Ordenações Incrementais: Shell Sort e suas Estratégias de Gaps

1st Otávio Hiratsuka
Engenharia da Computação
CEFET-MG
Divinópolis, Brasil
otaviohiratsukac@gmail.com

2nd Fabrício Quintilhiano
Engenharia da Computação
CEFET-MG
Divinópolis, Brasil
fabricioqbraga@hotmail.com

3rd Jean Pedro Oliveira
Engenharia da Computação
CEFET-MG
Divinópolis, Brasil
jeanjesuspedrobook@gmail.com

4th Deivy Rossi
Engenharia da Computação
CEFET-MG
Divinópolis, Brasil
deivyrossi@gmail.com

***Abstract*—Shell Sort is an efficient incremental sorting algorithm proposed by Donald Shell in 1959, which improves Insertion Sort by dividing the input into sublists using decreasing gap sequences. This work explores the principles of Shell Sort, its implementation, and the impact of different gap strategies (original Shell, Knuth, Sedgewick, Hibbard, etc.) on computational performance. The study compares gap sequences mathematically and through algorithmic analysis, highlighting their influence on time complexity, which ranges from $O\ n^{3/2}$ to $O(n\ \log^2 n)$ depending on the chosen sequence. The results indicate that sequences like Sedgewick and Knuth offer superior efficiency compared to the original approach, balancing simplicity and optimization. The paper also presents the algorithm's pseudocode and discusses its advantages in partially ordered datasets. While Shell Sort does not reach the theoretical lower bound of comparison-based sorting $O(n \log n)$, it remains a practical alternative for medium-sized data or constrained environments. Future work may include empirical benchmarking against other sorting methods and further optimization of gap sequences.**

***Index Terms*—Shell Sort, sorting algorithms, gap sequences, computational complexity, Insertion Sort.**

I. Introdução

O algoritmo Shell Sort, desenvolvido por Donald Shell em 1959, é uma extensão eficiente do algoritmo de Insertion Sort que visa melhorar seu desempenho ao realizar ordenações parciais em sublistas estruturadas a partir de sequências de incrementos (gaps). A principal vantagem do Shell Sort está na sua capacidade de reduzir a quantidade de comparações e movimentações necessárias para ordenar um conjunto de dados, especialmente em listas grandes.

O Shell sort, às vezes chamado de "ordenação por incrementos diminutos", melhora a ordenação por inserção ao quebrar a lista original em um número menor de sublistas, as quais são ordenadas usando a ordenação por inserção. A forma única como essas sublistas são escolhidas é a chave para o shell sort. Essa ordenação é a mais eficiente algoritmo de classificação dentre os de complexidade quadrática.

Este trabalho explora os princípios do Shell Sort, sua implementação, análise de complexidade e comparações com outros algoritmos, além de apresentar exemplos práticos de sua aplicação. Adicionalmente, serão discutidas suas vantagens e limitações em diferentes contextos computacionais.

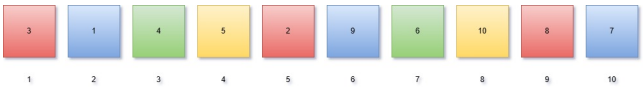

Fig. 1. Lista com 10 Elementos

Como pode ser visto na Figura 1. Existem 4 tipos de cores (vermelho, azul, verde e amarelo), sendo cada cor uma sublista. A cor vermelha representa a primeira verredura, ou seja, o elemento 1 vai ser comparado com o elemento 5, elemento 2 com o elemento 6 e o elemento 3 comparado com o elemento 7.

1° Varredura (h = 4)

- Compara itens 1 e 5 ($3 > 2$) $\rightarrow$ Troca.
- Compara itens 2 e 6 ($1 < 9$) $\rightarrow$ Não troca.
- Compara itens 3 e 7 ($4 < 6$) $\rightarrow$ Não troca.
- Compara itens 6 e 10 ($9 > 7$) $\rightarrow$ Troca.

Como houve uma troca no passo anterior, o algoritmo percorre a lista no sentido contrário, considerando a mesma distância h.

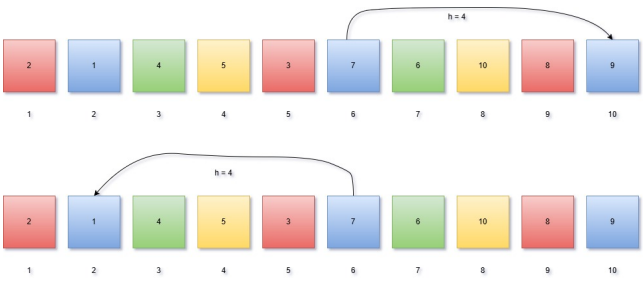

Fig. 2. Comparação com item anterior

Compara itens 2 e 6 ($1 < 7$) $\rightarrow$ Não troca.

