Uma Avaliação Empírica da Eficiência de Algoritmos de Ordenação sob Diferentes Cenários de Entrada

Bia Gabriela[1], Illan spala[2]

[1]Departamento de Computação – Universidade Federal do Espírito Santo (UFES)
Alegre – ES – Brasil

[2]Department of Computer Science – University of Durham
Durham, U.K.

[3]Departamento de Sistemas e Computação
Universidade Regional de Blumenal (FURB) – Blumenau, SC – Brazil

dayan.bissoli@ufes.br, R.Bordini@durham.ac.uk

Abstract. *This paper aims to implement and compare classical internal sorting algorithms using the C programming language. Thirteen algorithms were tested with input sequences of varying order (random, ascending, and descending) and sizes (9,999; 99,999; and 999,999 elements). The metrics analyzed include the number of comparisons, number of swaps, and execution time. Results are presented using tables and graphs, supporting a detailed performance analysis of each algorithm.*

Resumo. *Este trabalho tem como objetivo implementar e comparar algoritmos clássicos de ordenação interna em linguagem C. Foram testados treze algoritmos sob diferentes condições de entrada (aleatória, crescente e decrescente) e tamanhos (9.999, 99.999 e 999.999 elementos). As métricas analisadas foram: número de comparações, número de trocas e tempo de execução. Os resultados obtidos foram apresentados por meio de gráficos e tabelas, permitindo uma análise detalhada da eficiência e comportamento de cada algoritmo.*

1. informações gerais

Este trabalho foi desenvolvido no contexto da disciplina Estrutura de Dados II (COM10078), ofertada pelo Departamento de Computação da Universidade Federal do Espírito Santo (UFES), com o objetivo de aprofundar o estudo e a aplicação de algoritmos de ordenação em memória primária. Foram implementados e analisados treze algoritmos clássicos de ordenação interna, com foco na avaliação do seu desempenho prático. A ordenação em memória primária é um processo fundamental para a organização eficiente de dados em estruturas computacionais. Os algoritmos estudados neste trabalho incluem abordagens baseadas tanto em comparações (como Quicksort, Mergesort, Heapsort, entre outros), quanto em distribuição (como Bucketsort e Radixsort), permitindo uma visão ampla e comparativa das principais técnicas disponíveis. As implementações foram desenvolvidas na linguagem C, e os testes experimentais seguiram uma metodologia padronizada, com entradas geradas automaticamente em três tamanhos distintos: 9.999, 99.999 e 999.999 elementos. Para cada tamanho, foram utilizados três tipos de ordenação inicial: aleatória, crescente e decrescente. As métricas utilizadas para análise de desempenho dos algoritmos foram: Número de comparações realizadas; Número de trocas executadas Tempo de execução total em milissegundos. Os resultados são apresentados por meio

de gráficos e tabelas, permitindo uma análise comparativa clara do comportamento de cada algoritmo nas diferentes situações propostas. Este estudo visa não apenas avaliar a eficiência dos métodos, mas também fornecer embasamento teórico-prático para a escolha adequada de algoritmos em contextos reais de aplicação.

2. Fundamentos Teóricos

A ordenação de dados é uma das operações mais fundamentais da ciência da computação, estando presente em diversas aplicações que vão desde bancos de dados até algoritmos de inteligência artificial. Algoritmos de ordenação interna, que operam inteiramente em memória primária, são particularmente relevantes pela sua eficiência e aplicabilidade em contextos com conjuntos de dados que cabem integralmente na RAM do sistema.

Segundo Cormen et al. (2009), os algoritmos de ordenação podem ser classificados em duas categorias principais: algoritmos baseados em comparação e algoritmos baseados em distribuição. Os primeiros comparam pares de elementos para definir sua posição relativa (ex.: Quicksort, Mergesort, Heapsort), enquanto os segundos distribuem os dados em categorias para ordená-los sem comparações diretas (ex.: Bucketsort, Radixsort).

A complexidade assintótica de tempo é uma medida teórica importante para avaliação de algoritmos. O teorema do limite inferior para algoritmos de comparação estabelece que nenhuma estratégia baseada exclusivamente em comparações pode ordenar n elementos em menos de (log)O(nlogn) operações no pior caso (Cormen et al., 2009). No entanto, conforme aponta Knuth (1998), o comportamento prático desses algoritmos pode variar significativamente, especialmente em função da implementação, da arquitetura do sistema e da natureza dos dados de entrada.

Além disso, McGeoch (2012) enfatiza que avaliações experimentais são essenciais para compreender a eficiência real dos algoritmos, pois permitem observar fatores como tempo de execução, número de comparações e trocas, além de questões não previstas na análise teórica, como o impacto da hierarquia de memória e do cache do processador.

Estudos experimentais, como o realizado por Nguyen e Do (2019), demonstram que algoritmos como Quicksort, apesar de apresentarem um pior caso de (2)O(n2), costumam ter excelente desempenho médio, superando Mergesort em diversos cenários práticos. Por outro lado, algoritmos como Radixsort, que operam em tempo linear teórico sob certas condições, dependem fortemente do tipo de dado e da base de representação escolhida, o que pode afetar negativamente sua eficiência prática.

Dessa forma, a seleção adequada de um algoritmo de ordenação depende não apenas de sua complexidade teórica, mas também do contexto da aplicação, do volume de dados, do estado inicial da entrada e da estrutura de hardware subjacente. O presente trabalho visa justamente explorar esses aspectos, fornecendo uma análise comparativa entre treze algoritmos clássicos, implementados na linguagem C, com foco em desempenho prático sob diferentes cenários.

3. Algoritmos Estudados

Foram abordados treze algoritmos, divididos em dois grupos: métodos baseados em comparação e métodos baseados em distribuição.

