Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Kayky Cristof Eduardo Domingos Silva" - "8118"

"Análise e comparação de algoritmos de ordenação"

Rio Paranaíba - MG

2024

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Kayky Cristof Eduardo Domingos Silva" - "8118"

"Análise e comparação de algoritmos de ordenação"

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmos da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2024

1 RESUMO

Algoritmos de ordenação são fundamentais para organizar dados em uma sequência ordenada, facilitando o acesso, análise e manipulação destes dados. Esses algoritmos recebem uma entrada e os reordenam gerando uma saída.

Existem diferentes tipos de algoritmos de ordenação, cada um com suas características e desempenho. Alguns dos mais conhecidos incluem o Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Bubble Sort. Cada um desses algoritmos é analisado de acordo com a complexidade de tempo e espaço, que mede a eficiência de execução e o uso de memória, especialmente em grandes volumes de dados. Em geral, os algoritmos de ordenação são estudados para identificar quais são mais adequados para situações específicas, como pequenas coleções de dados ou sistemas onde a memória é limitada.

Além disso, algoritmos de ordenação servem como introdução a conceitos mais complexos em estrutura de dados e análise de algoritmos, sendo utilizados em áreas que envolvem grande processamento de dados, como pesquisa e aprendizado de máquina.

Neste trabalho vamos, entender, implementar e analisar diferentes algoritmos de ordenação com o foco em estudar suas variações e comportamentos em diferentes contextos. Faremos uma análise de complexidade, considerando o tempo de execução, abordando os cenários de melhor, pior e caso médio para cada algoritmo.

A partir dos dados coletados, será feita uma comparação da eficiência dos algoritmos de ordenação, usando como critério o tempo de execução em diferentes tipos de entradas e tamanho: crescente, decrescente e aleatória. Cada algoritmo terá seu tempo de execução calculado e organizados em tabelas. Com essas tabelas, será possível construir gráficos para visualizar melhor o comportamento de cada algoritmo sob diferentes condições.

Essas representações visuais nos permitirão identificar padrões de desempenho e estabelecer quais algoritmos são mais eficientes em cada cenário. Esse processo possibilitará uma visão prática e teórica sobre o impacto da escolha dos algoritmos de ordenação em aplicações reais, resultando em uma conclusão fundamentada sobre a adequação de cada algoritmo para diferentes tipos de problemas.

Sumário

2 INTRODUÇÃO

2.1 Algoritmos e estruturas de dados

Os algoritmos fazem parte do nosso dia-a-dia. Instruções para medicamentos, uma receita culinária são exemplos de algoritmos. De acordo com [2] um algoritmo é como uma sequência de ações executáveis para obter a solução de um problema. Segundo Dijkstra (1971) um **algoritmo** é uma descrição de um padrão de comportamento que é expresso em conjuntos finitos de ações. Ao executar a operação $a + b$ percebe o mesmo padrão de comportamento, mesmo que os valores sejam diferentes para a e b.

Estruturas de dados e algoritmos estão diretamente ligados. Não é possível estudar estruturas de dados sem considerar os algoritmos ligado a elas, logo como a escolha dos algoritmos dependem da representação da estrutura de dados. Para resolver problemas é necessário abstrair a realidade, através de uma definição de um conjunto de dados que representa o mundo real.

A eficiência de um algoritmo muitas vezes depende da estrutura de dados utilizada para armazenar e manipular informações. Por exemplo, para um problema precisa de acesso rápido aos dados, pode ser mais adequado utilizar uma tabela hash, enquanto que para um conjunto de dados que necessite de ordenação, uma árvore binária pode ser melhor.

A análise da complexidade de algoritmos é uma área fundamental dentro da ciência da computação. Essa análise permite avaliar o desempenho de um algoritmo, possibilitando a escolha da solução mais adequada para um problema. Compreender o tempo de execução e o uso de memória é crucial, especialmente em aplicações que lidam com grandes volumes de dados, onde até pequenas melhorias de desempenho podem resultar em economias significativas de tempo e recursos.