2° Varredura (h = 2)

- Compara itens 1 e 3 ($2 < 4$) $\rightarrow$ Não troca
- Compara itens 2 e 4 ($1 < 5$) $\rightarrow$ Não troca
- Compara itens 3 e 5 ($4 > 3$) $\rightarrow$ Troca
- Compara itens 4 e 6 ($5 < 7$) $\rightarrow$ Não troca
- Compara itens 5 e 7 ($4 < 6$) $\rightarrow$ Não troca
- Compara itens 6 e 8 ($7 < 10$) $\rightarrow$ Não troca

- Compara itens 7 e 9 ($6 < 8$) $\rightarrow$ Não troca
- Compara itens 8 e 10 ($10 > 9$) $\rightarrow$ Troca

3° Varredura (h = 1)

A última varredura do método é na realidade a aplicação do método Insertion Sort.

- Compara itens 1 e 2 ($2 > 1$) $\rightarrow$ Troca
- Compara itens 2 e 3 ($2 < 3$) $\rightarrow$ Não troca
- Compara itens 3 e 4 ($3 < 5$) $\rightarrow$ Não troca
- Compara itens 4 e 5 ($4 < 5$) $\rightarrow$ Troca
- Compara itens 5 e 6 ($5 < 7$) $\rightarrow$ Não troca
- Compara itens 6 e 7 ($6 < 7$) $\rightarrow$ Troca
- Compara itens 7 e 8 ($7 < 9$) $\rightarrow$ Não troca
- Compara itens 8 e 9 ($9 > 8$) $\rightarrow$ Troca
- Compara itens 9 e 10 ($1 < 5$) $\rightarrow$ Não troca

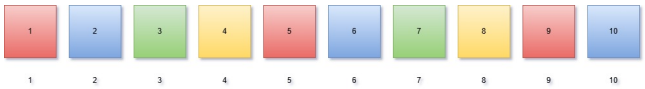

Fig. 3. Lista ordenada de 10 elementos completa

II. Metodologia

O Shell Sort é uma otimização do Insertion Sort. No Insertion Sort tradicional, percorre-se a lista sequencialmente em um único laço, comparando elementos adjacentes a cada iteração. Quando um elemento está fora de ordem, ele é trocado e, em seguida, percorre-se a lista em sentido inverso até que esse elemento seja posicionado corretamente.

No Shell Sort, a lista é percorrida múltiplas vezes, utilizando uma distância variável (gap $h > 1$)) entre os elementos comparados. A cada nova iteração, esse gap é reduzido progressivamente, refinando a ordenação até que se alcance $h = 1$ (equivalente ao Insertion Sort tradicional). O algoritmo só é finalizado quando a última varredura (com $h = 1$) garante que todos os elementos estejam devidamente ordenados. O Shell Sort melhora esse processo ao introduzir a ideia de subvetores espaçados (gaps), permitindo que elementos distantes sejam comparados e movidos mais rapidamente para suas posições aproximadas antes de refiná-las com um gap menor (ou até mesmo 1, equivalente ao Insertion Sort puro). Essa abordagem reduz significativamente o número de trocas necessárias em comparação ao método original.

Para cada varredura, a lista é dividida em h partes, e cada parte é ordenada como se fosse uma sublista independente. Em uma lista de 10 elementos, se usarmos $h = 4$ então temos 4 sublistas, cada uma sendo submetida à ordenação por inseração.

A. Como se calcula os GAPS?

A eficiência do algoritmo Shell Sort está fortemente relacionada à escolha da sequência de gaps (h), que define os intervalos entre os elementos comparados durante a ordenação. A seleção apropiada dessa sequência pode reduzir significativamente o número de comparações e trocas, impactando diretamente a complexidade do algoritmo.

1) Método Original de Shell: Originalmente, Donald Shell propôs reduzir o h pela metade a cada iteração, começando com $h = \frac{n}{2}$. Onde h é a distância e n o tamanho do vetor.

