




Algoritmos de Ordenação Avançados

SHELL SORT



Centro Universitário Ruy Barbosa
Disciplina: Algoritmos e Complexidade
Professor: Heleno Cardoso
Aluno: Jônatas Gomes

Índice

Introdução

Visão Geral de Algoritmos de Ordenação

A Necessidade do Shell Sort

Limitações dos Algoritmos de Ordenação Convencionais

Algoritmo de Ordenação por Inserção

Necessidade de um Algoritmo Eficiente

Shell Sort como Solução

Conceito Básico

Abordagem Divide e Conquista

Gap e Comparação

Gradual Redução das Lacunas

Eficiência

Passo a Passo

Vantagens do Shell Sort

Limitações do Shell Sort

Aplicações Práticas do Shell Sort

Exemplo

Agradecimentos



Introdução

Algoritmos de ordenação desempenham um papel fundamental na computação e são aplicados em uma variedade de cenários, desde sistemas de bancos de dados até a organização de dados para apresentação ao usuário. Por definição o Shell Sort é um algoritmo de ordenação que se destaca por sua abordagem dividida e conquistada para classificar elementos em uma lista. Ele é uma extensão e melhoria do Insert Short (algoritmo de ordenação por inserção) e foi proposto por Donald Shell em 1959 como uma melhoria a fim de solucionar os problemas e limitações do modelo citado anteriormente.





Visão Geral de Algoritmos de Ordenação

1

São procedimentos utilizados para organizar um conjunto de elementos em uma ordem específica, seja em ordem crescente ou decrescente. Esses algoritmos desempenham um papel fundamental na computação.

2

Os algoritmos de ordenação são aplicados em uma variedade de cenários, sendo essenciais em: Sistemas de Bancos de Dados e Organização de Dados.

3

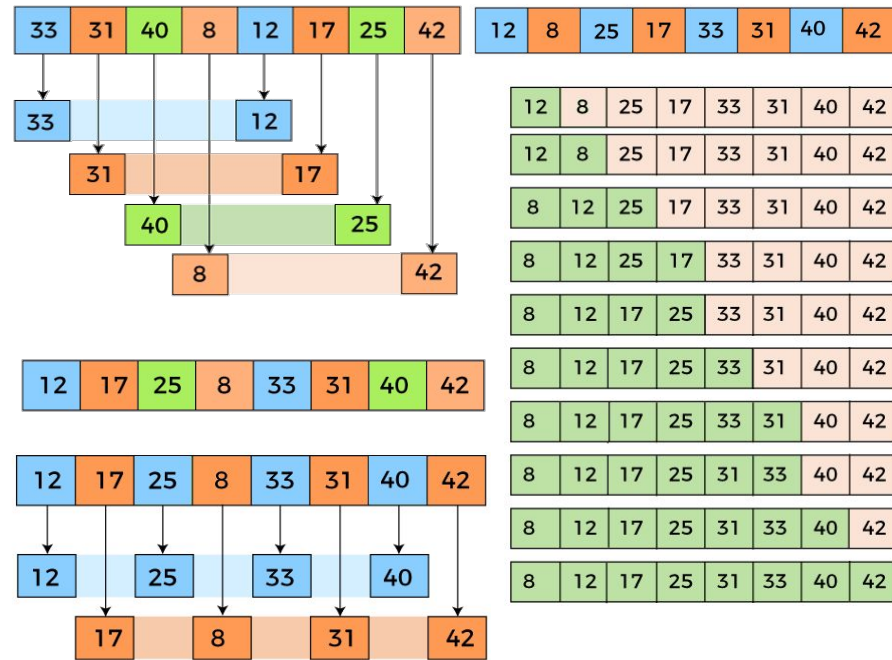
A eficiência dos algoritmos de ordenação é crucial, especialmente em conjuntos de dados grandes. A escolha do algoritmo certo pode impactar significativamente o desempenho de um sistema.

4

Existem muitos algoritmos de ordenação diferentes, cada um com suas características e eficiência em cenários específicos.

A Necessidade do Shell Sort

- 01 | Limitações dos Algoritmos de Ordenação Convencionais
- 02 | Algoritmo de Ordenação por Inserção
- 03 | Necessidade de um Algoritmo Eficiente
- 04 | Shell Sort como Solução





Limitações dos Algoritmos de Ordenação Convencionais

Muitos algoritmos de ordenação, como o Bubble Sort e o Selection Sort, têm desempenho insatisfatório em conjuntos de dados grandes.



Algoritmo de Ordenação por Inserção

O algoritmo de ordenação por inserção (Insertion Sort) é eficaz para conjuntos de dados pequenos, mas seu desempenho diminui drasticamente à medida que o tamanho dos dados aumenta.



Necessidade de um Algoritmo Eficiente

A necessidade de um algoritmo de ordenação eficiente que possa lidar com conjuntos de dados de tamanhos variados se tornou evidente.