3.1 Bolha e Bolha com critério de parada

O método BubbleSort (Bolha), neste algoritmo cada elemento da posição i será comparado com o elemento da posição i+1, ou seja, um elemento da posição dois será comparado com o elemento da terceira posição três, caso o elemento da posição dois for maior

que o da terceira, fazendo as trocas sucessivamente até que o vetor esteja completamente ordenado (OLIVEIRA, 2002). O método bolha é um dos algoritmos mais simples de ordenação, a ideia dele é fazer várias passagens pelo vetor, deixando o número maior no topo, fazendo isso até deixar o vetor completamente ordenado. De acordo com Honorato (2016), o BublleSort é o mais simples e menos eficiente, nele o elemento da posição i será comparado com o da posição i+1, se o elemento 2 for maior, eles trocam de posição, e assim sucessivamente. Para Honorato (2016) o método bolha é o mais ineficiente, uma vez que ele leva mais tempo e comparações devido a suas múltiplas passagens pelo vetor.

Figura 1. Exemplo da representação do método Bolha: os valores são trocados conforme a verificação do campo anterior e do próximo.

```
Função do Código de algoritmo bolha:
Metricas bubbleSort(int vetor[], int tamanho) {
    Metricas metricas = {0,0};
    int i, j, temp;
    for(i = 0; i < tamanho -1; i++) {
        for(j = 0; j < tamanho - 1 - i; j++) {
            metricas.comparacoes++;
            if(vetor[j] > vetor[j+1]) {
                temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
                metricas.trocas++;
            }
        }
    }
    return metricas;
}
```

```
Função do Código de algoritmo bolha
com critério de parada:
Metricas bubbleSorteComCriterioDeParada
(int vetor[], int tamanho) {
    Metricas metricas = {0,0};
    int i, j, temp, trocou;
    for(i = 0; i < tamanho - 1; i++) {
        trocou = 0;
        for(j = 0; j < tamanho - 1 - i; j++) {
            metricas.comparacoes++;
            if(vetor[j] > vetor[j+1]) {
                temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
                trocou = 1;
                metricas.trocas++;
            }
        }
        if (trocou == 0) return metricas;
    }
    return metricas;
}
```

3.2 Inserção Direta e Inserção Binária:

Método InsertionSort (Inserção Direta): neste algoritmo é percorrida uma lista sempre da esquerda para a direita, comparando as duas primeiras posições do vetor, fazendo com que os dados sejam ordenados mais à esquerda, e assim por diante até que o vetor esteja completamente ordenado (ZIVIANE, 2007). O método de inserção direta, de acordo com Honorato (2016) é o mais eficiente para listas pequenas, sua ideia é percorrer o vetor da esquerda para a direta, e a medida que ele avança, ele vai deixando os elementos mais à esquerda ordenados. Para Honorato (2016), pode-se comparar o método de inserção direta com uma mão de cartas, onde a cada carta recebida, é colocada na posição correta de acordo com o seu valor, esta é a ideia por trás da ordenação por inserção onde ele percorre as posições do array, começando com o índice 1, e a cada nova posição é como a nova carta que recebeu, e precisa inseri-la no lugar correto. A Figura 1 exemplifica o método de inserção direta, em que ao receber a carta 7, ela já é inserida na posição correta.

Figura 2. Exemplo da representação de comportamento de inserção direta.

```
Parte do Código de inserção direta:
Metricas insertionSort(int vetor[], int tamanho) {
    Metricas metricas = {0,0};
    int i, j, temp;
    for(i = 1; i < tamanho; i++) {
        temp = vetor[i];
        j = i - 1;
        while(j >= 0 && (metricas.comparacoes++,
        vetor[j] > temp)) {
            vetor[j+1] = vetor[j];
            metricas.trocas++;
            j--;
        }
        vetor[j + 1] = temp;
        if(i != j + 1) {
            metricas.trocas++;
        }
    }
    return metricas;
}
```

```
Parte do Código de inserção direta binária:
Metricas insertionSortBinario
(int vetor[], int tamanho) {
    Metricas metricas = {0,0};
    int i, j, temp, left, right, mid;
    for(i = 1; i < tamanho; i++) {
        temp = vetor[i];
        left = 0;
        right = i - 1;
        while(left <= right) {
            metricas.comparacoes++;
            mid = left + (right - left) / 2;
            if(temp < vetor[mid])
                right = mid - 1;
            else
                left = mid + 1;
        }
        for(j = i - 1; j >= left; j--) {
            vetor[j + 1] = vetor[j];
            metricas.trocas++;
        }
        vetor[left] = temp;
        if (i != left) {
            metricas.trocas++;
        }
    }
    return metricas;
}
```

3.3 Shellsort:

O método ShellSort pode ser considerado o refinamento do método InsertionSort, diferindo pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores (OLIVEIRA, 2002). O ShellSort permite trocas de registros que estão distantes um do outro. Os itens que estão separados n posições são rearranjados de forma que todo n-éssimo item leva a uma sequência ordenada. A Figura 3 ilustra a representação gráfica do método. aperfeiçoa a ordenação por inserção permitindo trocas distantes, apresentando desempenho entre $\mathcal{O}\left(n^{3/2}\right)$ e $\mathcal{O}\left(n \log^2 n\right)$, dependendo da sequência de incrementos adotada.

