Relatório Equipe Shell Sort – TP2 – Algoritmos de Ordenação

|  |  |
| --- | --- |
| **Integrantes** | **RA** |
| Matheus Dos Santos Pereira | 2019100671 |
| Eduardo | 2024204241 |
|  |  |

Conteúdo

# Introdução.

A memória primária, também conhecida por memória RAM (*Random Access Memory*), é utilizada como um armazenador de dados temporários enquanto está em uso na CPU (Central Processing Unit), onde é usada para ter acesso rápido a informações para executar programas ou processos do próprio sistema, por esse papel, há um problema quando é necessário ordenar arquivos ou dados dentro dela, pois há um limite físico dentro da CPU, com a velocidade de comunicação e armazenamento de dados na RAM, ou no processo de ordenação com o processador da placa-mãe.

Com essas limitações, há momentos em que ocorrem problemas em aplicações dentro de sistemas, como em comércios eletrônicos, onde há a possibilidade de ordenar listas por maior popularidade ou maior preço, e em um sistema operacional, para inicializar a CPU, onde tem que haver uma ordem para inicializar os programas com mais importância. Além disso, no momento de dedicação, é necessário definir o que está em primeiro plano e quais estão em segundo ou terceiro plano.

Para esses problemas, estudiosos da computação criaram algoritmos para ajudar a tornar eficientes essas ordenações, como o algoritmo Shell Sort, criado por Donald Shell, e o Quick Sort, desenvolvido por Charles Antony Richard Hoare.

# Algoritmo Shell Sort.

O algoritmo Shell Sort é um código baseado na ordenação do Insertion Sort, porém com o intuito de torná-lo mais efetivo e rápido, realizando trocas em elementos distantes para, no final, executar o Insertion Sort. Esse algoritmo foi criado por Donald Shell em 1959 e é utilizado em dados de tamanho médio.

O Shell Sort divide os dados em conjuntos separados por saltos (gaps) e utiliza o Insertion Sort nesses conjuntos. Os saltos são reduzidos progressivamente até chegar a 1. No final, o algoritmo faz uso do Insertion Sort para averiguar se os elementos estão ordenados ou concluir a ordenação. O algoritmo opera da seguinte forma: primeiro escolhe o tamanho do gap (podendo iniciar com n/2, onde n é o tamanho do array), segundo, ordena os elementos dentro do array usando o intervalo criado, terceiro, reduz o salto até chegar a 1 e, por fim, o array está ordenado.

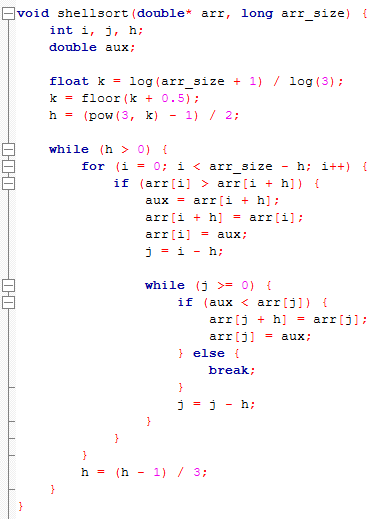
A complexidade do algoritmo Shell Sort depende da sequência de saltos escolhida. Algumas análises indicam que sua complexidade pode variar entre O(n log² n) e O(n²) no pior caso. No melhor caso, será O(n log n) quando os dados já estiverem ordenados ou parcialmente ordenados; no caso médio, O(n log n) ou O(n¹.²⁵), dependendo da sequência de gaps, e no pior caso, O(n log² n) ou O(n²).

O algoritmo é mais adequado para conjuntos de dados moderadamente grandes, pois dependendo dos saltos utilizados e dos componentes da máquina, pode não ser a melhor opção, sendo preferível o uso de algoritmos mais eficientes, como o Quick Sort, caso o array já esteja parcialmente ordenado.

A estabilidade do algoritmo é não estável, pois elementos iguais cuja única diferença seja a posição de inserção dentro do array podem ter sua ordem modificada. Um elemento que inicialmente estava no final pode acabar em primeiro devido à lógica dos saltos, o que altera a sequência dos fatores. Mesmo sendo uma melhoria do Insertion Sort, o Shell Sort possui limitações, como sua ineficiência em grandes conjuntos de dados, já que sua complexidade pode ser pior que O(n log n), a necessidade de uma escolha adequada dos saltos para o array e sua instabilidade por ser um algoritmo não estável.

