UNIVERSIDADE DO VALE DO ITAJAÍ

GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

ALGORITMOS DE ORDENAÇÃO

por

Artur Ramiro Furtado, Diogo Henrique Ribicki

Itajaí (SC), Dezembro de 2024

ALGORITMOS DE ORDENAÇÃO

Artur Ramiro Furtado, Diogo Henrique Ribicki

Dezembro / 2024

Professor: Marcos Carrard.

Linha de Pesquisa: Algoritmos de ordenação

Número de páginas: 18

RESUMO

Este trabalho apresenta a implementação e análise de diferentes algoritmos de ordenação em C++, abordando seus aspectos teóricos e práticos. Foram implementados sete algoritmos: *Insertion Sort, Bubble Sort, Shell Sort, Selection Sort, Heap Sort, Quick Sort e Merge Sort*. O programa permite ao usuário testar o desempenho desses algoritmos em diferentes cenários (melhor caso, pior caso e caso médio), configurando o tamanho do vetor, o número de repetições e a situação inicial dos dados.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste trabalho é implementar e analisar diferentes algoritmos de ordenação, explorando suas características em termos de eficiência temporal e espacial. O estudo envolve cenários variados para os dados de entrada, incluindo o melhor caso (vetor já ordenado), o pior caso (vetor em ordem inversa) e o caso aleatório (valores dispostos de maneira aleatória).

A partir da implementação em C++, foi desenvolvido um programa capaz de medir o tempo de execução de cada algoritmo em diferentes condições e tamanhos de entrada. Os dados foram coletados e utilizados para criar representações gráficas, permitindo uma visualização clara do desempenho de cada método. Além disso, o relatório também explora a lógica por trás de cada função implementada, discutindo suas implicações teóricas e práticas, como a complexidade computacional e o uso de memória.

O estudo apresenta *insights* sobre como diferentes estratégias de ordenação se comportam em cenários reais, destacando vantagens e limitações de cada abordagem. Os resultados obtidos serão úteis para fundamentar a escolha de algoritmos em contextos práticos e ilustrar a importância de considerar características dos dados de entrada ao selecionar uma estratégia de ordenação.

2 DESENVOLVIMENTO

Realizamos a implementação manual de algoritmos famosos como *Heap Sort, Bubble Sort, Direct Insert, Merge Sort, Shell Sort, Direct Selection*, todos eles foram criados usando C++, os gráficos referentes aos resultados foram realizados usando *Python* com a biblioteca *matplotlib*.

Os tempos de execução foram coletados e comparados para cada algoritmo, e a média dos tempos foi calculada com base no número de execuções solicitadas. A análise incluiu discussões sobre a eficiência de tempo e espaço de cada algoritmo, considerando sua complexidade teórica em notação *Big-O* e os resultados empíricos obtidos. Gráficos e tabelas foram utilizados para ilustrar as diferenças de desempenho em termos de tempo médio para cada cenário.

2.1 GERAR VETORES

A função gerarVetor é responsável por criar os vetores utilizados nos testes, ajustando-se aos cenários escolhidos pelo usuário.

Melhor caso: Cria um vetor ordenado em ordem crescente.
Pior caso: Cria um vetor ordenado em ordem decrescente.
Caso aleatório: Gera valores inteiros aleatórios no intervalo definido.

Complexidade:

Melhor caso e pior caso têm complexidade O(n) devido à criação sequencial dos elementos. Caso aleatório também apresenta O(n), mas o uso de geradores de números aleatórios pode influenciar a performance.

```
int* gerarVetor(int tamanho, int opcao) {
    int* vetor = new int[tamanho];
    if (opcao == 1) {
        for (int i = 0; i < tamanho; ++i) {
            vetor[i] = i + 1;
        }
    }
    else if (opcao == 2) {
        for (int i = 0; i < tamanho; ++i) {
            vetor[i] = tamanho - i;
        }
    }
    else {
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(1, tamanho);
        for (int i = 0; i < tamanho; ++i) {
            vetor[i] = dis(gen);
        }
    }
    return vetor;
}

```

2.2 *BUBBLE SORT*

Lógica: Repetidamente percorre o vetor, comparando pares de elementos adjacentes e trocando-os se estiverem fora de ordem. Esse processo se repete até que o vetor esteja ordenado.