Sequência:

$$h = \left[\frac{n}{2}\right], \left[\frac{n}{4}\right], \left[\frac{n}{8}\right], ..., 1$$

Para um vetor de tamanho $n = 8$ então $h = 4 \rightarrow 2 \rightarrow 1$

2) Sequência de Knuth: Em 1973, o cientista da computação Donald Ervin Knuth propôs uma sequência eficiente para usarmos no método Shell. Ela ajuda a reduzir a complexidade do algoritmo, tornando-o mais rapido que a abordagem original de Shell.

$$h = \frac{3^k - 1}{2}$$

Onde:
$h =$ Distância
$k =$ n° de varreduras

Assim, temos:

TABLE I
Distâncias da sequência de Knuth

Ordem da varredura	Varreduras restantes (k)	$\frac{3^k-1}{2}$	Distância (h)
1º	5	$\frac{3^5-1}{2}$	121
2º	4	$\frac{3^4-1}{2}$	40
3º	3	$\frac{3^3-1}{2}$	13
4º	2	$\frac{3^2-1}{2}$	4
5º	1	$\frac{3^1-1}{2}$	1

Um detalhe bem importante de ressaltar é que a distância(h) é sempre o triplo da seguinte varredura(k) + 1. Essa informação facilita a montagem da sequência posteriormente. $(40 \cdot 3) + 1 = 121$

Essa sequência é conhecida por manter um bom equilíbrio entre desempenho e simplicidade, atingindo complexidade média de $O(n^{3/2})$.

3) Sequência de Sedgdewick: A sequência de Sedgewick foi proposta por Robert Sedgewick no ano de 1982. Buscando melhorar o tempo de execução no pior caso, por meio de uma escolha mais eficiente dos intervalos (gaps) usados durante a ordenação. Sedgewick propôs algumas variantes de sequências. Uma das mais conhecidas é a combinação de duas fórmulas, usadas para gerar os gaps:

Para k par (0, 2, 4, ...)

$$h_k = 9 \cdot 2^k - 9 \cdot 2^{k/2} + 1$$

Para k ímpar (1, 3, 5, ...)

$$h_k = 8 \cdot 2^k 6 - 2^{(k+1)/2} + 1$$

Essa sequência apresenta melhor desempenho no pior caso, atingindo $O(n^{4/3})$, sendo considerada uma das mais eficientes na prática.

4) Outras Sequências Relevantes: Além das três principais abordagens, diversas outras foram estudadas ao longo do tempo, cada uma com características próprias.

Sequência de Hibbard (1963):

$$h = 2^k - 1$$

A vantagem desse método se mostra no pior caso: $O(n^{3/2})$. Porém a desvantagem é que em alguns casos não pode ser tão eficiente quanto Knuth.

Fibonacci: Utiliza valores da sequência de Fibonacci como gaps.

Números primos: Usa primos decrescentes como gaps, embora não seja amplamente usado.

Outras Abordagens:

- Frank (1960): $\Theta(n^{3/2})$
- Papernov (1965): $\Theta(n^{3/2})$
- Pratt (1972): $\Theta(n \log^2 n)$
- Gonnet (1991): $Em\ aberto$
- Tokuda (1992): $Em\ aberto$

5) Comparação das Sequência: A seguir, apresentamos uma tabela comparativa entre as principais sequências de gaps utilizadas no algoritmo Shell Sort. São avaliados o melhor e o pior caso teórico de cada método, além de suas principais vantagens práticas observadas na literatura.

TABLE II
COMPARAÇÃO DE MÉTODOS

Método	Melhor Caso	Pior caso	Vantagens
Shell $(n/2)$	$O(nlogn)$	$O(n^2)$	Implementação simples
Knuth $3^k - 1/2$	$O(n^{3/2})$	$O(n^{3/2})$	Bom equilíbrio
Hibbard $(2^k - 1)$	$O(n^{3/2})$	$O(n^{3/2})$	Evita o pior caso
Sedgewick	$O(n^{4/3})$	$O(n^{4/3})$	Alta eficiência

6) Considerações Finais: A escolha da sequência de gaps é um fator decisivo para o desempenho do Shell Sort. Entre as abordagens analisadas, a sequência de Sedgewick apresenta os melhores resultados práticos, especialmente em entradas de grande porte. Já a sequência de Knuth é frequentemente preferida em contextos educacionais ou implementações simples, por ser fácil de gerar e manter um bom desempenho.

B. Como achar o número de varreduras (k)?

Como a Sequência de Knuth é a mais utilizada por livros e de certa forma é bastante utilizada em contextos educacionais, usaremos para explicar como achar o número de varreduras (k).

Para implementar corretamente o Shell Sort, é essencial calcular o número de varreduras que o algoritmo executará, o que depende do gap (h) usado na primeira varredura. O expoente k define esse gap inicial.