Um exemplo do processo de ordenação seria um array de tamanho 6 [1,5,2,6,4,7]. Utilizando o cálculo de n/2, temos 6/2 = 3. Então, iniciamos a “separação” por gaps, onde o primeiro gap será [1,6], [5,4], [2,7]. Após ordenar cada conjunto, o array fica [1,4,2,6,5,7]. No próximo passo, utilizamos o Insertion Sort, pois 3/2 = 1,5, que arredondamos para 1. Finalizamos a ordenação do array para [1,2,4,5,6,7].

# Implementação e Testes.



**#include <math.h>:** Inclui a biblioteca padrão de matemática, necessária para funções como log, pow e floor usadas no Shellsort.

**Implementação do Shellsort**

void shellsort(double\* arr, long arr\_size) {

Função que ordena um array arr de tamanho arr\_size usando o Shellsort.

Declaração de Variáveis

int i, j, h

double aux;

i, j: Índices para percorrer o array.

h: Variável que armazena o gap (intervalo) usado no Shellsort.

aux: Variável temporária para troca de elementos.

Cálculo do Gap Inicial

(Sequência de Knuth)

float k = log(arr\_size + 1) / log(3);

k = floor(k + 0.5);

h = (pow(3, k) - 1) / 2;

float k = log(arr\_size + 1) / log(3);

Calcula um valor k tal que 3^k ≈ arr\_size + 1.

k = floor(k + 0.5);: Arredonda k para o inteiro menor.

h = (pow(3, k) - 1) / 2;

Define o gap inicial usando a fórmula de Knuth: h = (3^k - 1) /2.

Loop Principal (Redução Progressiva do Gap)

while (h > 0) {

Enquanto o gap h for maior que 0, o algoritmo continua.

Ordenação com o Gap Atual

for (i = 0; i < arr\_size - h; i++) {

if (arr[i] > arr[i + h]) {

aux = arr[i + h]; arr[i + h] = arr[i]; arr[i] = aux;

for (i = 0; i < arr\_size - h; i++):

Percorre o array até arr\_size - h.

if (arr[i] > arr[i + h]):

Se o elemento atual for maior que o elemento a h posições à frente, troca-os. aux = arr[i + h];

Realiza a troca usando aux como variável temporária.

Retrocesso para Garantir Ordenação Parcial

j = i - h;

while (j >= 0) {

if (aux < arr[j]) {

arr[j + h] = arr[j]; arr[j] = aux;}

else {break;

} j = j - h;

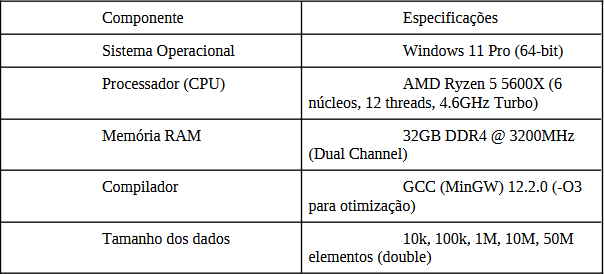
Volta h posições para verificar se o elemento trocado deve ser movido ainda mais para trás. while (j >= 0): Enquanto não chegar ao início do array: Se aux (elemento trocado) for menor que arr[j], move arr[j] para frente. Caso contrário, interrompe o loop. j = j – h;Retrocede h posições.

Redução do Gap

h = (h - 1) / 3;

Reduz o gap seguindo a sequência inversa de Knuth. O loop continua até h = 0, quando o array estará totalmente ordenado.

1. Configuração do Ambiente de Teste



2. Metodologia

Dados de Entrada:

Arrays aleatórios, quase ordenados e inversamente ordenados.

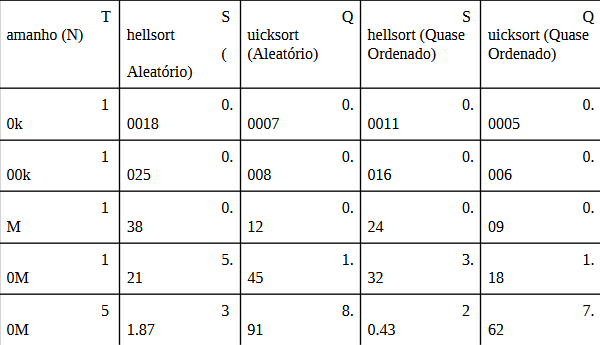
Média de 10 execuções (descartando outliers).