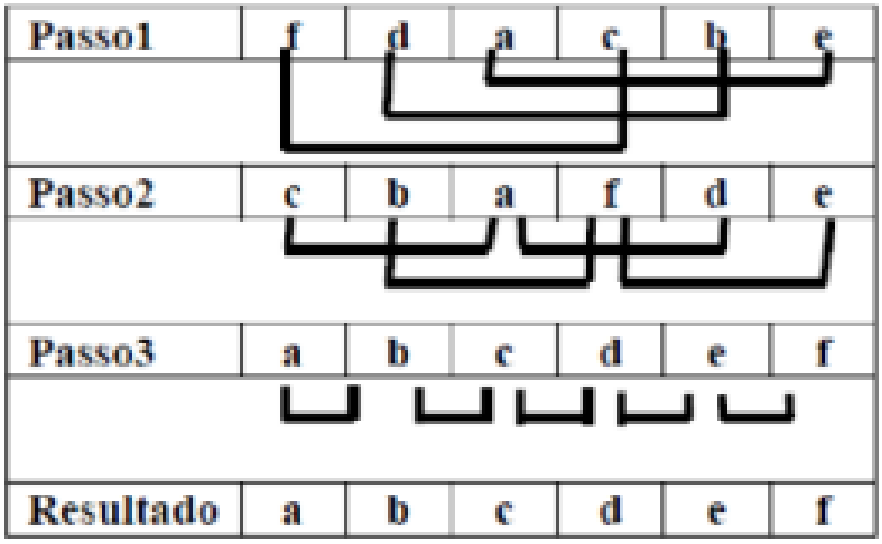

Figura 3. Exemplo da representação de comportamento de inserção direta.

```
Parte do Código de shellsort:
Metricas shellSort(int vetor[], int tamanho) {
    Metricas metricas = {0, 0};
    int gap, i, j, temp;
    for(gap = tamanho / 2; gap > 0; gap /= 2) {
        for(i = gap; i < tamanho; i++) {
            temp = vetor[i];
            j = i;
            while(j >= gap && (metricas.
            comparacoes++,
            vetor[j - gap] > temp)) {
                vetor[j] = vetor[j - gap];
                metricas.trocas++;
                j -= gap;
            }
            vetor[j] = temp;
            if(i != j) {
                metricas.trocas++;
            }
        }
    }
    return metricas;
}
```

3.4 Seleção Direta:

O método da Inserção direta é considerado um dos métodos mais simples, onde primeiramente, são ordenados os 2 primeiros membros de um vetor. Após, é inserido o 3º elemento na sua posição ordenada com relação aos 2 primeiros. O processo irá continuar

até que todos os elementos do vetor estejam devidamente ordenados (ZIVIANI, 2007). A classificação por seleção é aquela na qual sucessivos elementos são selecionados em sequência e dispostos em suas posições corretas pela ordem. Esses elementos da entrada precisem ser pré-processados acontecendo a seleção ordenada. método simples, porém pouco eficiente; realiza $\mathcal{O}(n^2)$ comparações e poucas trocas.

```
Parte do Código de seleção direta:
Metricas selectionSort(int vetor[], int tamanho){
    Metricas metricas = {0, 0};
    int i, j, minIndex, temp;

    for(i = 0; i < tamanho - 1; i++){
        minIndex = i; // considera o índice atual
        como o menor
        for(j = i + 1; j < tamanho; j++){
            metricas.comparacoes++;
            if(vetor[j] < vetor[minIndex]){
                minIndex = j; // atualiza o
                índice do menor elemento
            }
        }
        if(minIndex != i){ // se o menor elemento
        não está na posição correta
            temp = vetor[i];
            vetor[i] = vetor[minIndex];
            vetor[minIndex] = temp;
            metricas.trocas++;
        }
    }
    return metricas;
}
```

3.5 Heapsort:

O Heapsort, é baseado em uma estrutura de dados denominada heap. Um heap é uma árvore estritamente binária que satisfaz a seguinte propriedade: dado qualquer elemento da árvore, este será sempre maior ou igual que seus filhos diretos. Considerando a transitividade, cada elemento é maior ou igual que todos os seus descendentes na árvore. A árvore é, na verdade, implícita, simulada em um vetor, fazendo com que cada elemento no vetor com índice i seja pai dos elementos com índice (i*2)+1 e (i*2)+2 (considerando o endereço inicial do array igual a zero). A ordenação de um vetor com n elementos, pelo Heapsort é feita em dois passos: a) transformação do vetor em um heap. Essa transformação visa atender à propriedade de heap, e é feita através de um procedimento denominado DesceHeap, ilustrado na Figura 4, para as letras do string 'ANDRE'. Ao final deste procedimento, a chave mais alta do vetor estará na posição inicial do mesmo. b) execução de n-1 passos, onde se troca o primeiro elemento do vetor com o último, e executa-se a função DesceHeap para o primeiro elemento do vetor. Esse processo é ilustrado na Figura 5.

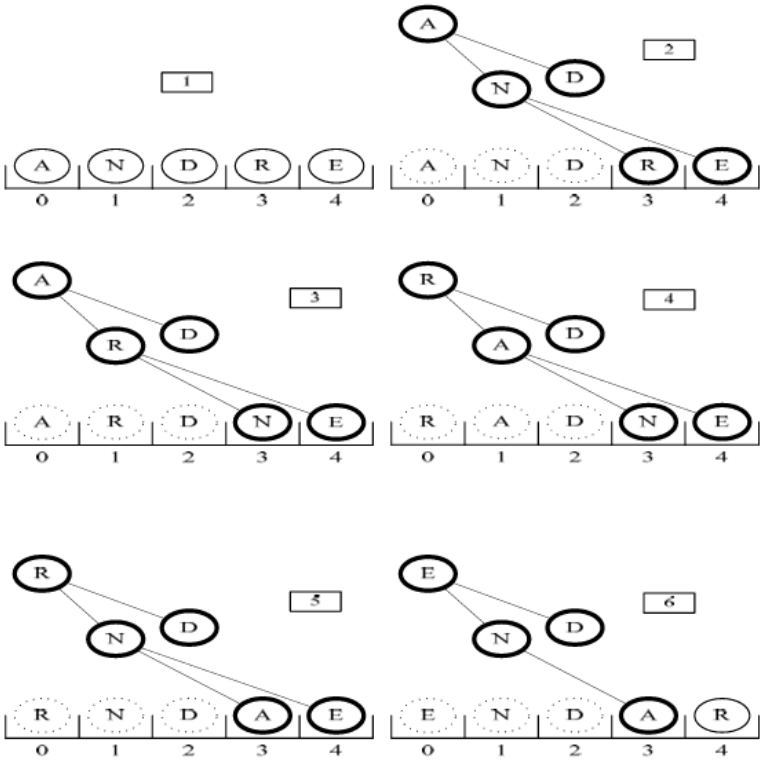

Figura 4. Heapsort em ação (1/2)