Eficiência: Nos melhores casos irá apresentar O(n) já que o vetor já está ordenado e o algoritmo irá apenas passar pelos valores sem realizar nenhuma troca. Nos piores casos irá apresentar O(n^2)

O Bubble Sort está longe de ser o melhor algoritmo de ordenação é comumente usado apenas para lecionar a funcionalidade desses algoritmos, em todos iremos perceber que ele não se destaca.

```
void bubbleSort(int* vetor, int tamanho) {
    for (int i = 0; i < tamanho - 1; ++i) {
        for (int j = 0; j < tamanho - i - 1; ++j) {
            if (vetor[j] > vetor[j + 1]) {
                swap(vetor[j], vetor[j + 1]);
            }
        }
    }
}
```

2.3 *HEAP SORT*

Lógica: *Transforma o array no que chamamos de heap ou um amontoado máximo usando a função heapify() e, iterativamente, remove o maior elemento (raiz do heap) para colocá-lo na posição final do vetor.*

Eficiência: *Em todos os casos irá apresentar a complexidade de O(n log n)*

O Heap Sort demonstra ser um algoritmo equilibrado que apresenta valores similares em todos os casos desde o pior até o melhor. Ao que tudo indica é um algoritmo que lida bem com questões de memória.

```
void heapify(int* vetor, int tamanho, int i) {
    int maior = i;
    int esq = 2 * i + 1;
    int dir = 2 * i + 2;

    if (esq < tamanho && vetor[esq] > vetor[maior])
        maior = esq;

    if (dir < tamanho && vetor[dir] > vetor[maior])
        maior = dir;

    if (maior != i) {
        swap(vetor[i], vetor[maior]);
```

```
14          heapify(vetor, tamanho, maior);
15      }
16  }
17
18  void heapSort(int* vetor, int tamanho) {
19      for (int i = tamanho / 2 - 1; i >= 0; --i) {
20          heapify(vetor, tamanho, i);
21      }
22
23      for (int i = tamanho - 1; i > 0; --i) {
24          swap(vetor[0], vetor[i]);
25          heapify(vetor, i, 0);
26      }
27  }
```

2.4 INSERÇÃO DIRETA

Lógica: *Percorre o vetor a partir do segundo elemento, comparando-o com os elementos anteriores e deslocando-os para encontrar a posição correta do elemento atual. Assim, mantém a porção inicial do vetor ordenada conforme avança.* ***Eficiência:*** *No melhor caso apresenta complexidade O(n), visto que o vetor já está ordenado, o algortimo apenas realiza as comparações, sem necessidade de trocas.*

Porém no pior caso e no caso médio esse algoritmo apresenta uma complexidade de On^2*, quando o vetor está totalmente inverso, exige o número máximo de deslocamentos possível o qual depende do tamanho do vetor, enquanto para vetores aleatórios, o número de comparações e trocas será proporcional ao quadrado do tamanho do vetor.*

É um algoritmo que tem seu espaço em vetores pequenos ou quase ordenados, porém quando o temos um grande volume de dados ele se torna impraticálve devido à sua complexidade quadrática.

```
1  void insercaoDireta(int* vetor, int tamanho) {
2      for (int i = 1; i < tamanho; ++i) {
3          int chave = vetor[i];
4          int j = i - 1;
5          while (j >= 0 && vetor[j] > chave) {
6              vetor[j + 1] = vetor[j];
7              j--;
8          }
9          vetor[j + 1] = chave;
10     }
11 }
```

2.5 *MERGE SORT*

Lógica: *Reparte o vetor ao meio recursivamente, em diveresas pequenas parte até que cada uma tem um único valor, Em seguida, combina cada parte com sua subparte de forma ordenanda utilizando a função merge().*

Eficiência: *Assim como o Heap Sort ele não apresenta um grau de complexidade muito diferente para cada caso, normalmente se mantém equilibrado nos dando uma nível de abstração de*

O(n log n)