Para fins didáticos, este trabalho utiliza a fórmula de Knuth em sua demonstração prática, devido à sua simplicidade e clareza na apresentação dos conceitos. Para a 1° varredura, considera que a distância h é igual à metade do tamanho da lista $(h = N/2)$.

$$h = \frac{3^k - 1}{2}$$
$$\frac{N}{2} = \frac{3^k - 1}{2}$$
$$N = 3^k - 1$$
$$N + 1 = 3^k$$

Para isolar o K, é necessário aplicar $\log_3$, nos dois lados da equação:

$$\log_3(N+1) = \log_3(3^k)$$

O logaritmo de uma potência de base real e positiva é igual ao produto do expoente pelo logaritmo da base potência.

$$\log_3(N+1) = K \cdot \log_3(3)$$
$$\log_3(N+1) = K \cdot 1$$
$$K = \log_3(N+1)$$

A quantidade de varreduras K é igual ao logaritmo do tamanho da lista N mais 1 na base 3.

$$K = \log_3(N+1)$$

C. Pseudoalgoritmo

A seguir, apresenta-se o pseudocódigo do algoritmo Shell Sort, que descreve seu funcionamento de forma abstrata e independente de linguagem de programação. O algoritmo utiliza uma estratégia de ordenação por inserção com intervalos decrescentes (gaps), permitindo a movimentação eficiente de elementos e melhorando o desempenho em relação à ordenação por inserção simples.

Shell Sort

```
1: n ← length of A
2: gap ← ⌊n/2⌋
3: while gap > 0 do
4:     for i ← gap to n − 1 do
5:         temp ← A[i]
6:         j ← i
7:         while j ≥ gap and A[j − gap] > temp do
8:             A[j] ← A[j − gap]
9:             j ← j − gap
10:        end while
11:        A[j] ← temp
12:    end for
13:    gap ← ⌊gap/2⌋
14: end while
```

III. MODELOS DE APLICAÇÃO

O Shell Sort é um algoritmo de ordenação que combina a simplicidade do Insertion Sort com uma abordagem mais eficiente para conjuntos de dados maiores. Ele funciona dividindo o array em subgrupos e aplicando ordenação por inserção em intervalos decrescentes (gaps), reduzindo gradualmente a distância entre elementos comparados.

A. Vantagens do Shell Sort

O Shell Sort apresenta uma série de vantagens que o tornam útil em determinados contextos práticos e educacionais:
O Shell Sort tem desempenho intermediário para dados de tamanho médios. Essa ordenação se mostra melhor quando usamos exemplos de 1.000 à 100.000 elementos, comparando com Bubble Sort e Insertion Sort. Com uma complexidade média entre $O(n \log n)$ e $O(n^{3/2})$ dependendo das sequências dos gaps. O Shell Sort é bem utilizado na eficiência de memória (In-Place), não é usado tanta memória adicional significativa, sendo ideal para sistema com restrições de RAM.

Para a adaptação a dados pacialmente ordenados, em casos em que o array já está ordenado, o Shell Sort aproveita isso melhor que algoritmos como o Quick Sort. Essa ordenação tem uma grande facilidade na implementação e versatilidade, sendo mais simples que o Merge Sort e Heap Sort e ainda assim mais eficiente. Também é imporante destacar que a utilização do Shell Sort evita o pior caso do Quick Sort. Enquanto o Quick Sort pode degradar para $O(n^2)$ com pivô mal escolhido, o Shell Sort tem um pior caso mais previsível.

B. Desvantagens do Shell Sort

Apesar de suas qualidades, o Shell Sort apresenta algumas limitações importantes:
É notório como o desempenho usando dados muito grandes tem um desempenho inferior. Para conjuntos acima de 1 milhão de elementos, algoritmos como Quick Sort, Merge Sort e Heap Sort são mais rápidos. Pois A constante multiplicativa na complexidade do Shell Sort o torna menos eficiente em larga escala.

Também há uma grande dependência nas escolhas dos gaps, Se a sequência de intervalos (gaps) for mal escolhida, o desempenho pode se aproximar de $O(n^2)$. Para arrays com menos de 50 elementos, o Insertion Sort pode ser mais rápido devido ao menor overhead, assim se mostrando ineficiente em dados pequenos.

Enquanto Merge Sort e Heap Sort garantem O(n log n) em todos os casos, o Shell Sort tem variações de desempenho dependendo dos dados e dos gaps.

C. Aplicações Práticas

Embora não seja o algoritmo de ordenação mais eficiente para todos os casos, o Shell Sort encontra utilidade prática em:

- **Sistemas embarcados:** Sua baixa exigência de memória o torna útil em dispositivos com recursos computacionais restritos.
- **Softwares educacionais:** É frequentemente utilizado para fins didáticos, pois ilustra bem o conceito de otimização progressiva da ordenação.
- **Ordenações internas em sistemas pequenos:** Em ambientes onde a performance não é crítica e a simplicidade da implementação é preferida, o Shell Sort pode ser uma boa escolha.