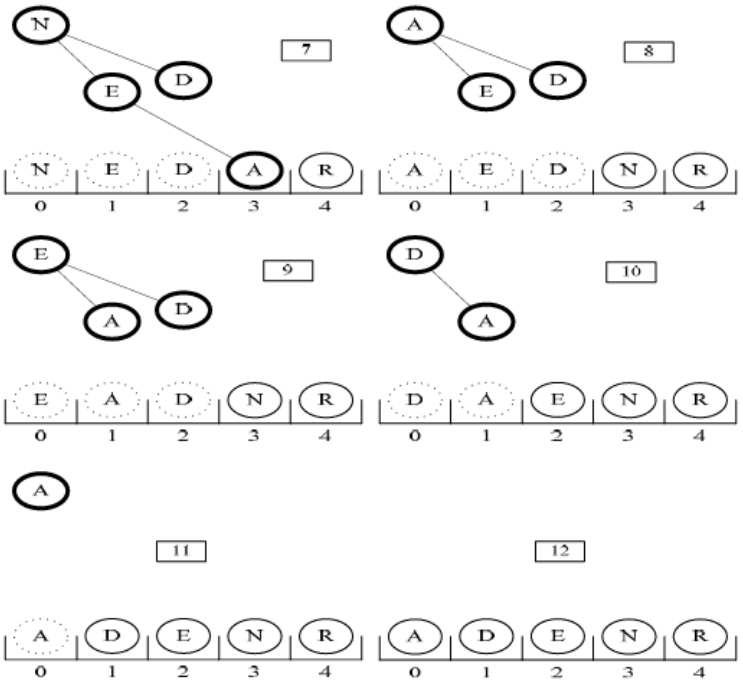

Figura 5. Heapsort em ação (2/2)

```
Parte do Código heapsort:
Metricas heapSort(int vetor[], int tamanho){
    Metricas metricas = {0, 0};
    int i = 0;
    // inicializa as metricas // a heap
    if (tamanho <= 0) return metricas;
    // Constrói a heap (max heap)

    for(i = (tamanho - 1) / 2; i >= 0; i--){
        criaHeap(vetor, i, tamanho, &metricas);
    }

    for(i = tamanho - 1; i > 0; i--){
        trocar(&vetor[0], &vetor[i], &metricas);
        // Reconstroi a heap reduzida
        criaHeap(vetor, 0, i, &metricas);

    }
    return metricas; // retorna as métricas
}
```

3.6 Mergesort:

Este algoritmo tem como objetivo a reordenação de uma estrutura linear por meio da quebra, intercalação e união dos nelementos existentes. Em outras palavras, a estrutura a ser reordenada será, de forma recursiva, subdividida em estruturas menores até que não seja mais possível fazê-lo. Em seguida, os elementos serão organizados de modo que cada subestrutura ficará ordenada. Feito isso, as subestruturas menores (agora ordenadas) serão unidas, sendo seus elementos ordenados por meio de intercalação(SZWARCFITER e MARKENZON, 2015). O mesmo processo repete-se até que todos os elementos estejam unidos em uma única estrutura organizada. algoritmo estável, com desempenho garantido de $\mathcal{O}(n \log n)$, à custa de maior uso de memória auxiliar.

Função recursiva do Código mergesort:

```
static Metricas mergeSortRecursivo(int vetor[],
int inicio, int fim){
    Metricas metricas_total = {0, 0};
    if(inicio < fim){
        int meio = inicio + (fim - inicio) / 2;

        Metricas metricas_esquerda =
        mergeSortRecursivo(vetor, inicio, meio);
        Metricas metricas_direita =
        mergeSortRecursivo(vetor, meio + 1, fim);
        Metricas metricas_intercalacao =
        merge(vetor, inicio, meio, fim);

        metricas_total.comparacoes =
        metricas_esquerda.comparacoes +
        metricas_direita.comparacoes +
        metricas_intercalacao.comparacoes;
        metricas_total.trocas =
        metricas_esquerda.trocas +
        metricas_direita.trocas +
        metricas_intercalacao.trocas;
    }
    return metricas_total;
}
```

Função auxiliar do Código mergesort:

```
static Metricas merge(int vetor[], int inicio,
int meio, int fim){
    Metricas metricas = {0, 0};
    int i, j, k;
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;

    int* L = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++){
        L[i] = vetor[inicio + i];
        metricas.trocas++;
    }
    for(j = 0; j < n2; j++){
        R[j] = vetor[meio + 1 + j];
        metricas.trocas++;
    }

    i = 0;
    j = 0;
    k = inicio;

    while(i < n1 && j < n2){
        metricas.comparacoes++;
        if(L[i] <= R[j]){
            vetor[k] = L[i];
            i++;
        } else {
            vetor[k] = R[j];
            j++;
        }
        metricas.trocas++;
        k++;
    }

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

    while(j < n2){
        vetor[k++] = R[j++];
        metricas.trocas++;
    }
    free(L);
    free(R);
    return metricas;
}
```

3.7 Quicksort:

O método de ordenação quicksort é considerado um dos métodos mais rápidos. Consiste em selecionar primeiramente um dos elementos do conjunto a ordenar para ser o elemento pivô, o qual é um elemento já ordenado, em seguida faz uma subdivisão do conjunto inicial em dois subconjuntos, onde o primeiro subconjunto irá conter todos os elementos menor que o elemento pivô, e o segundo subconjunto todos os elementos maiores que o pivô (ZIVIANI, 2007). Esse método utiliza a seleção em árvore para a obtenção dos elementos do vetor na ordem desejada. Consiste em duas fases, onde a primeira monta uma árvore binária (heap) contendo todos os elementos do vetor, de tal forma que o valor contido em qualquer nó seja maior do que os valores de seus filhos. E a segunda utiliza o heap para a seleção dos elementos na ordem desejada (CORMEN, 2002). amplamente utilizado devido ao ótimo desempenho médio ($\mathcal{O}(n \log n)$). Neste trabalho foram avaliadas três estratégias de pivô: Quicksort Centro; Quicksort Fim; Quicksort Mediana de Três.