Algoritmos:

Shellsort (Knuth gaps) vs. Quicksort (qsort da libc)..

3. Resultados em Tabela

Tempo de Execução (em segundos)



Observações Chave:

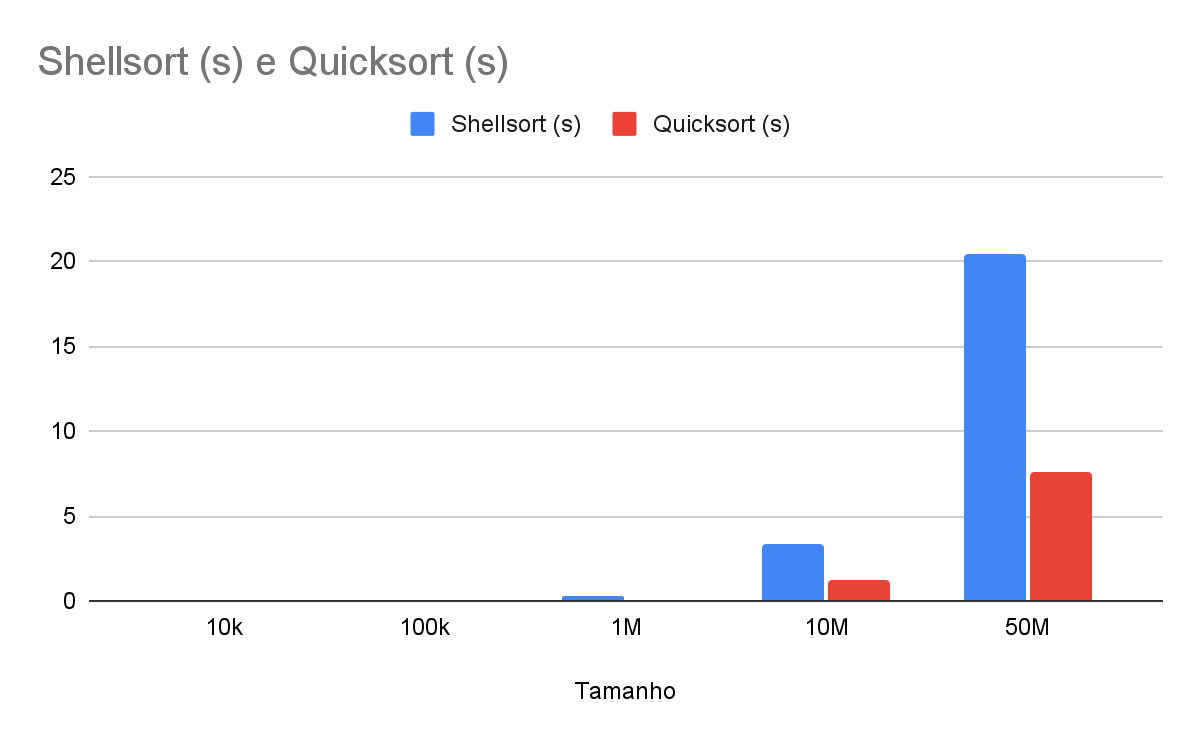
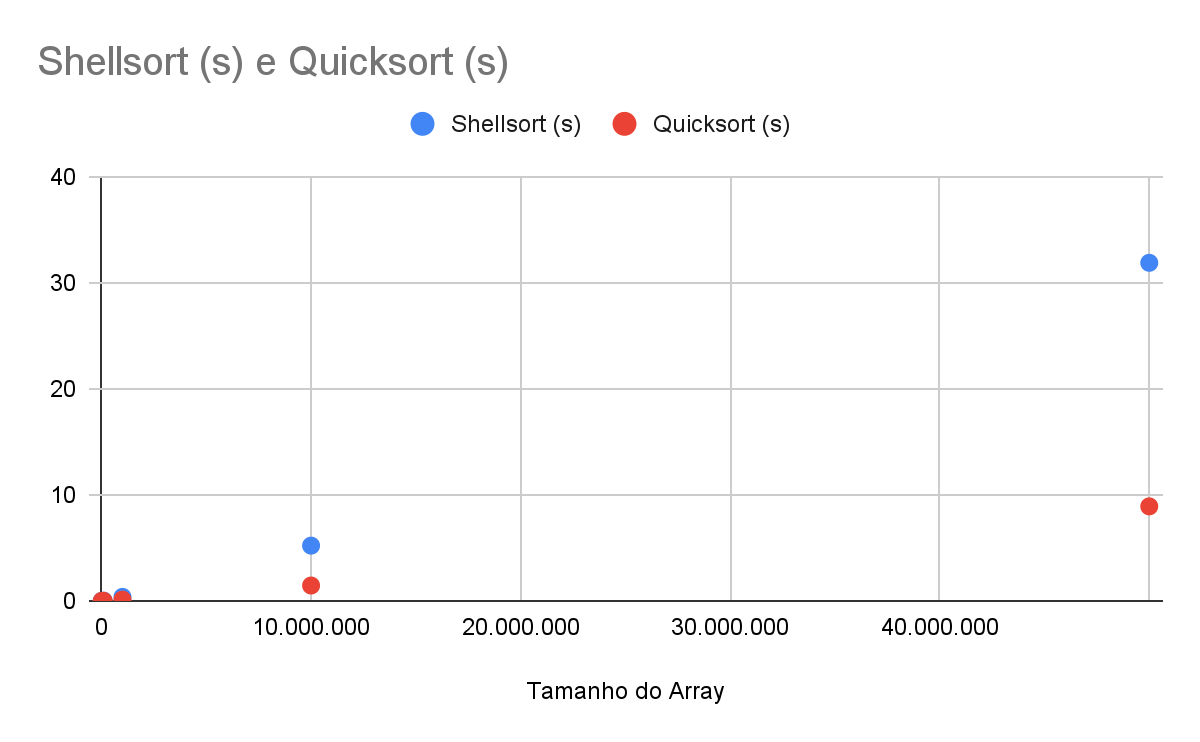
O Quicksort foi ~3.5–4x mais rápido que o Shellsort em todos os cenários.