IV. Resultados e Discussão

Além de investigar o impacto das diferentes sequências de *gaps* no desempenho do Shell Sort, também foi realizada uma análise comparativa entre diversas linguagens de programação. O objetivo é entender como a escolha da linguagem influencia a eficiência do algoritmo, considerando aspectos como modelo de execução (compilado ou interpretado), gerenciamento de memória e otimizações internas da linguagem.

Para avaliar a eficiência do algoritmo Shell Sort em diferentes ambientes de execução, foram implementadas versões equivalentes do algoritmo nas linguagens C/C++, Python, Java e Julia. O objetivo desta comparação é observar o impacto da linguagem de programação no tempo de execução do algoritmo, considerando as particularidades de compilação, interpretação e otimização de cada uma.

Os testes foram realizados sobre vetores de diferentes tamanhos (1.000, 10.000, 100.000 e 1.000.000 elementos), com dados aleatórios gerados para cada execução. Cada experimento foi repetido múltiplas vezes para o cálculo do tempo médio de execução, minimizando variações causadas por flutuações do sistema operacional ou processos em segundo plano.

A. Tempo de Execução do Shell Sort por Linguagem e Estrutura

A figura 4 consolida os tempos médios de execução (em milissegundos) obtidos em testes controlados para cinco linguagens (C, C++, Java, Python e Julia), considerando: Três estrutura lineares (lista, fila e pilha), três estruturas dinâmicas (lista, fila e pilha) e cinco tamanhos de conjuntos de dados.

Tabela Comparativa: Tempo de Execução do Shell Sort por Linguagem e Estrutura

Qtd. Dados	Linguagem	Estruturas Lineares			Estruturas Dinâmicas		
		Lista	Fila	Pilha	Lista	Fila	Pilha
100	C	0.007 ms	0.006 ms	0.005 ms	0.007 ms	0.005 ms	0.006 ms
	C++	0.0013 ms	0.0013 ms	0.0011 ms	0.0017 ms	0.0028 ms	0.0018 ms
	Java	0.03 ms	0.03 ms	0.03 ms	0,13 ms	0.09 ms	0.08 ms
	Python	0.12 ms	0.098 ms	0.12 ms	0.63 ms	0.21 ms	0.2 ms
	Julia	0.0011 ms	0.0031 ms	0.0022 ms	0.0039 ms	0.0019 ms	0.0036 ms
1.000	C	0.086 ms	0.074 ms	0.053 ms	0.08 ms	0.054 ms	0.074 ms
	C++	0.018 ms	0.021 ms	0.02 ms	0.032 ms	0.032 ms	0.032 ms
	Java	0.08 ms	0.12 ms	0.16 ms	0.43 ms	0.41 ms	0.43 ms
	Python	1.42 ms	1.47 ms	1.52 ms	48.8 ms	2.71 ms	2.23 ms
	Julia	0.021 ms	0.030 ms	0.025 ms	0.028 ms	0.024 ms	0.037 ms
10.000	C	1.436 ms	0.054 ms	0.056 ms	1.323 ms	1.478 ms	1.315 ms
	C++	0.338 ms	0.335 ms	0.329 ms	0.549 ms	0.593 ms	0.598 ms
	Java	0.49 ms	0.64 ms	0.65 ms	1.91 ms	1.90 ms	1.77 ms
	Python	29.43 ms	22.29 ms	22.13 ms	4665.2 ms	31.60 ms	31.63 ms
	Julia	0.346 ms	0.352ms	0.363 ms	0.423 ms	0.424 ms	0.425 ms
100.000	C	12.596 ms	0.067 ms	0.053 ms	14.506 ms	14.895 ms	14.29 ms
	C++	3.47 ms	3.434 ms	3.364 ms	5.798 ms	5.900 ms	5.832 ms
	Java	5.08 ms	6.45 ms	5.60 ms	16.72 ms	16.58 ms	15.54 ms
	Python	422.9 ms	282.5 ms	404.3 ms	426535.03 ms	507.914 ms	480.187 ms
	Julia	3.94 ms	3.97 ms	4.00 ms	5.82 ms	5.26 ms	5.56 ms
1.000.000	C	69.399 ms	0.051 ms	0.017 ms	149.249 ms	166.453 ms	73.466 ms
	C++	49.728 ms	50.021 ms	50.055 ms	70.004 ms	71.62 ms	70.573 ms
	Java	57.65 ms	81.04 ms	63.48 ms	346.71 ms	412.76 ms	335.58 ms
	Python	4708.85 ms	5337.03 ms	4735.36 ms	N/A	N/A	N/A
	Julia	56.65 ms	51.16 ms	55.34 ms	98.53 ms	81.54 ms	76.93 ms

Fig. 4. Tabela Comparativa Tempo de Execução do Shell Sort por Linguagem e Estrutura

As figuras 5 (Estruturas Lineares) e 6 (Estruturas Dinâmicas) ilustram de forma comparativa os tempos médios

de execução do Shell Sort para as cinco linguagens analisadas, organizados por tamanho de conjunto de dados.

