FACULDADE DE TECNOLOGIA DE PRAIA GRANDE

CENTRO PAULA SOUZA
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LUIZ MIGUEL SANTOS RODRIGUES
RICARDO CIDELINO EVANGELISTA

ALGORITMOS DE ORDENAÇÃO
Shell Sort, Merge Sort, Quick Sort

PRAIA GRANDE
2021

1. INTRODUÇÃO

O algoritmo é uma solução para o problema. Pode ser implementado com qualquer sequência de valores ou objetos com lógica infinita (como português, Pascal, C, sequências numéricas, um conjunto de objetos (como lápis e borrachas)), sim, qualquer coisa que possa fornecer uma sequência lógica.

Podemos usar receitas culinárias como exemplo para ilustrar um algoritmo, embora muitos algoritmos sejam mais complicados. O algoritmo mostra passo a passo os passos necessários para resolver o problema.

O objetivo desta pesquisa é avaliar experimentalmente os dados gerados usando algoritmos de ordenação *Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort* e *Shell Sort*, bem como, suas utilidades e eficiências.

1.1 Quick Sort

Entre os algoritmos de ordenação interna é um dos mais rápidos e com uma ampla usabilidade o algoritmo foi inventado por C. A. R. Hoare em 1960, enquanto visitava a Universidade de Moscou como estudante, o algoritmo foi publicado poucos anos depois por Hoare, em 1962, após muitas revisões.

1.2 Merge Sort

O algoritmo visa reordenar estruturas lineares destruindo, fundindo e conectando n elementos existentes. Em outras palavras, a estrutura a ser reordenada será subdividida recursivamente em estruturas menores até que não seja mais possível fazê-lo. Em seguida, os elementos serão organizados para ordenar cada subestrutura. Feito isso, as subestruturas menores (agora ordenadas) serão mescladas e seus elementos serão classificados por intercalação. Repita o mesmo processo até que todos os elementos sejam combinados em uma única estrutura organizacional.

1.3 Shell Sort

Desenvolvido pelo cientista da computação e matemático Donald Shell em 1959, o algoritmo de Shell Sort, tem a proposta de ser um revisão do algoritmo de ordenação por inserção, o método criado por Donald foi publicado na Universidade de Cincinnati adotando o seu sobrenome.

2. COMO FUNCIONA

2.1 Quick Sort

O algoritmo de *Quick Sort*, utiliza a estratégia denominada divisão e conquista, que funciona ordenando um vetor com *n* valores em dois vetores menores, em seguida os vetores menores são ordenados independentemente e depois os resultados são combinados para produzir uma vetor organizado. O momento mais delicado deste método é em relação ao procedimento de partição dos vetores, o qual tem que rearranjar o vetor a esquerda, representado na figura 01, pelo traço em preto, e um outro vetor a direita, representado pelo traço vermelho e escolha arbitrária de um item x do vetor chamado de pivô, de tal forma que ao final o vetor está particionado em uma parte esquerda com chaves menores ou iguais ao pivô e uma parte a direita com chaves maiores ou iguais a pivô.