Código Quicksort Centro:

```
Metricas quickSortCentro(int vetor[], int tamanho)
{
    Metricas metricas = {0, 0};
    quickSortRecursivo(vetor, 0, tamanho - 1,
    &metricas, 0); // Pivô Central
    return metricas;
}
```

Código Quicksort Mediana de Três:

```
static int particionar(int vetor[], int inicio,
int fim, Metricas* metricas) {
    int pivo = vetor[fim];
    int i = inicio - 1;
    for (int j = inicio; j <= fim - 1; j++) {
        metricas->comparacoes++;
        if (vetor[j] <= pivo) {
            i++;
            trocar(&vetor[i], &vetor[j], metricas);
        }
    }
    trocar(&vetor[i + 1], &vetor[fim], metricas);
    return i + 1;
}
```

```
Código Quicksort Fim:
static void quickSortRecursivo(int vetor[],
int inicio, int fim, Metricas*metricas, int tipoPivo)
{
   if (inicio < fim) {
   // Estratégia para escolher pivô:
   if (tipoPivo == 2) { // Mediana de três
   int meio = inicio + (fim - inicio) / 2;
   if (vetor[inicio] > vetor[meio])
   trocar(&vetor[inicio], &vetor[meio], metricas);
   if (vetor[inicio] > vetor[fim])
   trocar(&vetor[inicio], &vetor[fim], metricas);
   if (vetor[meio] > vetor[fim])
   trocar(&vetor[meio], &vetor[fim], metricas);
   trocar(&vetor[meio], &vetor[fim], metricas);
   } else if (tipoPivo == 0) { // Pivô central
       int meio = inicio + (fim - inicio) / 2;
       trocar(&vetor[meio], &vetor[fim], metricas);
       }
   // tipoPivo == 1 já usa o fim como pivô,
   não precisa trocar

   int pi=particionar
   (vetor, inicio, fim, metricas);
   quickSortRecursivo
   (vetor, inicio, pi - 1, metricas, tipoPivo);
   quickSortRecursivo
   (vetor, pi + 1, fim, metricas, tipoPivo);
   }
}
```

3.8 Bucketsort:

A ideia do método é dividir o intervalo que vai de 0 até k em n subintervalos de mesmo tamanho. Cada subintervalo estará associado a uma lista ligada que irá conter os elementos da lista que pertencem àquele subintervalo. Por exemplo, se a lista tem oito elementos é o maior deles é 71, teremos oito intervalos: [0, 9), [9, 18), ..., [63, 72). A posição zero do bucket apontará para uma lista encadeada que irá conter os elementos da lista que são maiores ou iguais a zero e menores que 9, e assim por diante. Chamaremos o bucket de vetor B. Para construir as listas encadeadas devemos inserir cada valor j contido na lista a ser ordenada na lista encadeada apontada por B[j * n / (k + 1)]. Em seguida, ordenamos as listas encadeadas com um método de ordenação qualquer (de preferência estável). Após isso, a concatenação das listas encadeadas produz a lista original ordenada. ideal para dados uniformemente distribuídos; alcança complexidade $\Theta(n)$ em condições favoráveis.

Código Bucketsort função principal:

```
Metricas bucketSort(int vetor[], int tamanho) {
    Metricas metricas = {0, 0};
    if (tamanho <= 0) return metricas;

    int maxValor = vetor[0];
    for (int i = 1; i < tamanho; i++) {
        if (vetor[i] > maxValor) {
            maxValor = vetor[i];
        }
    }
    maxValor++;

    int num_baldes = tamanho;
    No** baldes = (No**)calloc(num_baldes,
    sizeof(No*));
    if (!baldes) exit(1);

    for (int i = 0; i < tamanho; i++) {
     int indice_balde = (int)(((double)vetor[i] /
     maxValor) * num_baldes);
     baldes[indice_balde] = inserirNoBalde(baldes
     [indice_balde], vetor[i], &metricas);
    }

    int indice_vetor = 0;
    for (int i = 0; i < num_baldes; i++) {
        No* no_atual = baldes[i];
        while (no_atual != NULL) {
        vetor[indice_vetor++] = no_atual->valor;
        metricas.trocas++;
        No* temp = no_atual;
        no_atual = no_atual->proximo;
        free(temp);
        }
    }

    free(baldes);
    return metricas;
}
```

```
Código Bucketsort Função auxiliar::
static No* inserirNoBalde(No* cabeca, int valor,
Metricas* metricas) {
No* novoNo = (No*)malloc(sizeof(No));
if (!novoNo) exit(1);

    novoNo->valor = valor;
    metricas->trocas++;

if(cabeca == NULL || (metricas->comparacoes++,
    cabeca->valor >= valor)) {
        novoNo->proximo = cabeca;
        return novoNo;
    }

No* atual = cabeca;
 while (atual->proximo != NULL &&
 (metricas->comparacoes++, atual->proximo->valor
    < valor)) {
        atual = atual->proximo;
    }

    novoNo->proximo = atual->proximo;
    atual->proximo = novoNo;

    return cabeca;
}
```

3.9 Radixsort:

Esse método permite ordenar listas cujos elementos sejam comparáveis dois a dois. Uma versão desse método foi utilizada em 1887 por Herman Hollerith, fundador da Tabulating Machine Company que mais tarde deu origem à IBM Em cada iteração do Radixsort, ordenamos a lista por uma de suas posições, começando pela posição menos signifi cante, até chegar à posição mais signifi cante (os elementos da lista devem que ser representados com a mesma quantidade de posições). Cada ordenação tem que ser feita com um método estável. algoritmo que ordena elementos dígito a dígito, apresentando complexidade linear $\Theta(nk)$, onde k é o número de dígitos.