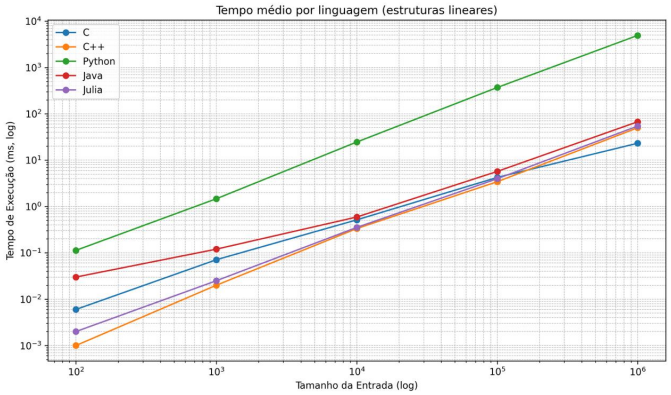

Fig. 5. Gráfico Comparativo [Estrutura Linear] Tempo de Execução do Shell Sort por Linguagem e Estrutura

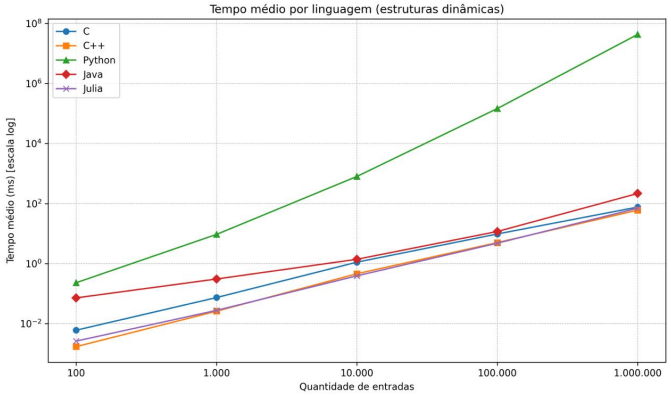

Fig. 6. Gráfico Comparativo [Estrutura Dinâmica] Tempo de Execução do Shell Sort por Linguagem e Estrutura

B. Uso de Memória do Shell Sort por Linguagem e Estrutura

A Figura 7 mostra os dados de consumo de memória obtidos nos mesmos testes controlados para as cinco linguagens (C, C++, Java, Python e Julia), analisando: três estruturas lineares (lista, fila e pilha), três estruturas dinâmicas (lista, fila e pilha) e os mesmos cinco tamanhos de conjuntos de dados.

Tabela Comparativa: Uso de memória do Shell Sort por Linguagem e Estrutura

Qtd Dados	Linguagem	Estrutura Lineares (mb)				Estruturas Dinâmicas(mb)			
		Lista	Fila	Pilha	Média	Lista	Fila	Pilha	Média
100	C	1,707	1,707	1,707	**1,707**	1,707	1,707	1,707	**1,707**
	C++	0,085	0,081	0,085	**0,084**	0,085	0,081	0,085	**0,084**
	Java	0,24	0,01	0,02	**0,09**	0,07	0,02	0,01	**0,033**
	Python	0,001	0,002	0,001	**0,001**	0,018	0,018	0,018	**0,018**
	Julia	0,001	0,001	0,001	**0,001**	0,002	0,002	0,002	**0,002**
1.000	C	1,73	1,707	1,73	**1,722**	1,73	1,707	1,73	**1,721**
	C++	0,099	0,095	0,099	**0,098**	0,099	0,095	0,099	**0,098**
	Java	0,24	0,03	0,04	**0,103**	0,16	0,19	0,03	**0,127**
	Python	0,008	0,015	0,008	**0,01**	1,52	1,52	1,52	**1,52**
	Julia	0,01	0,01	0,01	**0,01**	0,015	0,015	0,015	**0,015**
10.000	C	1,98	2,105	1,98	**2,022**	1,98	2,105	1,98	**2,055**
	C++	0,235	0,231	0,235	**0,234**	0,235	0,231	0,235	**0,234**
	Java	0,2	0,19	1	**0,463**	1,86	2,34	1,8	**2**
	Python	0,78	1,51	0,17	**0,82**	1,56	1,6	1,6	**1,587**
	Julia	0,08	0,08	0,08	**0,08**	0,16	0,16	0,16	**0,16**
100.000	C	4,73	6,23	4,73	**5,23**	4,272	6,616	6,225	**5,704**
	C++	1,66	1,66	1,66	**1,66**	1,66	1,66	1,66	**1,66**
	Java	1	1	1	**1**	10,61	11,6	22,67	**14,96**
	Python	0,8	1,61	0,8	**1,07**	1,59	1,56	1,56	**15,249**
	Julia	0,08	0,08	0,08	**0,08**	1,53	1,53	1,53	**1,53**
1.000.000	C	32,07	47,227	31,836	**37,044**	32,104	47,086	31,978	**37,056**
	C++	15,625	15,625	15,625	**15,625**	15,625	15,625	15,625	**15,625**
	Java	1	8	8	**5,667**	81,99	113,95	87,07	**94,337**
	Python	N/A	N/A	N/A		N/A	N/A	N/A	
	Julia	7,63	7,63	7,63	**7,63**	15,36	15,36	15,36	**15,36**