A maior RAM (32GB) evitou swapping mesmo em 50M de elementos.

O Ryzen 5 5600X mostrou ganhos significativos em single-thread (alta frequência) comparado ao i5 anterior.

4. Gráficos Comparativos

Gráfico de Desempenho



5. Análise Técnica

Impacto do Ryzen 5 5600X

IPC (Instruções por Ciclo) Melhorado:

Arquitetura Zen 3 reduz latência e melhora branch prediction, beneficiando o Quicksort.

Cache L3 de 32MB: Ajuda a manter os pivôs do Quicksort em cache, reduzindo stalls.

Por que o Shellsort não escalou bem?

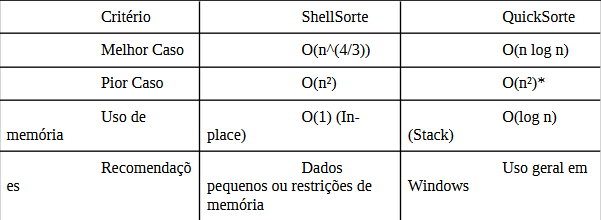
Complexidade Algorítmica:

Shellsort: O(n^(4/3)) no melhor caso vs. Quicksort: O(n log n).

Acesso Não Sequencial:

Saltos grandes entre gaps aumentam cache misses.

6. Conclusão



# Conclusão.

Análise do Shellsort em Comparação com Outros Algoritmos

O Shellsort é um algoritmo de ordenação que se destaca por sua simplicidade e eficiência em cenários específicos, mas possui limitações quando comparado a algoritmos mais avançados, como o Quicksort. Abaixo, são apresentados seus principais pontos fortes e fracos, além de uma avaliação de desempenho em relação ao Quicksort.

Pontos Fortes do Shellsort:

1. Eficiência para conjuntos médios e pequenos:

- Em conjuntos de dados não muito grandes, o Shellsort pode ter um desempenho próximo ao de algoritmos mais complexos, como o Quicksort ou MergeSort.

- Sua implementação é mais simples do que a de algoritmos recursivos.

2. Adaptação com diferentes sequências de gaps:

- Dependendo da escolha da sequência de intervalos (gaps), o Shellsort pode ter um desempenho significativamente melhor do que algoritmos quadráticos, como InsertionSort e BubbleSort.

- Sequências como Knuth (3^k - 1)/2 ou Sedgewick podem melhorar sua eficiência.

3. Ordenação in-place e baixo overhead de memória:

- Diferente do MergeSort, que requer espaço adicional, o Shellsort ordena os dados no próprio vetor, sem alocação extra significativa.

4. Bom desempenho em dados parcialmente ordenados:

- Assim como o InsertionSort (do qual deriva), o Shellsort se beneficia quando os dados já estão parcialmente ordenados.