Código Radixsort:

```
Metricas radixSort(int vetor[], int tamanho) {
    Metricas metricas = {0, 0};
    if (tamanho <= 0) return metricas;

    int i;
    int max = vetor[0];
    for (i = 1; i < tamanho; i++) {
        if (vetor[i] > max) {
            max = vetor[i];
        }
    }

    for (int exp = 1; max / exp > 0; exp *= 10)
    {int* output = (int*)
    malloc(tamanho * sizeof(int));
        if (output == NULL) {
    fprintf(stderr, "ERRO: Falha ao alocar
    'output' no RadixSort.\n");
            return metricas;
        }

        int count[10] = {0};

        for (i = 0; i < tamanho; i++) {
            count[(vetor[i] / exp) % 10]++;
        }

        for (i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }

        for (i = tamanho - 1; i >= 0; i--) {
            output[count[(vetor[i] / exp) % 10] - 1]
            = vetor[i];
            metricas.trocas++;
            count[(vetor[i] / exp) % 10]--;
        }

        for (i = 0; i < tamanho; i++) {
            vetor[i] = output[i];
            metricas.trocas++;
        }

        free(output);
    }
    return metricas;
}
```

4. Resultados

4.1 Ambiente de teste e Limitações:

Todo o projeto foi desenvolvido em C, no VisualStudio Code, compilado utilizando GCC via terminal. A máquina utilizada nos testes possui o seguinte hardware: Intel I5 7400, 16gb Ram DDR4 (2333Mhz), Nvidia Gtx 1050ti (4GB), HD (110MBp/s), Windows 10 PRO (64Bits). Vale ressaltar que dada a escolha de design do software possa ser que ao rodar todos os algoritmos um por vez haverá de ter um stackOverflow em certos algoritmos, como o QuickSort(Pivô no Fim).

4.2 Resultados:

Nestas três tabelas são apresentados os valores de tempo, nûmero de comparações e trocas de cada algoritmo, em ordem crescente, decrescente e aleatória:

VETOR 9.999									
ORDEM	CRESCENTE			DECRESCENTE			ALEATÓRIA		
ALGORITMO	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas
BubbleSort	0,135	49.985	0	0,255	49.985.001	49.,985001	0,377	49.985.001	24.981
BubbleSort PARADA	0	9.998	0	0,204	49.985.001	49.985.001	0,289	49.960.911	25.082.982
InsertionSort	0	10	0	0,149	2.497.657.3	2.497.657.3	0,075	24.806.760	24.806.746
InsertionSort Binario	0,001	123.603	0	0,147	113.618	49.994.999	0,074	118.986	25.020.593
SelectionSort	0,11	49.985.001	0	0,115	49.985.001	4.999	0,112	49.985.001	9.985
MergeSort	0,002	69.001	267.202	0,003	64.600	267.202	0,003	120.500	267.202
QuickSort CENTRO	0,1	113.618	66.415	0	117.361	77.477	0,1	150.350	90.154
QuickSort FIM	0,335	49.985.001	49.994.9	0,226	49.985.001	24.999.999	0,2	168.289	104.402
QuickSort MED 3	0,1	125.424	66.415	0,2	223.459	135.853	0,2	145.308	84.241
BucketSort	0	0	19.998	0,1	0	19.998	0,2	4	19.998
ShellSort	0,001	119.996	0	0	168.163	168.163	0,02	257.733	199.494
RadixSort	0,11	0	79.992	0,11	0	79.992	0,11	0	79.992
HeapSort	0,02	244.435	24.996	0,02	226.707	24.996	0,02	235.463	24.996

Figura 6. Tabela de resultados para array de tamanho 9.999.

VETOR 99.999									
ORDEM	CRESCENTE			DECRESCENTE			ALEATÓRIA		
ALGORITMO	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas
BubbleSort	13000	499.850.001	0	24000	49.985.000.	499.850.00	35000	4.998.500	2.484.903
BubbleSort PARADA	0	99.998	0	20,666	4.999.850.0	4.999.850.0	29,373	4.999.711.926	2.501.826.4
InsertionSort	0	99.998	0	16,057	4.999.850.0	4.999.949.9	8,576	2.497.657.385	2.497.657.3
InsertionSort Binario	0,009	1.568.930	0	15,278	1.468.930	4.999.949.9	8,42	1.522.783	2.500.051.9
SelectionSort	11000	4.999.850.001	0	11000	4.999.850.0	49.999	11000	4.999.850.001	99.988
MergeSort	0,23	853.896	3.337.82	0,23	815.014	3.337.820	0,003	1.536.040	3.337.820
QuickSort CENTRO	0,8	1.468.930	846.095	0,8	1.513.064	899.704	0,18	2.122.888	1.216.339
QuickSort FIM	34,784	4.999.850.001	4.999.94	22,873	4.999.850.0	2.499.999.9	0,19	2.049.477	1.113.495
QuickSort MED 3	0,8	1.600.000	846.095	0,15	2.985.166	1.771.592	0,17	1.929.733	1.068.559
BucketSort	0,1	0	199.998	0,11	0	199.998	0,17	68.786	199.998
ShellSort	0,006	1.499.995	0	0,01	2.244.518	1.655.649	0,028	4.177.057	3.497.784
RadixSort	0,11	0	999.990	0,11	0	999.990	0,11	0	999.990
HeapSort	0,015	3.112.484	249.996	0,014	2.931.750	249.996	0,021	3.019.256	249.996

Figura 7. Tabela de resultados para array de tamanho 99.999.