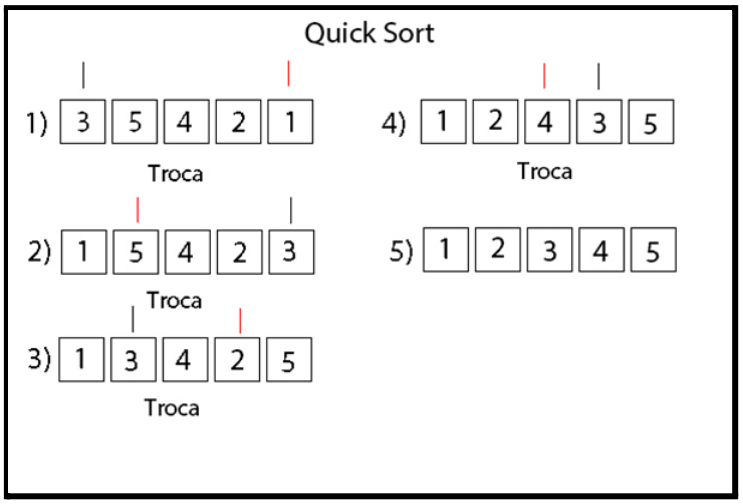

Figura 01: Processo de particionamento utilizando *Quick sort.*

Após obter os dois vetores menores através do particionamento, onde cada pedaço é ordenado recursivamente, temos a troca de posições com o pivô, até o este encontrar sua posição de fato, onde temos o fim do algoritmo de ordenamento.

2.2 Merge Sort

O método Merge Sort foi criado para resolver problemas de entradas grandes, visa separar todos os itens do vetor, em sub listas, até que haja apenas um item em cada uma dessas listas, depois ele re-ordenará na ordem contrária, agrupando de 2 em 2, de 4 em 4 e assim sucessivamente(divisão).

Cada pedaço dessa entrada é tratado separadamente(conquista) e ao final, os resultados parciais são combinados para gerar o resultado final esperado.

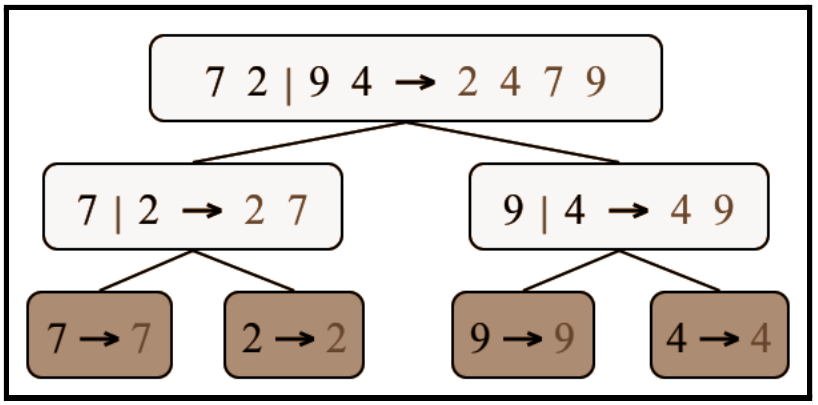

Figura 02: Árvore de recursividade para o *Merge Sort.*

A técnica de divisão e conquista consiste de 3 passos:

- Divisão: Dividir o problema original em subproblemas menores;

- Conquista: Resolver cada subproblema recursivamente;

- Combinação: Combinar as soluções encontradas, compondo uma solução para o problema original.

2.3 Shell Sort

Para analisar o método de *Shell Sort*, deve-se elucidar acerca do método de ornamento por inserção, que consiste na troca de itens adjacentes quando se está procurando o ponto de inserção na sequência destino. Se o menor item estiver na

posição mais à direita no vetor então o número de comparações e movimentações é igual a n-1 para encontrar seu ponto de inserção.

Contudo Donal Shell percebeu que este método se torna ineficiente quando tem-se vetores maiores e com desorganização complexa, sendo assim o método de Shell Sort propõe contornar este problema permitindo trocas de registros que estão distantes um do outro, os itens que são separados "*h"* posições são rearranjados de tal forma que todo h-ésimo item leva uma sequência ordenada. Tal sequência é dita estar h-ordenada. A figura 3 mostra como um arquivo é ordenado usando os incrementos 4, 2 e 1 para h.