A inter-relação entre algoritmos e estruturas de dados não apenas influencia a eficiência das soluções, mas também a sua implementação prática em diversas linguagens de programação. A escolha de uma linguagem pode impactar a maneira como um algoritmo é desenvolvido. A compreensão de ambos os conceitos é essencial para qualquer profissional que deseje atuar na área, desenvolvendo soluções eficientes e eficazes.

3 ALGORITMOS

3.1 Insertion Sort

O primeiro algoritmo abordado será o algoritmo de ordenação por inserção para resolver problemas de ordenação.

Entrada: Uma sequência de n números $(a_1, a_2, \ldots, a_n)$.

Saída: Uma permutação (reordenação) $(a_1, a_2, \ldots, a_n)$ da sequência de entrada, tal que $a_1 \leq a_2 \leq \ldots \leq a_n$.

Os números que iremos ordenar são chamados de **chaves**.

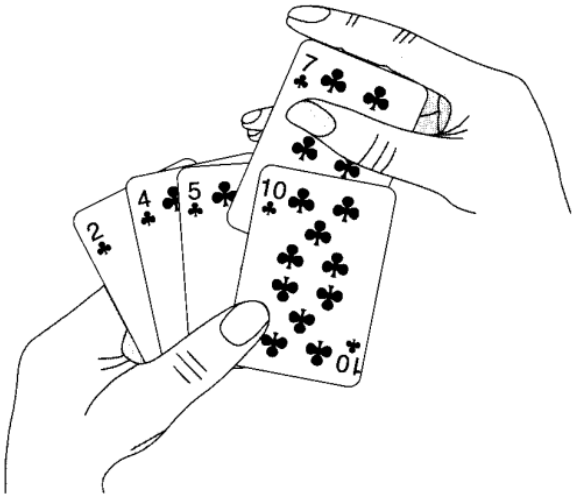

Figura 1: Ordenando cartas com o uso de ordenação por inserção do livro[1]

Este algoritmo é eficiente para ordenar um pequeno conjunto de elementos. Segundo o livro[1] sua ordenação funciona de maneira similar a ordenação de cartas de um jogo de pôquer. Começamos com a mão esquerda vazia com as cartas viradas com a face para baixo. Logo, removemos uma carta de cada vez, inserindo a mesma na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, comparamos esta com cada uma das cartas que já está na mão, da direita para a esquerda. Em cada instante, as cartas da mão esquerda são ordenadas. Tomamos como parâmetro um arranjo A[1...n] contendo uma sequência de comprimento n que deverá ser ordenada. Os números da entrada são **ordenados no local**, os números são reorganizados dentro de um arranjo A com no máximo um número

constante deles armazenado. O arranjo A conterá a sequência armazenada quando o insertion-sort terminar.

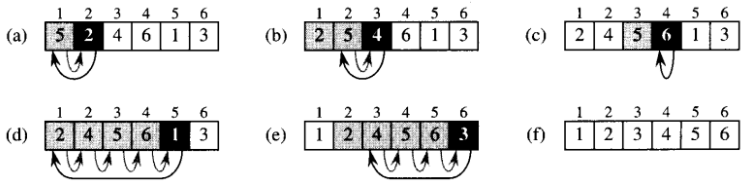

Figura 2: Operação de insertion sort sobre o arranjo A = [5,2,4,6,1,3] do livro[1].

Os índices do arranjo ficam em cima dos retângulos e seus valores armazenados dentro dos retângulos. Em cada iteração o retângulo preto contém a chave obtida de A[j] que é comparada com os valores contidos nos retângulos sombreados à sua esquerda. As setas sombreadas indicam valores do arranjo deslocados uma posição a direita e setas pretas para onde a chave é deslocada.

```
void insertion_Sort( int vetor, int tam ) {
    int i;
    int j;
    int chave;

    for( i = 1; i < tam; i++ ) {
        chave = vetor[i];
        j = i - 1;
        while( j >= 0 && vetor[j] > chave ) {
            vetor[j+1] = vetor[j];
            j = j - 1;
        }
        vetor[j+1] = chave;
    }
}
```

Figura 3: Insertion-sort implementado na linguagem C pelo autor

O loop **for** irá percorrer o arranjo, a chave recebe a posição [1], a variável j recebe o indice - 1 (indice da esquerda) o insertion sort irá comparar os elementos do arranjo da esquerda para direita. O loop **while** irá verificar se a variável j é maior ou igual a zero(quantidade de elementos a serem comparados) e verifica se o arranjo na posição j é maior do que a chave, se esta condição for verdadeira os elementos irão trocar de posição o j será decrementado, caso a condição seja falsa o a chave será inserida na posição j + 1 logo, ordenando o arranjo.