VETOR 999.999									
ORDEM	CRESCENTE			DECRESCENTE			ALEATÓRIA		
ALGORITMO	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas	Tempo(s)	Comp.	Trocas
BubbleSort	1923000	499.998.500.0	0	2230000	499.998.500	499.998.500	3220	499.998.500.0	24.980.133.
BubbleSort PARADA	0,002	999.998	0	2064,622	499.998.500	499.998.500	2921,568	499.997.697.9	250.287.405
InsertionSort	0,003	999.998	0	1546,369	499.998.500	499.999.499	757,328	250.190.723.6	250.190.723
InsertionSort Binario	0,099	18.951.405	0	1455,266	17.951.426	499.999.499	760605	18.547.089	249.838.805
SelectionSort	150000	499.998.500.0	0	1148000	499.998.500	499.999	1191000	499.998.500.0	999.961
MergeSort	0,244	10.066.423	39.902.8	0,245	9.884.980	39.902.806	0,437	18.673.672	39.902.806
QuickSort CENTRO	0,9	17.951.426	9.933.56	0,102	18.395.623	10.544.512	0,277	34.640.883	26.794.628
QuickSort FIM	3430,394	499.998.500.0	499.999.	2303,34	499.998.500	249.999.999	0,273	34.650.003	25.395.852
QuickSort MED 3	0,93	19.000.000	9.933.56	0,2	37.891.653	22.388.069	0,274	34.059.248	25.313.662
BucketSort	0,1	0	1.999.99	0,105	0	1.999.998	0,255	967.231	1.999.998
ShellSort	0,071	17.999.994	0	0,111	26.736.674	19.145.025	0,355	62.079.877	53.066.728
RadixSort	0,14	0	11.999.9	0,141	0	11.999.988	0,118	0	9.999.990
HeapSort	0,164	37.689.586	2.499.99	0,169	35.977.136	2.499.996	0,276	36.793.949	2.499.996

Figura 8. Tabela de resultados para array de tamanho 999.999.

4.3 Discussão:

Agora uma ánalise mais unitária de cada algoritmo em cada cenário, refletindo as dificuldades e também vantagens obtidas por cada um.

1. O BubbleSort é notoriamente ineficiente para grandes volumes de dados. Nos vetores de 99.999 e 999.999, o tempo de execução ultrapassou 13 segundos e 32 minutos, respectivamente, mesmo em ordem crescente, o que demonstra seu comportamento quadrático ($O(n^2)$) mesmo nos melhores casos. Isso ocorre porque o algoritmo percorre o vetor inteiro várias vezes, independentemente da ordenação. Ele realiza sempre um número massivo de comparações e trocas próximo de n^2 e é muito sensível ao tamanho do vetor.

2. BubbleSort com Critério de Parada A versão otimizada com critério de parada apresentou melhora significativa para vetores em ordem crescente, com tempo praticamente nulo mesmo com 999.999 elementos. Isso reflete o melhor caso O(n) quando nenhum elemento precisa ser trocado. No entanto, o desempenho continua ruim para dados decrescentes ou aleatórios, com tempos de execução semelhantes ao BubbleSort padrão 24s e 29s em 99.999 elementos e crescendo ainda mais no vetor de 999.999. O número de comparações não é reduzido, mas o número de trocas é otimizado em cenários favoráveis.

3. InsertionSort O InsertionSort teve excelente desempenho em ordem crescente, com tempos nulos em todos os tamanhos de vetores, refletindo seu melhor caso linear. Porém, o comportamento se deteriora severamente com dados decrescentes: 16s com 99.999 e 1456s (mais de 24 minutos) com 999.999 elementos. Isso reforça que seu pior caso é $O(n^2)$, causado pelo grande número de movimentações necessárias para inserir itens em suas posições corretas. Em vetores aleatórios, também apresenta lentidão.

4. InsertionSort Binário A versão binária reduz o número de comparações, mas mantém o mesmo número de trocas. O tempo é um pouco melhor que o InsertionSort clássico em vetores decrescentes, porém ainda cresce exponencialmente com o tamanho dos dados. Mesmo com busca binária, o desempenho é ruim para ordens desorganizadas e vetores grandes. Em ordem crescente, como esperado, o tempo é desprezível.

5. SelectionSort O SelectionSort mostrou desempenho previsível. Mesmo com a ordem crescente, o algoritmo faz sempre o mesmo número de comparações, independentemente da ordem dos dados. Isso se reflete nos tempos praticamente constantes nos três tipos de ordenação. O tempo, porém, escala com o tamanho do vetor, chegando a 11 segundos para 99.999 elementos e 229 segundos para 999.999, mostrando seu custo

quadrático. Seu uso só é indicado quando trocas são muito custosas, o que não é o caso aqui.

6. MergeSort O MergeSort teve excelente desempenho em todas as ordens e tamanhos. Os tempos se mantêm baixos (milissegundos) mesmo com 999.999 elementos, refletindo seu comportamento O(n log n). Além disso, como é estável e divide o vetor recursivamente de forma balanceada, ele não sofre com a ordem inicial dos dados, mantendo desempenho semelhante em vetores crescentes, decrescentes ou aleatórios.

7. QuickSort – Pivô no Centro Para 9.999 elementos, o tempo foi bom, mas o algoritmo não é robusto o suficiente sem estratégias mais inteligentes de escolha do pivô. Para os demais tamanhos houve uma queda significativa no desempenho

8. QuickSort – Pivô no Fim O uso de pivô fixo (no fim) é especialmente problemático com dados crescentes ou decrescentes, pois causa partições extremamente assimétricas e degrada para $O(n^2)$. O comportamento é semelhante ao do pivô no centro, e ambos são inadequados para aplicações em larga escala sem melhorias. Nos seus testes com o vetor de ordem aleatória, o mesmo demonstrou um desempenho excelente. Para N=9.999, ele foi um dos mais rápidos, terminando em 0.002 segundos, com um número de comparações e trocas muito baixo. A história muda completamente quando o algoritmo enfrenta um vetor crescente ou decrescente. Seus resultados mostram um colapso total de performance

9. QuickSort – Mediana de Três (MED 3) Essa versão foi a mais eficiente entre os QuickSorts, evitando overflows e mantendo tempo de execução baixo mesmo com vetores aleatórios de 999.999 elementos. A estratégia de escolher a mediana de três elementos como pivô evita partições desbalanceadas, garantindo desempenho próximo de O(n log n) mesmo em ordens crescentes ou decrescentes. É uma solução prática e segura para aplicar o QuickSort em ambientes reais com grandes volumes.

10. BucketSort O BucketSort teve desempenho moderado: bom em vetores crescentes e decrescentes, mas pior do que esperado em aleatórios grandes. Isso pode ser resultado da má distribuição dos elementos entre os baldes ou da escolha de algoritmo de ordenação interna (provavelmente BubbleSort). Seu tempo cresce mais do que o ideal em vetores aleatórios com 999.999 elementos, o que limita sua utilidade prática fora de contextos com dados uniformemente distribuídos.