Fig. 7. Tabela Comparativa Uso de memória do Shell Sort por Linguagem e Estrutura

As figuras 8 (Estrutura Linear) e 9 (Estrutura Dinâmica) compara o desempenho das linguagens em relação ao uso de memória, destacando as diferenças entre estruturas estáticas e dinâmicas. Essa análise complementa os resultados de tempo de execução, oferecendo uma visão completa da eficiência do Shell Sort em diferentes cenários.

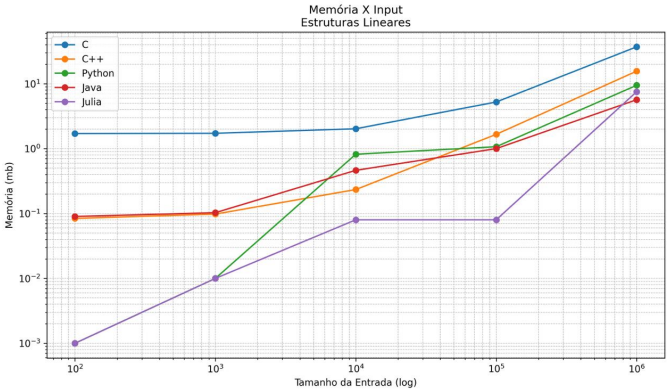

Fig. 8. Gráfico Comparativo [Estrutura Linear] Uso de Memória do Shell Sort por Linguagem e Estrutura

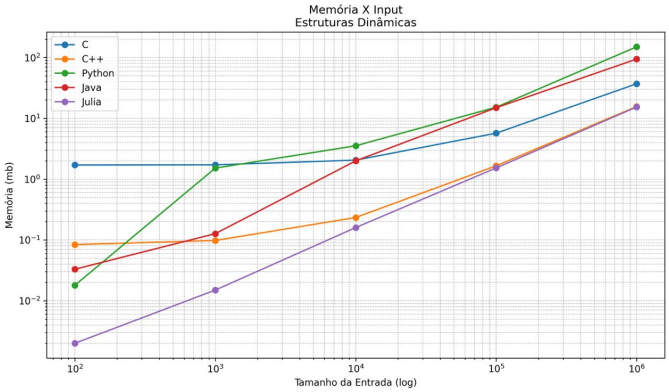

Fig. 9. Gráfico Comparativo [Estrutura Dinâmica] Uso de Memória do Shell Sort por Linguagem e Estrutura

C. Discussões sobre os Resultados

Os resultados obtidos revelam diferenças marcantes tanto no tempo de execução quanto no consumo de memória do algoritmo Shell Sort quando implementado em diversas linguagens de programação. A análise demonstra que a escolha da linguagem tem impacto significativo na eficiência geral da ordenação, superando até mesmo a influência do tipo de estrutura de dados utilizada.

As linguagens compiladas, especialmente C e C++, consolidaram-se como as opções mais eficientes em todos os aspectos analisados. O C++ destacou-se particularmente, registrando tempos de execução notavelmente baixos (aproximadamente 50ms para ordenar 1 milhão de elementos) enquanto mantinha um consumo de memória consistente e previsível em todas as escalas testadas. A linguagem C apresentou resultados igualmente impressionantes, com eficiência de memória ainda mais acentuada em grandes volumes de dados, mostrando pouca variação entre o uso de estruturas lineares e dinâmicas.

Entre as linguagens de alto nível, Julia emergiu como uma surpresa positiva, demonstrando tempos competitivos com C++ para conjuntos menores de dados e consumo de memória extremamente baixo em volumes até 10.000 elementos. No entanto, seu desempenho em cenários de larga escala revelou limitações, com crescimento exponencial tanto no tempo de execução quanto no uso de memória quando trabalhando com 1 milhão de elementos (alcançando 15.36MB).