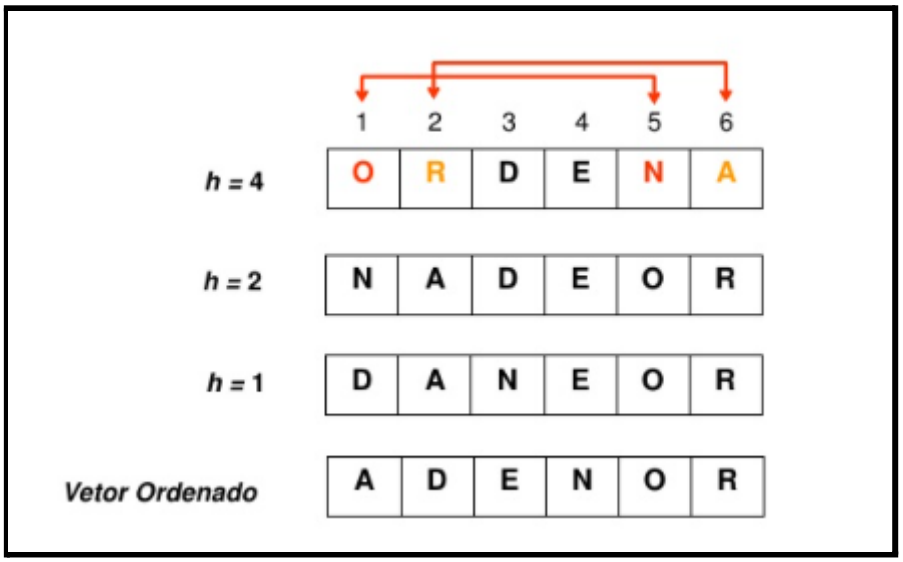

Figura 03: Exemplo de ordenamento de Shell Sort.

3. MELHOR E PIOR CENÁRIO

3.1.1 Quick Sort pior cenário

Uma característica interessante do *Quick Sort* é a sua ineficiência para arquivos já ordenados quando a escolha do pivô é inadequada. Por exemplo, a escolha sistemática dos extremos de um arquivo já ordenado leva ao seu pior caso. Neste caso, as partições serão extremamente desiguais, e o procedimento "quicksort" será chamado recursivamente "n" vezes, eliminando apenas um item em cada chamada. Esta situação é desastrosa pois o número de comparações passa a

cerca de “n²/2”, e o tamanho da pilha necessária para as chamadas recursivas é cerca de “n”. Entretanto, o pior caso pode ser evitado através de pequenas modificações no programa.

3.1.2 Quick Sort melhor cenário

A melhor situação possível ocorre quando cada partição divide o arquivo em duas partes exatamente iguais. Logo, “C(n) = 2C(n/2) + n” onde “C(n/2)” representa o custo de ordenar uma das metades e “*n”* é o custo de examinar cada item. A solução para esta recorrência é “C(n) - 1,4n” o que significa que em média o tempo de execução do *Quick Sort* é o “O(n log n)”, conforme a figura 04.

