figura 1 – Tempo de execução x Tamanho da entrada (Bubble, Quick, Shell).

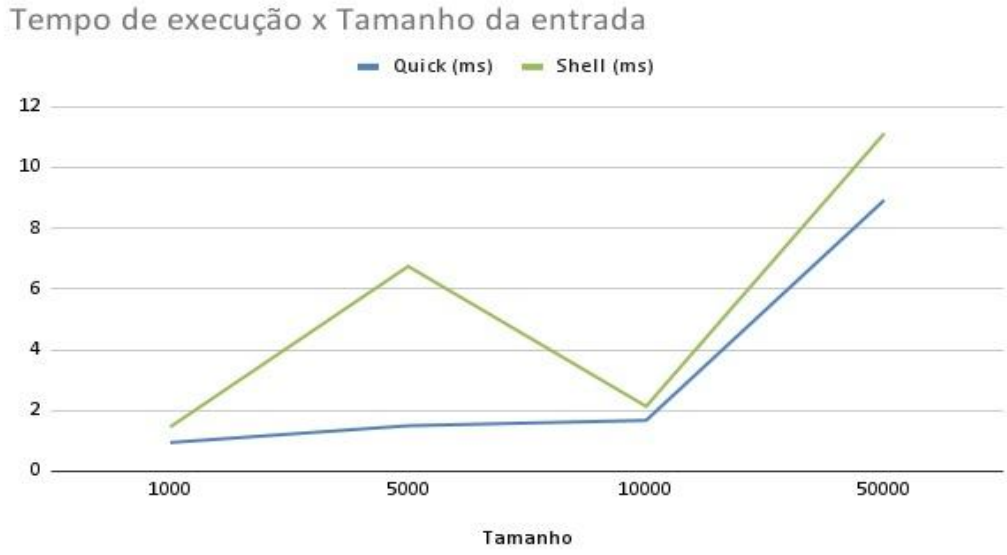

Figura 2 – Tempo de execução x Tamanho da entrada (Quick x Shell, ampliado).

No gráfico com os três algoritmos (Figura 1), a curva do Bubble Sort domina a escala, tornando as curvas de Quick Sort e Shell Sort quase imperceptíveis. Por isso, foi gerado um segundo gráfico (Figura 2) apenas com Quick Sort e Shell Sort, permitindo visualizar melhor a diferença de desempenho entre eles.

7. Análise Comparativa

Nesta seção, são comparados os algoritmos Bubble Sort, Quick Sort e Shell Sort quanto à complexidade de tempo, estabilidade, uso de memória e facilidade de implementação. A análise considera o comportamento teórico de cada método e suas implicações práticas.

7.1. Complexidade de tempo

Bubble Sort

- Melhor caso: O(n) se a implementação interrompe quando não há trocas (vetor já ordenado).

- Médio caso: $O(n^2)$.

- Pior caso: $O(n^2)$.

Quick Sort

- Melhor caso: O(n log n) (pivô divide bem o vetor em partes semelhantes).

- Médio caso: O(n log n) (na prática, muito eficiente).

- Pior caso: $O(n^2)$ (quando o pivô é sempre muito ruim, por exemplo sempre o menor ou o maior elemento).

Shell Sort

- Melhor caso: aproximadamente O(n log n)

- Médio/pior caso: entre O(n^1,25) e O(n^2), dependendo da sequência de gaps, mas com desempenho prático melhor que os sorts quadráticos simples (Bubble/Insertion).

Os resultados experimentais apresentados na tabela de tempo de execução (ms) confirmam esse comportamento teórico: à medida que o tamanho da entrada aumenta de 1.000 para 50.000 elementos, o tempo de execução do Bubble Sort cresce de forma muito mais rápida (de aproximadamente 22 ms para mais de 8.000 ms), evidenciando o custo quadrático. Em contraste, o Quick Sort manteve tempos sempre próximos da ordem de poucos milissegundos (por exemplo, cerca de 9 ms para 50.000 elementos), condizentes com uma complexidade próxima de O(n log n). O Shell Sort apresentou desempenho intermediário, significativamente melhor que o Bubble Sort e levemente pior que o Quick Sort para os tamanhos testados.

7.2. Estabilidade

Um algoritmo de ordenação é estável quando não altera a ordem relativa de elementos com chaves iguais. Por exemplo, se dois registros possuem o mesmo valor de chave, um algoritmo estável garante que, após a ordenação, eles aparecerão na mesma ordem em que estavam originalmente.

- **Bubble Sort** → normalmente estável, desde que a troca só ocorra quando > e não quando >=.

- **Quick Sor**t → não estável na forma clássica (como a que implementamos).

- **Shell Sort** → não estável, pois os “saltos” de gap podem inverter a ordem de elementos iguais.

7.3. Uso de memória

Bubble Sort:

- Usa apenas algumas variáveis auxiliares (aux, índices).

- Não cria vetores extras.

- In-place (em memória).

Quick Sort:

- Também reorganiza os elementos no próprio vetor.

- Mas usa recursão, então há consumo de pilha de chamadas.

- Memória extra proporcional à profundidade da recursão (em média O(log n), no pior caso O(n)).

Shell Sort:

- Assim como o Insertion Sort, usa apenas algumas variáveis auxiliares.

- Não precisa de vetor extra.

- In-place.

As medições práticas de memória indicaram que, para um mesmo tamanho de vetor, os três algoritmos utilizam quantidades muito próximas de memória, sempre na mesma ordem de grandeza. Isso sugere que o custo dominante de memória está em manter o vetor de dados carregado em um ArrayList, e não nas estruturas internas específicas de cada algoritmo. Assim, mesmo que teoricamente o Quick Sort utilize memória adicional de O(log n) para a pilha de recursão, na prática essa diferença se mostrou pequena em comparação ao espaço necessário para armazenar os dados, reforçando o caráter in-place dos três métodos.