Teste de Mesa para o Insertion Sort

j	chave = A[j]	i = j - 1	A[i+j] = A[i]	i = i - 1	A[i+1] = chave	Sequência
2	chave = A[2]	i = 2 - 1	A[1+1] = A[1]	i = 1 - 1	A[0+1] = 2	5, 2, 4, 6, 7
2	chave = 2	i = 1	A[2] = 5	i = 0	A[1] = 2	2, 5, 4, 6, 7
4	chave = A[3]	i = 3 - 1	A[2+1] = A[2]	i = 2 - 1	A[1+1] = 4	2, 5, 4, 6, 7
4	chave = 4	i = 2	A[3] = 5	i = 1	A[2] = 4	2, 4, 5, 6, 7
6	chave = A[4]	i = 4 - 1	null	null	A[3+1] = 6	2, 4, 5, 6, 7
6	chave = 6	i = 3	null	null	A[4] = 6	2, 4, 5, 6, 7
7	chave = A[5]	i = 2 - 1	null	null	A[4+1] = 6	2, 4, 5, 6, 7
7	chave = 7	i = 1	null	null	A[5] = 6	2, 4, 5, 6, 7

Tabela 1: Teste de Mesa simplificado para o Insertion Sort com a sequência $5, 2, 4, 6, 7$ feito pelo autor (Este teste foi inspirado no teste apresentado em sala de aula).

4 ANÁLISE DE COMPLEXIDADE

4.1 Insertion Sort

4.1.1 Análise da ordenação por inserção

Analisar um algoritmo significa prever os recursos que um algoritmo irá utilizar. Sendo estes memória, largura de banda de comunicação ou de hardware são as principais preocupações mas o tempo de computação é o que iremos medir. Pela análise podemos decidir qual algoritmo é mais eficiente para determinado problema.

O **tempo de execução** do insertion-sort depende da entrada: a ordenação de mil números demora mais do que a ordenação de 10 números. Além disso, o insertion-sort pode demorar tempos diferentes para ordenar duas sequências de entrada do mesmo tamanho pois depende do quão ordenada a entrada está. No geral o tempo de execução de um algoritmo cresce com o **tamanho da entrada**.

O tempo de execução de um algoritmo de uma determinada entrada é o número de operações primitivas executadas. Um período constante de tempo é exigido para executar cada linha do código, onde estas linhas de código podem levar diferentes tempos para serem executadas, vamos considerar que cada execução da i-ésima linha leva um tempo c_i onde c_i é uma constante.

Assim apresentamos o insertion-sort com o custo de tempo de cada instrução e o número de vezes que cada instrução é executada. Seja t_j o número de vezes que o teste do loop while é executado para esse valor de j. Quando um loop **for** ou **while** termina, o teste é executado mais uma vez além do corpo do loop testando se a condição é verdadeira ou falsa.

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução, uma instrução que demanda c_i passos para ser executada e é executada n vezes, contribuirá com c_in para o tempo de execução total. Para calcular T(n), somamos os produtos das colunas custo e vezes obtendo.

$T(n) = c_i \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot \sum_{j=2}^{n} t_j + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot \sum_{j=2}^{n} (t_j - 1) + c_8 \cdot (n-1)$

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
2 **do** *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *chave*	c_8	$n-1$

Figura 4: Insertion sort com seus respectivos custos do livro[1]

4.1.2 Melhor caso

O melhor caso ocorre se o arranjo já se encontra ordenado. Para cada j = 2,3...,n, $A[i] \leq b$ quando i tem seu valor inicial j - 1. Portanto $t_j = 1$ para j = 2,3,...,n, e o tempo de execução do melhor caso é:

$T(n) = c_i \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot (n-1) + c_8 \cdot (n-1)$

$= (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)$

Esse tempo de execução pode ser expresso com $a \cdot n + b$ para constantes a e b que dependem dos custos de instrução c_i, assim ele é uma **função linear** de n.