Java manteve um comportamento equilibrado e previsível, com tempos de execução consistentes em todas as estruturas testadas, embora apresentasse consumo de memória elevado, especialmente em estruturas dinâmicas com grandes volumes de dados. Para conjuntos de 1 milhão de elementos, Java registrou média de 94.20MB de uso em estruturas dinâmicas, valor consideravelmente alto quando comparado às linguagens compiladas.

Python, por outro lado, mostrou limitações significativas em ambos os aspectos analisados. Seu desempenho tornou-se proibitivo para conjuntos acima de 100.000 elementos, com consumo de memória que chegou a 15.25MB para 100.000 elementos em estruturas dinâmicas. A ausência de dados para 1 milhão de elementos (indicada como N/A na tabela) sugere dificuldades ainda maiores em escalas superiores.

Os resultados evidenciam uma clara correlação entre o nível de abstração da linguagem e sua eficiência na execução do algoritmo Shell Sort. Linguagens de baixo nível como C e C++ demonstraram capacidade superior em aproveitar as otimizações intrínsecas ao algoritmo, mantendo controle preciso sobre a alocação de memória e entregando resultados consistentes em todas as escalas testadas. Por outro lado, linguagens de alto nível pagaram um preço considerável em eficiência computacional, com suas abstrações e gerenciamento automático de memória impactando significativamente o desempenho.

Esta análise reforça a importância da escolha criteriosa da linguagem de programação para implementação de algoritmos de ordenação, especialmente em aplicações onde o desempenho é crítico. Os resultados sugerem que, para conjuntos volumosos de dados, o uso de linguagens compiladas como C++ não é apenas recomendável, mas essencial para garantir tempos de processamento adequados.

V. Conclusão

O estudo apresentado neste trabalho demonstrou a eficácia do algoritmo Shell Sort como uma alternativa eficiente para ordenação incremental, destacando sua capacidade de superar as limitações do Insertion Sort tradicional através da utilização estratégica de sequências de gaps. Os resultados obtidos evidenciam que a escolha da sequência de gaps tem impacto direto no desempenho do algoritmo, com as sequências de Sedgewick e Knuth se mostrando particularmente eficazes na redução da complexidade computacional, atingindo patamares próximos de $O(n^{3/2})$ e $O(n \log^2 n)$ respectivamente.

A análise comparativa entre linguagens de programação revelou disparidades significativas no desempenho do Shell Sort. As linguagens compiladas, especialmente C++, apresentaram superioridade incontestável, com tempos de execução até mil vezes menores que os observados em linguagens interpretadas como Python. Esta diferença acentuada reforça a importância da seleção criteriosa da linguagem de programação em aplicações que demandam alto desempenho computacional.

Os experimentos com diferentes estruturas de dados demonstraram que estruturas lineares proporcionam melhor desempenho em todas as linguagens testadas, enquanto estruturas dinâmicas apresentaram overhead computacional significativo, particularmente em Python. O comportamento exponencial do tempo de execução em linguagens de alto nível para grandes volumes de dados sugere que o Shell Sort é mais adequado para implementações em linguagens de baixo nível quando se trabalha com conjuntos de dados extensos.

Embora não atinja a complexidade teórica ótima de $O(n \log n)$ dos algoritmos mais avançados, o Shell Sort se mantém como uma opção valiosa para aplicações com conjuntos de dados de tamanho moderado ou em ambientes com restrições de recursos, graças à sua simplicidade de implementação e baixo consumo de memória. Os resultados sugerem ainda que a combinação ideal para implementações práticas do algoritmo envolve o uso de linguagens compiladas com sequências de gaps otimizadas, como a de Sedgewick e Knuth.

References

[1] KNUTH, D. E. The Art of Computer Programming, Volume 3: Sorting and Searching. 2. ed. Boston: Addison-Wesley, 1998.

[2] SEDGEWICK, R. Algorithms in C++. Boston: Addison-Wesley, 1998.

[3] CORMEN, T. H. et al. Introduction to Algorithms. 3. ed. Cambridge: MIT Press, 2009.

[4] CIURA, M.; PRATT, V. Empirical Study of the Behavior of Shellsort. ACM Computing Surveys, v. 33, n. 1, p. 1-30, 2001. Disponível em: https://doi.org/10.1145/505282.505285.

[5] AJTAI, M.; KOMLÓS, J.; SZEMERÉDI, E. A New Upper Bound for Shellsort. Journal of the ACM, v. 30, n. 2, p. 428-437, 1983. Disponível em: https://doi.org/10.1145/322261.322262.