7.4. Facilidade de implementação

Bubble Sort:

- Estrutura simples: dois laços aninhados, comparação de elementos adjacentes.

- Pseudocódigo curto.

- Geralmente o algoritmo de ordenação mais fácil de entender e implementar.

- Desempenho ruim para vetores grandes.

Quick Sort:

- Exige entender: escolha de pivô, função de partição, recursão.

- Um pouco mais complexo de programar corretamente (fácil errar índices).

- Um dos algoritmos de ordenação mais eficientes na prática.

Shell Sort:

- Conceitualmente intermediário: é um Insertion Sort com gaps variáveis.

- Precisa decidir uma sequência de gaps e implementar laços com saltos.

- Mais fácil que Quick Sort em termos de lógica, mas menos direto que Bubble.

7.5. Tabela comparativa final

Comparação entre Bubble Sort, Quick Sort e Shell Sort

Algoritmo	Melhor caso	Caso médio	Pior caso	Estável?	Uso de memória	Facilidade de implementação
Bubble Sort	O(n) (vetor já ordenado, parada antecipada)	$O(n^2)$	$O(n^2)$	Sim	In-place, apenas variáveis auxiliares	Muito simples; indicado para fins didáticos
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	Não	In-place, mas usa pilha de recursão (O(log n) em média)	Complexidade média; exige compreender recursão e partição
Shell Sort	≈ O(n log n)*	Entre O(n^1,25) e $O(n^2)$*	Até $O(n^2)$* (depende dos gaps)	Não	In-place, apenas variáveis auxiliares	Intermediária; extensão do Insertion Sort com gaps

De forma geral, o Bubble Sort apresenta implementação muito simples, porém com desempenho quadrático, o que o torna pouco indicado para grandes volumes de dados. O Quick Sort, por outro lado, é mais complexo, mas oferece melhor desempenho médio, sendo amplamente utilizado em implementações práticas de bibliotecas de ordenação. O Shell Sort ocupa uma posição intermediária, oferecendo desempenho razoável com implementação relativamente simples, especialmente quando comparado a outros algoritmos mais avançados.

8. Conclusão

A partir da pesquisa e dos experimentos realizados, foi possível perceber que algoritmos de ordenação, embora resolvam o mesmo problema, apresentam comportamentos bastante diferentes em termos de desempenho, uso de memória e facilidade de implementação. O estudo conjunto de Bubble Sort, Quick Sort e Shell Sort permitiu visualizar bem esse contraste.

O Bubble Sort mostrou–se o método mais simples de compreender e implementar, o que o torna adequado para fins didáticos e para vetores muito pequenos. Entretanto, sua complexidade quadrática nos casos médio e pior ($O(n^2)$) o torna pouco indicado para aplicações reais que envolvem grandes volumes de dados.

O Quick Sort, por sua vez, apresentou melhor equilíbrio entre custo computacional e eficiência prática. Apesar de exigir maior cuidado na implementação, principalmente na função de partição e no controle da recursão, o algoritmo atinge complexidade média de O(n log n) e é amplamente utilizado em bibliotecas de ordenação. Seu ponto fraco é a possibilidade de pior caso quadrático ($O(n^2)$) e o fato de não ser estável.

Já o Shell Sort ocupou uma posição intermediária. Ele permanece relativamente simples de implementar, utiliza apenas memória adicional constante e, na prática, oferece desempenho superior ao dos algoritmos quadráticos puros, embora em geral fique abaixo de algoritmos O(n log n) bem otimizados. Sua eficiência depende fortemente da escolha da sequência de gaps, o que é ao mesmo tempo uma vantagem (possibilidade de ajuste) e uma limitação (ausência de uma configuração universalmente ótima).

Além da análise teórica, os testes práticos realizados com vetores de 1.000, 5.000, 10.000 e 50.000 elementos confirmaram os comportamentos esperados. O Bubble Sort, embora muito simples de implementar, apresentou tempos de execução significativamente maiores para entradas grandes, tornando-se inviável na prática para grandes volumes de dados. O Quick Sort, mesmo com implementação mais complexa, mostrou o melhor desempenho médio entre os três, com tempos muito baixos mesmo para 50.000 elementos. O Shell Sort manteve uma posição intermediária, oferecendo um compromisso interessante entre simplicidade de código e desempenho.

Em termos de uso de memória, as medições indicaram que todos os métodos operam de forma in-place, com pouca memória adicional além do vetor de dados. Dessa forma, a escolha entre eles, na maioria dos cenários, deve considerar principalmente o custo de tempo de execução e a facilidade de implementação, mais do que a memória

De forma geral, pode-se concluir que não existe um “melhor” algoritmo de ordenação absoluto, mas sim métodos mais adequados para cada contexto. Para fins de ensino e compreensão inicial do conceito de ordenação,

o Bubble Sort é suficiente. Para aplicações práticas em que o desempenho médio é prioridade, o Quick Sort tende a ser mais apropriado. Em cenários intermediários, em que se busca um bom compromisso entre simplicidade e desempenho, o Shell Sort surge como uma alternativa interessante.

Assim, o estudo comparativo realizado nesta atividade reforça a importância de conhecer diferentes algoritmos de ordenação e de saber escolher a técnica mais adequada às características do problema, ao tamanho dos dados e às restrições de recursos do sistema.

REFERÊNCIAS

Livro(s):

PUGA, Sandra G.; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 3. ed. São Paulo: Pearson, 2016.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi. Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java. 3. ed. São Paulo: Pearson Prentice Hall, 2008.