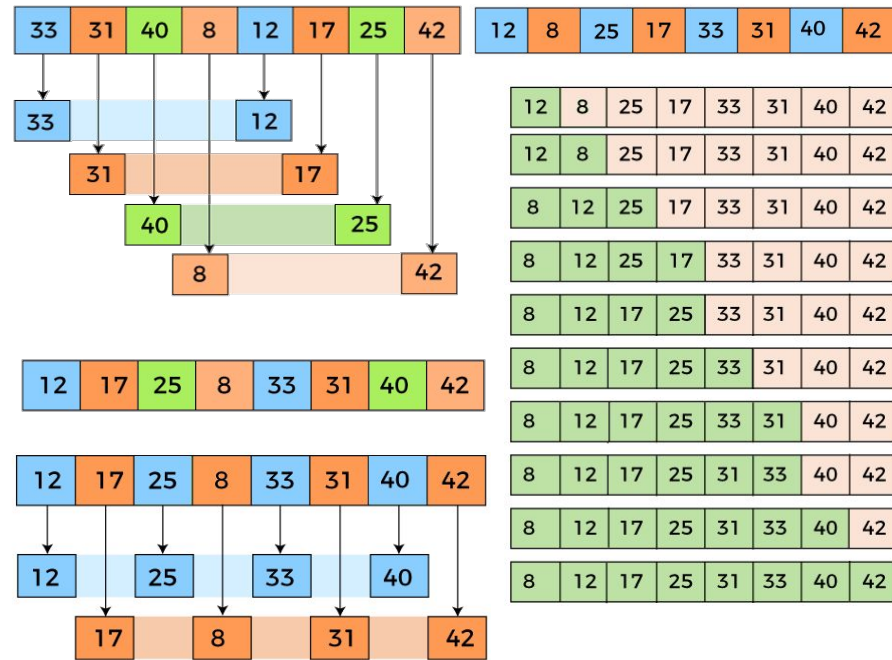


Shell Sort como Solução

O Shell Sort foi desenvolvido para superar essas limitações. Ele é uma extensão e melhoria do Insertion Sort e oferece um desempenho significativamente melhor em uma ampla gama de cenários.

Conceito Básico do Shell Sort

- 01 | Abordagem Divide e Conquista
- 02 | Gap e Comparação
- 03 | Gradual Redução das Lacunas
- 04 | Eficiência





Abordagem Divide e Conquista

O Shell Sort é um algoritmo de ordenação que segue a abordagem "divide e conquista". Ele divide o conjunto de dados em subconjuntos menores, ordena esses subconjuntos e, em seguida, combina-os em um único conjunto ordenado.



Gap e Comparação

O Shell Sort introduz o conceito de "gap" (lacuna). Os elementos em uma lista são comparados com outros elementos afastados por uma determinada lacuna, em vez de serem comparados diretamente uns aos outros.



Gradual Redução das Lacunas

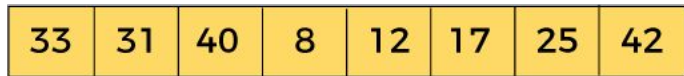
O algoritmo começa com lacunas maiores e, gradualmente, reduz o tamanho das lacunas à medida que a lista se torna mais ordenada. Isso contribui para o melhor desempenho do Shell Sort.



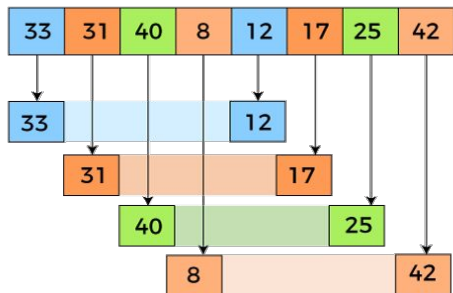
Eficiência

A abordagem divide e conquista, juntamente com a manipulação cuidadosa das lacunas, permite que o Shell Sort alcance um desempenho melhor em comparação com algoritmos de ordenação mais simples, especialmente em conjuntos de dados de tamanhos variados.

Passo a passo



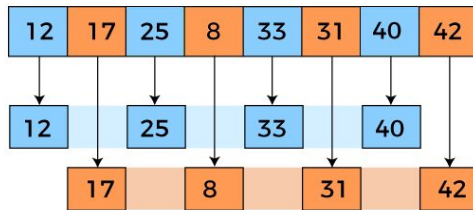
01 | Define o array inicial



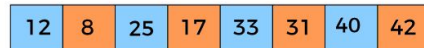
02 | Define o gap para metade do array total e ordena-o