4.1.3 Pior caso

Se o arranjo estiver ordenado em ordem inversa(ordem decrescente) resulta no pior caso. Devemos comparar cada elemento A[j] com cada elemento do subarranjo ordenado inteiro, A[1..j-1], e então $t_j = j$ para 2,3,...,n. Observando que:

$\sum_{j=2}^{n} j = \frac{n \cdot (n+1)}{2} - 1$

e

$\sum_{j=2}^{n} (j-1) = \frac{n \cdot (n-1)}{2}$

O tempo de execução do pior caso do insertion sort é:

$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c5 \cdot (\frac{n \cdot (n+1)}{2} - 1) + c_6 \cdot (\frac{n \cdot (n-1)}{2}) + c_7 \cdot (\frac{n \cdot (n-1)}{2}) + c_8 \cdot (n-1)$

$= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) \cdot n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)$

Esse tempo de execução no pior caso é expresso por $a \cdot n^2 + b \cdot n + c$ para constantes a, b e c que dependem dos custos de instrução c_i, logo é uma **função quadrática** de **n**.

4.1.4 Caso médio

O caso médio de um algoritmo é quase tão ruim quanto o pior caso. Suponha que seja escolhidos aleatoriamente n números e que se apliquem a eles a ordenação por inserção. Qual será o tempo para descobrir o lugar no subarranjo A[1..j-1] em que se deve inserir o elemento A[j]?Em média, metade dos elementos em A[1..j-1] são menores do que A[j], e metade dos elementos são maiores. Logo, verificamos metade do subarranjo A[1..j-1], e então $t_j = \frac{j}{2}$. Podemos observar que:

$\sum_{j=2}^{n}(\frac{j}{2}) = \frac{1}{2} \cdot \sum_{j=2}^{n} j = (\frac{1}{2}) \cdot (\frac{n \cdot (n+1)}{2} - 1) = \frac{n^2+n}{4} - \frac{1}{2}$

e

$\sum_{j=2}^{n}(\frac{j}{2} - 1) = (\frac{1}{2}) \cdot \sum_{j=2}^{n}(j-1) = (\frac{1}{2}) \cdot (\frac{n \cdot (n-1)}{2}) = \frac{n^2-n}{4}$

Logo o tempo de execução do médio caso é:

$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c5 \cdot (\frac{n \cdot (n+1)}{4} - \frac{1}{2}) + c_6 \cdot (\frac{n \cdot (n-1)}{4}) + c_7 \cdot (\frac{n \cdot (n-1)}{4}) + c_8 \cdot (n-1)$

$= (\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4}) \cdot n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{4} - \frac{c_6}{4} - \frac{c_7}{4} + c_8) \cdot n - (c_2 + c_4 + \frac{c5}{2} + c_8)$

Desenvolvemos o tempo de execução o tempo de execução do caso médio, e também será uma função quadrática assim como o pior caso.Um problema na análise do caso médio é que não pode ser aparente, o que é uma entrada "média"para um determinado problema.

5 TABELA E GRÁFICO

5.1 Insertion Sort

Entrada	Crescente	Decrescente	Randômica
10	0.0000	0.0000	0.0000
100	0.0000	0.0000	0.0000
1000	0.0000	0.0000	0.0000
10000	0.0000	0.2670	0.0480
100000	0.0000	21.4750	9.7210
1000000	0.0160	1839.450	796.112

Tabela 2: Tabela de tempo por segundo do algoritmo Insertion Sort feito pelo autor