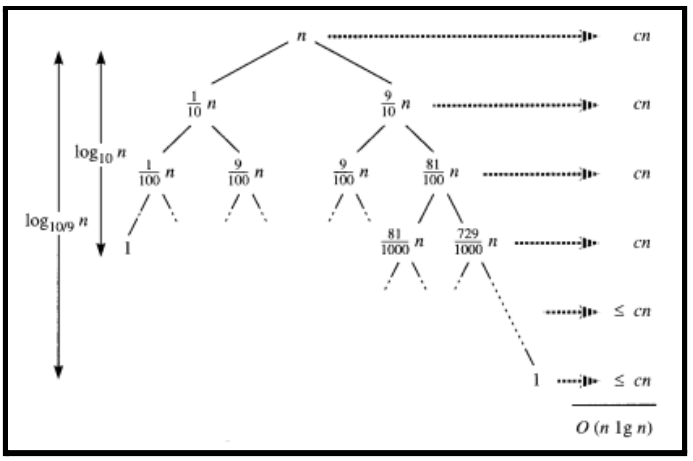

Figura 04: Árvore de recursividade para o *Quick Sort.*

Quick Sort é extremamente eficiente para ordenar arquivos de dados. O método necessita de apenas uma pequena pilha como memória auxiliar, e requer cerca de n log n operações em média para ordenar n itens. Como aspectos negativos cabe ressaltar que: (i) a versão recursiva do algoritmo tem um pior caso que é 0(n2) operações; (ii) a implementação do algoritmo é muito delicada e :difícil: um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados; (iii) o método não é estável. Uma vez que se consiga uma implementação robusta para o Quicksort, este deve ser o algoritmo preferido para a maioria das aplicações. No caso de se necessitar de um programa utilitário para uso frequente, então vale a pena investir na obtenção de uma implementação do algoritmo. Por exemplo, como evitar o pior caso do algoritmo? A melhor solução para este caso é

escolher três itens quaisquer do arquivo e usar a mediana dos três como o item divisor na partição.

3.2.1 Merge Sort pior cenário

Na prática, é mais lento que QuickSort no caso médio, requer um vetor auxiliar, o que aumenta o consumo de memória e tempo de execução.

3.2.2 Merge Sort melhor cenário

Eles requerem menos acessos à memória e são altamente paralelos. Se vários processadores estiverem disponíveis, essa estratégia aumentará a eficiência e é adequada para aplicativos que têm um limite de tempo (sempre executado em n em um horário específico). Pode ser convertido para um formato estável, seja extremamente cuidadoso ao realizar a intercalação e fácil de implementar.

3.3.1 Shell Sort pior cenário

Esta é a complexidade de resolver o problema para a pior entrada do tamanho de h, onde o pior cenário é dado quando “O(n)”.

3.3.2 Shell Sort melhor cenário

A complexidade de resolver um problema de melhor entrada de tamanho h, ocorre quando “O(n log2 n)”.

4. APLICABILIDADE

4.1 Aplicabilidade do algoritmo *Quick Sort*

Por se tratar de um algoritmo de alta complexidade, ele é utilizado em aplicações que necessitam uma alta velocidade de resposta, sendo o mais rápido em tempo de resposta, quando o ordenamento é feito de forma decrescente.

Algoritmo	Tempo(ms)	Comparações	Movimentações
Bubble sort	1455,9734	50005000	74237889
Selection Sort	545,1068	49995000	29997
Insertion sort	539,6891	9999	24765961
Quick sort	4,5072	176065	103635

Figura 05: Comparativo de resposta, número de comparações e movimentos

Entre os principais algoritmos do gênero, a figura 05 traz um comparativo, com vetores de ordem decrescente com 10.000 valores.

4.2 Aplicabilidade do algoritmo *Merge Sort*

Este algoritmo é adequado para o processamento de estruturas lineares, onde estruturas menores são direcionadas, como em filas de operações financeiras (bancos, day trade etc).

4.3 Shell Sort

O Shell Sort é uma opção para arquivos de tamanho moderado, mesmo porque sua implementação é simples e requer uma quantidade de código pequena. Existem métodos de ordenação mais eficientes, mas que são mais complicados de implementar. Sendo assim seu tempo de execução é sensível à ordem inicial do arquivo. Além disso, o método não é estável, pois ele nem sempre deixa registros com chaves iguais na mesma posição relativa.

5. O CÓDIGO

5.1 Quick sort (C++)

```
#include <iostream>

using namespace std;
```

```
void quicksort(int valores[], int inicio, int fim){
	int i, j, pivo, aux;
	i = inicio;
	j = fim-1;
	pivo = valores[(inicio + fim) / 2];
	while(i <= j){
		while(valores[i] < pivo && i < fim){
			i++;
		}
		while(valores[j] > pivo && j > inicio){
			j--;
		}
		if(i <= j){
			aux = valores[i];
			valores[i] = valores[j];
			valores[j] = aux;
			i++;
			j--;
		}
	}
	if(j > inicio){
		quicksort(valores, inicio, j+1);
	}
	if(i < fim){
```

```
            quicksort(valores, i, fim);
        }
}

int main(int argc, char *argv[]){
        int array[10] = {5, 8, 1, 2, 7, 3, 6, 9, 4, 10};
        cout<<"VETOR DESORGANIZADO\n";
        for(int i = 0; i < 10; i++){
            cout << array[i] << ' ';
        }
        cout << endl;
        cout<<"\n\n";
        cout<<"VETOR ORGANIZADO\n";
        quicksort(array, 0, 10);
        for(int i = 0; i < 10; i++){
            cout << array[i] << ' ';
        }
        return 0;
}
```