Infelizmente o Merge Sort não tende a ser o melhor caso quando memória é algo escasso, já que o mesmo utiliza de recursão.

```
void merge(int* vetor, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int* L = new int[n1];
    int* R = new int[n2];

    for (int i = 0; i < n1; ++i)
        L[i] = vetor[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = vetor[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vetor[k] = L[i];
            i++;
        }
        else {
            vetor[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        vetor[k] = L[i];
        i++;
        k++;
    }
```

```
31
32      while (j < n2) {
33          vetor[k] = R[j];
34          j++;
35          k++;
36      }
37
38      delete[] L;
39      delete[] R;
40  }
```

```
1
2   void mergeSort(int* vetor, int l, int r) {
3       if (l < r) {
4           int m = l + (r - l) / 2;
5           mergeSort(vetor, l, m);
6           mergeSort(vetor, m + 1, r);
7           merge(vetor, l, m, r);
8       }
9   }
```

2.6 *QUICK SORT*

Lógica: *O Quick Sort foi um caso à parte em nosso relatório onde tivemos uma dificuldade com as decisões de pivô dele, começamos deixando o pivô como o último valor do array e assim percebemos que nosso algoritmo era o pior, conseguimos fazer ele ser pior que o próprio Bubble Sort chegando a precisa de 100ms para ordenar um vetor no melhor caso, o que achamos esquisito.*

Segunda proposta foi implementar a divisão recursiva que parecia fazer sentido, deixamos uma função separada para fazer o parcionamento do vetor de modo que o array fosse divido e o pivot navegasse por ele, da mesma forma não tivemos sucesso.

Última e acertiva proposta foi a mediana de três, a qual deixamos em nosso relatório, notamos que a implementação da mediana de três fazia mais sentido visto que ela posicionava o pivot de forma mais concisa, como o nosso pivot era sempre o último elemento, quando o vetor estava ordenado, o Quick Sort caia em seu pior caso, pois o mesmo desbalanceava o vetor já ordenado criando partições completamente desbalanceadas. Erro que a mediana de três resolveu ao posicionat o pivot de forma mais representativa em relação aos elementos.

Eficiência: *Nos melhores e médios casos temos uma complexidade de nível O(n log n) em seu pior caso onde todas as partições estão desbalanceadas temos O(n^2)*

```
int medianOfThree(int* vetor, int low, int high) {
    int mid = low + (high - low) / 2;
    if (vetor[low] > vetor[mid]) swap(vetor[low], vetor[mid]);
    if (vetor[low] > vetor[high]) swap(vetor[low], vetor[high]);
    if (vetor[mid] > vetor[high]) swap(vetor[mid], vetor[high]);
    return mid;
}


void quickSort(int* vetor, int low, int high) {
    int pivotIndex = medianOfThree(vetor, low, high);
swap(vetor[pivotIndex], vetor[high]);
int pivot = vetor[high];

}
```

2.7 SELEÇÃO DIRETA

Lógica: *Itera sobre o vetor passando por cada elemento até encontrar o menor elemento não ordenado e o coloca na posição correta. É um algoritmo simples, porém em larga escala não é eficiente já que terá que iterar sobre vetores enormes, levando muito mais tempo do que o necessário.*
Eficiência: *Nos 3 casos apresenta* On^2

```
void selecaoDireta(int* vetor, int tamanho) {
    for (int i = 0; i < tamanho - 1; ++i) {
        int min_idx = i;
        for (int j = i + 1; j < tamanho; ++j) {
            if (vetor[j] < vetor[min_idx]) {
                min_idx = j;
            }
        }
        swap(vetor[min_idx], vetor[i]);
    }
}
```

2.8 *SHELL SORT*

Lógica: *Ordena elementos que estão distantes uns dos outros, reduzindo gradativamente essea distância até se tornar equivalente ao Bubble para elementos que estão ao lado.*

Eficiência: *Depende do número de sêquencias de passos utilizadas para aproximar os ponteiros, normalmente os melhores casos são O(n log n) e o pior caso On^2.*

```
void shellSort(int* vetor, int tamanho) {
    for (int gap = tamanho / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < tamanho; ++i) {
            int temp = vetor[i];
            int j;
            for (j = i; j >= gap && vetor[j - gap] > temp; j -= gap) {
                vetor[j] = vetor[j - gap];
            }
            vetor[j] = temp;
        }
    }
}
```

3 RESULTADOS E DISCUSSÃO

Os resultados apresentados a seguir foram obtidos a partir da ordenação de dois conjuntos de dados: um vetor de 5000 itens, onde os tempos médios foram calculados com base em 20 repetições de cada algoritmo, e um vetor de 10000 itens, com tempos médios calculados a partir de 100 repetições. Cada barra do gráfico corresponde a uma situação específica: o pior caso (representado pela cor azul), o melhor caso (cor laranja) e o caso médio ou caso aleatório (cor verde).