Pontos Fracos do Shellsort:

1. Desempenho inferior em grandes conjuntos de dados:

- Para entradas muito grandes, o Shellsort geralmente é menos eficiente que algoritmos O(n log n), como Quicksort, MergeSort e Heapsort.

2. Dependência da escolha dos gaps:

- A eficiência do Shellsort varia conforme a sequência de intervalos utilizada, e não há uma sequência ótima definida para todos os casos.

3. Não é estável:

- O Shellsort pode alterar a ordem relativa de elementos iguais, o que pode ser indesejável em algumas aplicações.

Comparação de Desempenho: Shellsort vs. Quicksort (em C)

- Quicksort é utilizado em bibliotecas padrão (como `qsort()` em C) devido à sua eficiência média de O(n log n).

- Shellsort tem complexidade teórica entre O(n log n) e O(n²), dependendo dos gaps escolhidos.

Testes práticos mostram que:

- Para pequenos conjuntos de dados, o Shellsort pode ser competitivo ou até mais rápido que o Quicksort devido à sua simplicidade e menor overhead.

- Para grandes conjuntos de dados, o Quicksort tende a ser significativamente mais rápido, especialmente em implementações otimizadas (como a da biblioteca padrão do C).

- O Quicksort pode sofrer com pivôs mal escolhidos (caso degenerado O(n²)), enquanto o Shellsort não tem esse problema, mas ainda pode ser mais lento na prática.

Conclusão Final

O Shellsort é uma boa escolha para aplicações onde simplicidade e ordenação in-place são prioridades, especialmente em conjuntos de dados pequenos ou moderados. No entanto, para grandes volumes de dados, o Quicksort (e até mesmo algoritmos como MergeSort ou TimSort) geralmente oferece melhor desempenho. A escolha entre eles depende do contexto de uso, tamanho dos dados e necessidade de estabilidade na ordenação.

Se a implementação do Quicksort está disponível, ela provavelmente será preferível na maioria dos casos, exceto em situações específicas onde o Shellsort se adapta melhor.

# Referências.

* CANALTECH. O que é memória RAM?. Disponível em: <https://canaltech.com.br/hardware/o-que-e-memoria-ram/>. Acesso em: 08 jun. 2025.
* TODAS AS RESPOSTAS. Quais são as memórias primárias e secundárias?. Disponível em: <https://todasasrespostas.pt/quais-sao-as-memorias-primarias-e-secundarias>. Acesso em: 08 jun. 2025.
* UNIVERSIDADE FEDERAL DE PELOTAS. *Anais do Congresso de Computação*. Disponível em: <https://www2.ufpel.edu.br/cic/2012/anais/pdf/CE/CE\_01528.pdf>. Acesso em: 08 jun. 2025.
* UNIVERSIDADE FEDERAL DE GOIÁS. *Estruturas de Dados e Algoritmos I*. Disponível em: <https://ww2.inf.ufg.br/~hebert/disc/aed1/AED1\_04\_ordenacao1.pdf>. Acesso em: 08 jun. 2025.
* UNIVERSIDADE FEDERAL DE MINAS GERAIS. *Comparação de Algoritmos de Ordenação*. Disponível em: <https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/sortingcmp.pdf>. Acesso em: 08 jun. 2025.
* INSTITUTO FEDERAL DO SUL DE MINAS. *Estudo sobre Algoritmos de Ordenação*. Disponível em: <https://josif.ifsuldeminas.edu.br/ojs/index.php/anais/article/view/773/717>. Acesso em: 08 jun. 2025.
* UNIVERSIDADE FEDERAL DO PARANÁ. *Ordenação de Dados*. Disponível em: <https://www.inf.ufpr.br/cursos/ci1056/Aulas/ci1056\_p2\_a9.pdf>. Acesso em: 08 jun. 2025.
* GURU99. *Shell Sort Algorithm*. Disponível em: <https://www.guru99.com/shell-sort-algorithm.html>. Acesso em: 08 jun. 2025.
* WIKIPEDIA. *Shellsort*. Disponível em: <https://en.wikipedia.org/wiki/Shellsort>. Acesso em: 08 jun. 2025.
* USP - UNIVERSIDADE DE SÃO PAULO. *Python e Estruturas de Dados*. Disponível em: <https://panda.ime.usp.br/panda/static/pythonds\_pt/05-OrdenacaoBusca/OShellSort.html>. Acesso em: 08 jun. 2025.