5.2 Merge sort (python)

```
import random
def mergeSort(lista):
    if len(lista) > 1:
        meio = len(lista)//2
```

```
    listaDaEsquerda = lista[:meio]
    listaDaDireita = lista[meio:]
    mergeSort(listaDaEsquerda)
    mergeSort(listaDaDireita)
    i = 0
    j = 0
    k = 0
    while i < len(listaDaEsquerda) and j < len(listaDaDireita):
        if listaDaEsquerda[i] < listaDaDireita[j]:
            lista[k]=listaDaEsquerda[i]
            i += 1
        else:
            lista[k]=listaDaDireita[j]
            j += 1
        k += 1
    while i < len(listaDaEsquerda):

        lista[k]=listaDaEsquerda[i]
        i += 1
        k += 1
    while j < len(listaDaDireita):
        lista[k]=listaDaDireita[j]
        j += 1
        k += 1
return lista
```

```
sorteio=random.sample(range(1,100),12)
minha_lista = sorteio
print(minha_lista)
print('\nOrdenado\n')
print(mergeSort(minha_lista))
```

5.3 Shell Sort (C++)

```
#include <iostream>
using namespace std;

void shellsort(int org[], int tam){
	for(int gap=tam/2; gap>0; gap/=2){
		for(int i = gap; i < tam; i++) {
			int temp = org[i];
			int j;
			for(j = i; j >= gap && org[j-gap] > temp; j -= gap){
				org[j]=org[j-gap];
			}
			org[j] = temp;
		}
	}
}

int main (int argc, char** argv){
	int tam;
	cout<<"Insira o tamanho do vetor: ";
	cin>> tam;
```

```
	int org[tam];
	cout<<"Insira os valores do vetor: \n";
	for(int i=0; i < tam; i++){
		cin>>org[i];
	}
	shellsort(org, tam);
	cout<<"Vetor organizado: ";
	for(int i=0; i < tam; i++){
		cout<<org[i]<<" ";
	}
	cout<<endl;
	return 0;
}
```

6. CONSIDERAÇÕES FINAIS

O Algoritmo de Ordenação é um tipo que coloca os elementos de uma dada sequência em uma certa ordem. Em outras palavras, efetua sua ordenação completa ou parcial. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista.

O algoritmo Merge Sort é geralmente expresso em algoritmos de classificação mediana. Como a recursividade é sua ferramenta principal, os melhores resultados vêm de lidar com estruturas lineares aleatórias. Porém, ao lidar com estruturas pequenas e / ou pré-determinadas (aumento ou diminuição), a recursão torna-se uma desvantagem, o que desperdiça tempo de processamento e torna as trocas desnecessárias.

Já o Shell Sort, com base nos dados deste trabalho, é o resultado mais satisfatório, principalmente porque a estrutura é grande e desorganizada. Como a classificação de shell é considerada um aprimoramento da Selection Sort, a Shell

Sort usada para o mesmo propósito de seus predecessores (ou seja, recursos que dificilmente precisam ser escritos) mostrará melhor desempenho e, portanto, estenderá a vida útil do recurso.

O Quick Sort, ao subdividir o vetor e fazer inserções diretas utilizando um valor de referência (pivô), reduz seu tempo de execução, mas, as quantidades de comparações (leitura) e, principalmente, trocas (escrita) ainda são muito altas.

Com isso, conclui-se que não existe o método certo ou errado, o que existe é o método que mais se adapta ao seu projeto.