É importante destacar que, ao comparar os valores numéricos coletados com a visualização gráfica, observamos que alguns algoritmos, como o Quick Sort, Heap Sort e Merge Sort, possuem tempos de execução extremamente baixos. Para permitir a visualização dessas diferenças no gráfico, foi utilizada uma escala logarítmica. Essa escolha é fundamental para evitar que algoritmos muito rápidos fiquem com barras praticamente invisíveis, ao passo que algoritmos mais lentos, como o Bubble Sort, dominariam a escala em uma representação linear.

A análise gráfica reforça as diferenças práticas de desempenho entre os algoritmos. O Bubble Sort e a Seleção Direta, com complexidade temporal O(n2) e O(n2), apresentam tempos de execução muito elevados à medida que o tamanho do vetor cresce, especialmente no pior caso. Em contraste, algoritmos mais avançados, como o Merge Sort e o Heap Sort O(nlogn) O(nlogn), mantêm desempenho consistente e significativamente superior em todos os cenários.

Uma observação especial deve ser feita para o Quick Sort, que, com a otimização da mediana de três, apresenta tempos de execução excepcionalmente baixos, mesmo no pior caso. Esse comportamento reflete a eficiência do algoritmo em lidar com cenários adversos, desde que implementado de forma a minimizar os riscos de degeneração para complexidade O(n2)O(n2).

Tabela 1. Tempos de execução dos algoritmos (em milissegundos)

Tamanho do Vetor	Algoritmo	Pior Caso	Melhor Caso	Caso Médio
5000	Bubble Sort	117.418	28	95
	Heap Sort	1.08662	1	1
	Inserção Direta	32.5269	0.01	17
	Merge Sort	0.456066	0.4	0.7
	Quick Sort	1.3	0.8	1.2
	Seleção Direta	29.6068	31	32
	Shell Sort	0.272353	0.2	0.8
10000	Bubble Sort	783	123	415
	Heap Sort	10	2.7	11
	Inserção Direta	132	0.03	64
	Merge Sort	4	1.1	7
	Quick Sort	1.0	0.8	1.1
	Seleção Direta	488	133	454
	Shell Sort	2	0.4	1.9

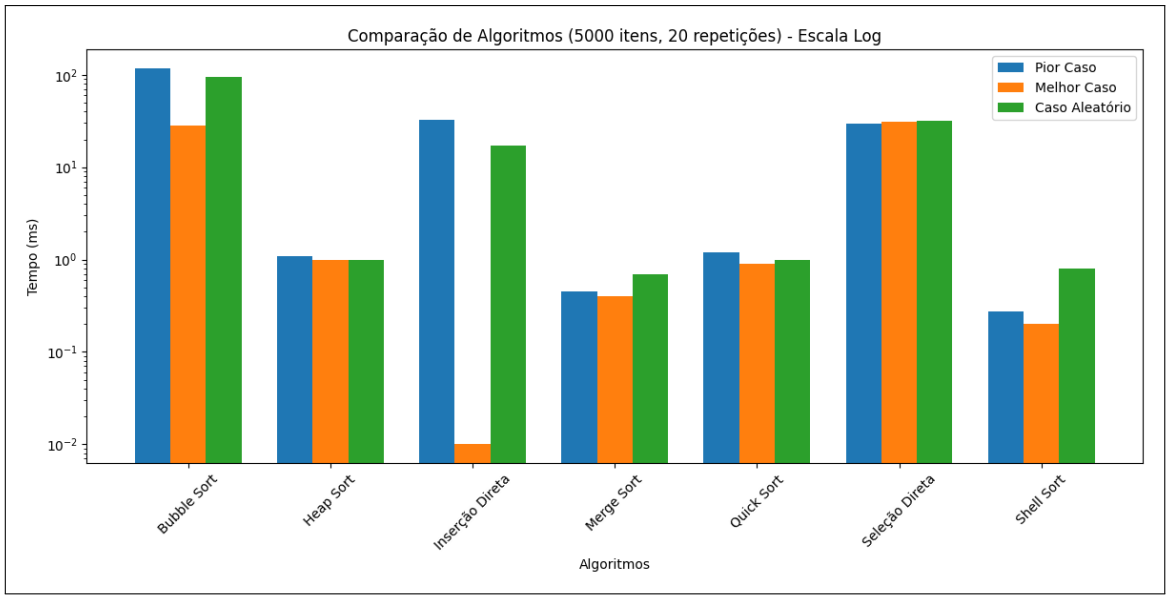

Figura 1. Comparação de tempo entre os algoritmos ordenando um vetor de 5000 itens, calculando a média de 20 ordenações.