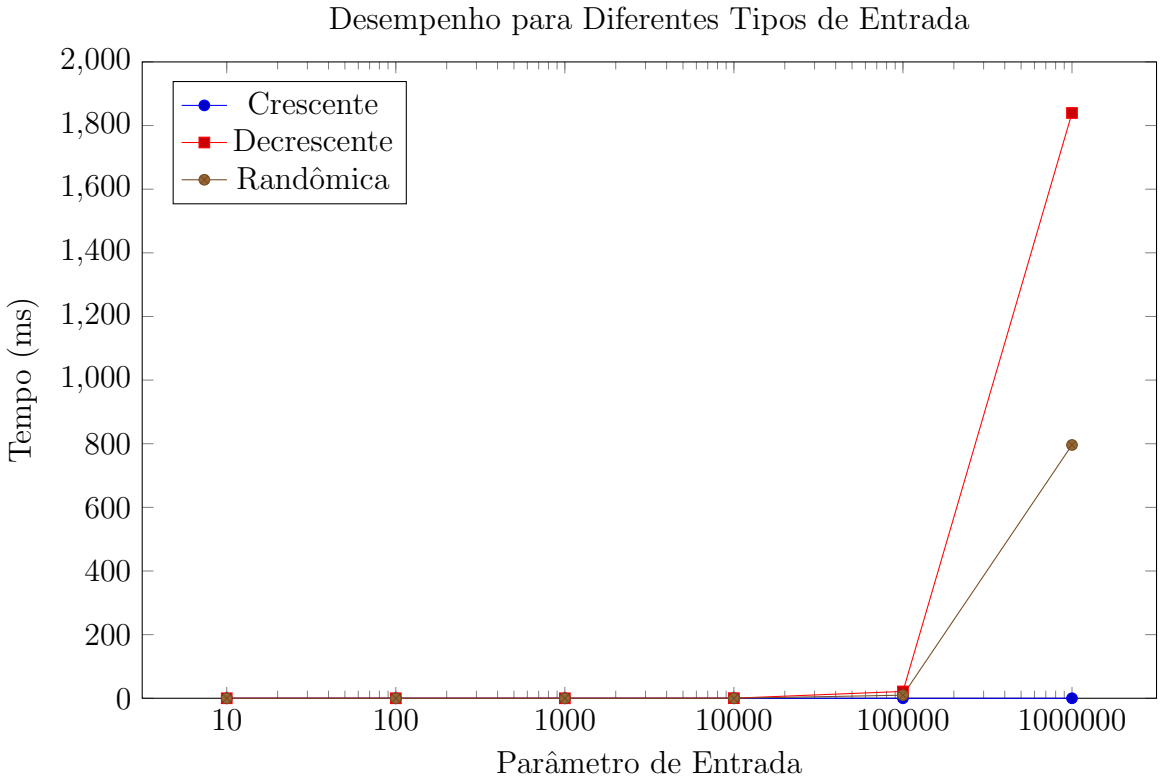

Figura 5: Gráfico de tempo por segundo do algoritmo Insertion Sort feito pelo autor

Com os dados coletados temos a conclusão que o insertion sort, trabalha bem conjuntos pequenos de dados, e que seu tempo de execução tende a subir conforme a entrada e conforme os casos a serem ordenados, crescente, decrescente e randômico. Nos piores casos é necessário fazer muitas comparações o que deixa o algoritmo ineficiente para grandes volumes de dados.

6 CONCLUSÃO

Em aberto...

...

7 REFERÊNCIAS BIBLIOGRÁFICAS

Referências

[1] Thomas H. Cormen et al. *Algoritmos: Teoria e Prática*. 2ª ed. Rio de Janeiro: Editora Campus, 2002, p. 296.

[2] Nivio Ziviani e et al. *Projeto de Algoritmos: com Implementações em Pascal e C*. Vol. 2. Luton: Thomson Luton, 2004.

8 Função para calcular o tempo

void operacoes(int tipo, int tamanho)

$clock_t start_t, end_t$; //Variavel para guardar o tempo

double tempoGasto;

int * vetor = gerarSequencia(tipo, tamanho); //Gera sequencia de numeros

//Salva a entrada de numeros

salvarEntrada(tipo, tamanho, vetor);

$start_t = clock()$; //Calcula o tempo atual

insertionSort(vetor, tamanho); //Ordena o Vetor

$end_t = clock()$; //Calcula o tempo apos ordenação

tempoGasto = $((end_t - start_t)$ / (double)$CLOCKS_PER_SEC)$; //Calcula diferença de tempo

salvarTempo(tipo, tamanho, tempoGasto); //Salva o tempo gasto

//Salva a saida do programa

salvarSaida(tipo, tamanho, vetor);

//Libera a memoria

free(vetor);