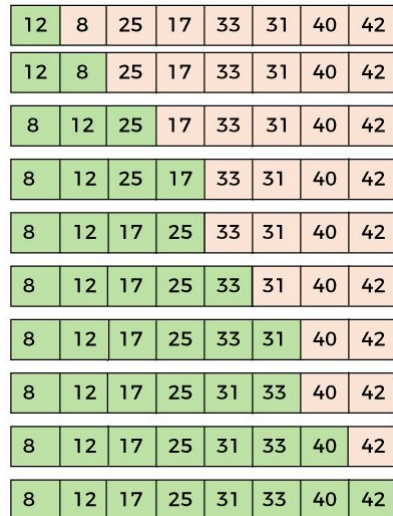
03 | Recebe o novo array após a primeira ordenação



04 | Define o gap para metade do que era anteriormente e ordena-o



05 | Recebe o novo array após a segunda ordenação





Vantagens do Shell Sort

1

O Shell Sort é conhecido por superar algoritmos de ordenação mais simples em termos de desempenho, especialmente em conjuntos de dados de tamanho moderado.

2

O algoritmo é flexível e pode ser adaptado para diferentes cenários, ajustando a escolha das sequências de lacunas.

3

O Shell Sort é um algoritmo estável, o que significa que ele não altera a ordem relativa de elementos iguais além disso, ele apresenta um desempenho razoável mesmo em conjuntos de dados parcialmente ordenados, o que é uma vantagem em muitos casos do mundo real.

4

O Shell Sort é comumente usado em sistemas embarcados, sistemas de gerenciamento de banco de dados e sistemas de arquivos, onde o desempenho é fundamental.



Limitações do Shell Sort

1

A implementação do Shell Sort pode ser mais complexa em comparação com algoritmos de ordenação mais simples, como o Bubble Sort ou o Selection Sort.

2

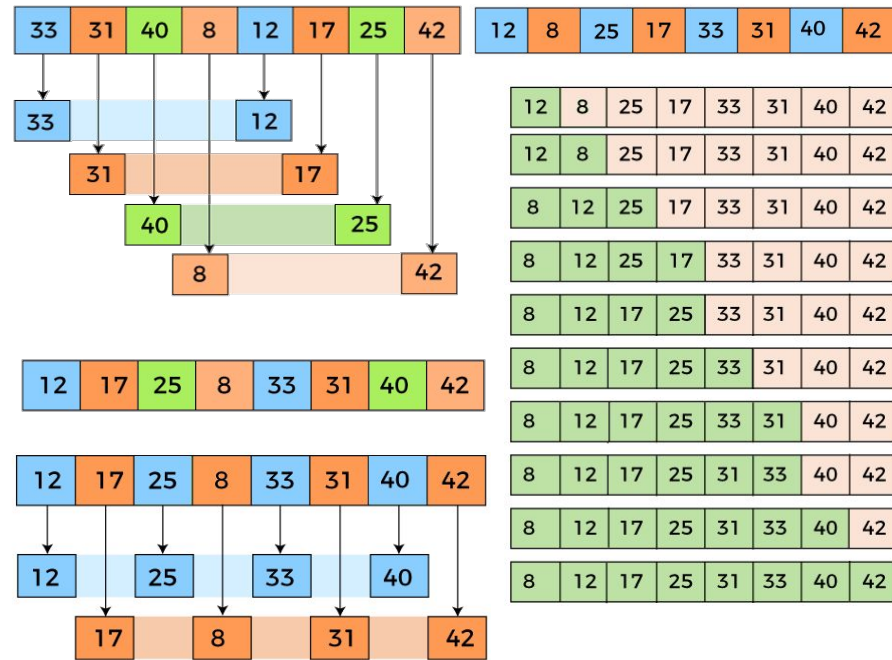
O desempenho do Shell Sort é altamente dependente da escolha das sequências de lacunas, e a seleção incorreta pode levar a um desempenho ruim.

3

Embora seja mais eficiente do que muitos algoritmos simples, o Shell Sort não é o algoritmo de ordenação mais rápido disponível.

Aplicações Práticas do Shell Sort

- 01 | Sistemas de Gerenciamento de Banco de Dados
- 02 | Sistemas de Arquivos
- 03 | Sistemas Embarcados
- 04 | Aplicações Científicas
- 05 | Aplicações de Rede





Exemplo

```
1 function shellSort(itensArray) {
2   const n = itensArray.length;
3   let gap = Math.floor(n / 2);
4
5   while (gap > 0) {
6     for (let i = gap; i < n; i++) {
7       let temp = itensArray[i];
8       let j = i;
9
10      while (j >= gap && itensArray[j - gap] > temp) {
11        itensArray[j] = itensArray[j - gap];
12        j -= gap;
13      }
14
15      itensArray[j] = temp;
16    }
17    gap = Math.floor(gap / 2);
18  }
19 }
20
21 const itensArray = [12, 1, 54, 3, 10, 47, 2];
22 shellSort(itensArray);
23 console.log("Sorted array is:", itensArray);
24 |
```

```
Sorted array is: [
  1,  2,  3, 10,
  12, 47, 54
]
```



Obrigado!