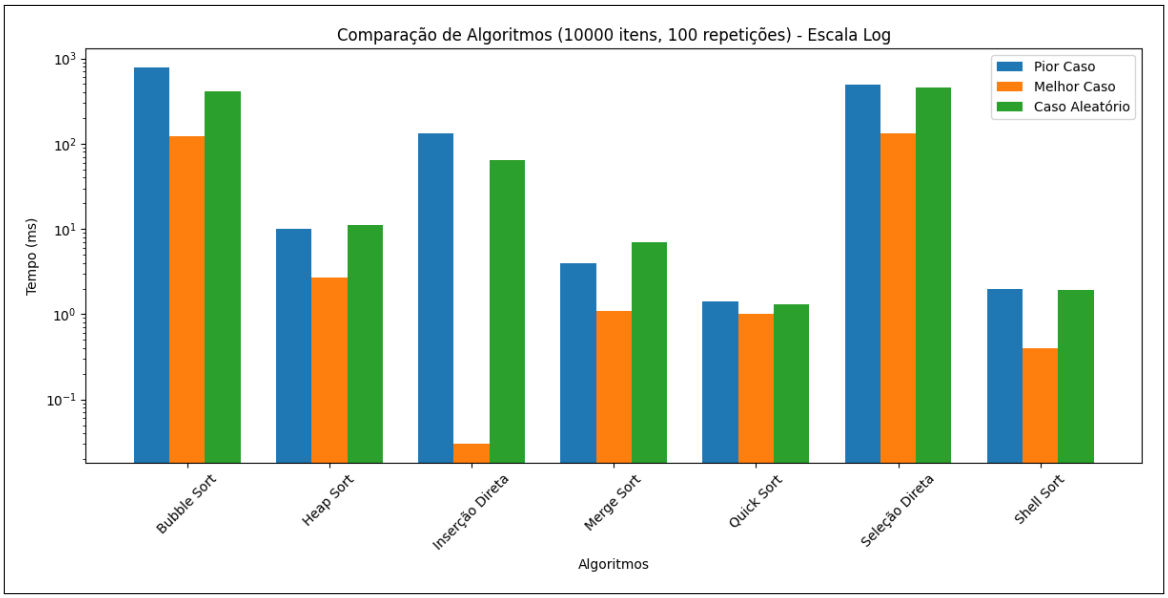

Figura 2. Comparação de tempo entre os algoritmos ordenando um vetor de 10000 itens, calculando a média de 100 ordenações.

Por fim, os resultados confirmam que a escolha do algoritmo de ordenação é um fator crítico e depende não apenas do tamanho dos dados, mas também de sua distribuição inicial. A análise detalhada de cada caso fornece uma visão prática sobre a aplicação eficiente de cada técnica, conforme as características do problema.

Bubble Sort: Extremamente ineficiente, com desempenho $O(n^2)$ em todos os cenários. Sua simplicidade de implementação é seu único ponto positivo. Heap Sort: Consistente com $O(n \log n)$ em qualquer caso, mas pode exigir memória adicional devido ao uso da estrutura de heap. Inserção Direta: Eficiente para dados pequenos ou parcialmente ordenados ($O(n)$ no melhor caso), mas lento em dados desordenados ($O(n^2)$). Merge Sort: Sempre eficiente ($O(n \log n)$), mas exige memória extra, o que pode ser uma limitação. Quick Sort: Muito rápido na prática, especialmente com otimizações como a mediana de três, embora o pior caso seja $O(n^2)$. Seleção Direta: Simples, mas com desempenho $O(n^2)$ que o torna pouco prático para grandes conjuntos de dados. Shell Sort: Mais eficiente que outros algoritmos quadráticos, mas com desempenho dependente da sequência de espaçamento utilizada.

Essa análise destaca a importância de alinhar as características do problema com o algoritmo mais adequado, equilibrando desempenho e requisitos computacionais.