11. ShellSort O ShellSort obteve excelente desempenho prático, especialmente em dados aleatórios e vetores grandes. Os tempos ficaram abaixo de 0,3 segundos mesmo em 999.999 elementos, com desempenho melhor que os Selection/InsertionSorts e até mais rápido que algumas versões do QuickSort. A ordem dos dados afeta pouco seu desempenho, e ele se mostrou uma alternativa muito eficiente com implementação simples.

12. RadixSort O RadixSort apresentou comportamento linear e estável, com tempos baixos independentemente da ordem dos dados. Seu desempenho foi consistentemente bom, até mesmo em vetores de 999.999 elementos aleatórios (0,24s). Por não depender de comparações, ele mantém eficiência em qualquer ordenação, desde que os dados sejam inteiros ou tenham chave digital fixa. É ideal para grandes volumes com esse perfil de dados.

13. HeapSort O HeapSort teve bom desempenho e robustez, com tempos constantes em todas as ordens. Mesmo em vetores de 999.999 elementos aleatórios, ficou em torno de 0,4 segundos, um valor competitivo. Seu ponto forte é o comportamento garantido de O(n log n), mesmo nos piores casos. Embora não seja o mais rápido, é

confiável, pouco sensível à ordem e não causa estouros de pilha como algumas versões do QuickSort.

4.4 Conclusão

A análise experimental dos treze algoritmos de ordenação em vetores de até 999.999 elementos revela uma paisagem de desempenho diversificada, demonstram de forma inequívoca que a escolha de um algoritmo de ordenação não é uma decisão trivial, mas uma engenharia de trade-offs entre velocidade, consistência, uso de memória e a natureza dos dados de entrada.

A primeira e mais gritante conclusão é a divisão intransponível entre os algoritmos de complexidade quadrática (O(N2)) e os de complexidade log-linear (O(NlogN)) ou linear (O(N)). Enquanto os algoritmos Bubble Sort, Insertion Sort e Selection Sort se mostraram adequados para o vetor de 9.999 elementos, seu tempo de execução explodiu para dezenas de segundos ou minutos no teste de 99.999 e se tornaram computacionalmente inviáveis para 999.999 elementos. Este ”muro”de desempenho ilustra a importância fundamental da escolha de um algoritmo escalável. Já em contrapartida o Insertion Sort e o Bubble Sort com Critério de Parada provaram ser imbatíveis para dados já ordenados, terminando a tarefa em tempo linear (O(N)) e quase instantaneamente. O Selection Sort, por sua vez, manteve-se consistentemente lento em tempo, mas cumpriu sua promessa de minimizar as operações de escrita, realizando um número de trocas linear (O(N)) mesmo nos piores cenários.

No patamar dos algoritmos eficientes, o Quick Sort Nas suas versões com pivô Central e Mediana de 3, ele consistentemente apresentou os tempos de execução mais baixos para dados aleatórios, justificando sua reputação como um dos algoritmos de ordenação de propósito geral mais rápidos na prática. Contudo, a implementação com Pivô no Fim foi bem diferente, foi-se mostrada um fraqueza , ao ser confrontado com dados ordenados, sua performance degradou para O(N 2), tornando-se tão lento quanto um Bubble Sort e. Fazendo assim a Mediana de 3 como a implementação de Quick Sort mais segura e recomendável.

O Merge Sort e o Heap Sort emergiram como os pilares da consistência. Ambos entregaram um desempenho O(NlogN) sólido e previsível em todos os cenários, sem sofrer degradação em vetores ordenados ou inversos. O Merge Sort se destaca por ter um número de movimentações de dados fixo para um dado N, tornando seu tempo de execução o mais estável de todos, embora ao custo de alocar memória auxiliar significativa. O Heap Sort oferece a mesma garantia de desempenho O(NlogN) mas com a vantagem de ser um algoritmo in-place, o que o torna uma escolha superior quando a economia de memória é uma prioridade. O Shell Sort, por sua vez, provou ser um meio-termo notável, superando com folga os algoritmos quadráticos e se aproximando dos log-lineares, sendo uma excelente opção de fácil implementação e bom desempenho geral.

Os algoritmos de não-comparação, Radix Sort e Bucket Sort, demonstraram uma classe de performance à parte. Ao distribuir os elementos em vez de compará-los, eles alcançaram tempos de execução lineares, sendo os mais rápidos de todo o experimento para os dados inteiros testados. O resultado de zero comparações para o Radix Sort é a prova de seu paradigma distinto.

Em suma, a escolha do algoritmo de ordenação ideal é uma função direta do problema em questão. Para pequenos volumes ou dados quase ordenados, o Insertion Sort é uma escolha simples e eficaz. Para um desempenho geral de alta velocidade em cenários

com dados imprevisíveis, o Quick Sort (Mediana de 3) é o campeão. Para missões críticas que exigem estabilidade e performance garantida, Heap Sort (com economia de memória) e Merge Sort (com uso de memória extra) são as opções mais seguras. E, para o nicho de ordenação de grandes volumes de inteiros, o Radix Sort ainda é o melhor.

Disponibilidade do Código-Fonte: O código-fonte completo desenvolvido para este trabalho, incluindo todas as implementações, tabela de resultados e a estrutura de testes, está disponível publicamente no seguinte repositório do GitHub: `https://github.com/IllanSpala/ED-II`.

5. Referencias

[SILVA et al.], [Schwade and Mühlbeier], [Souza et al. 2017], [Viana et al. 2015], and [Pfitzner et al. 2009].

Referências

Pfitzner, A. L., Pinto, P. E. D., and da Costa, R. M. E. M. (2009). O algoritmo de ordenação smoothsort explicado. *Cadernos do IME-Série Informática*, 28:23–32.

Schwade, D. E. and Mühlbeier, A. R. K. Métodos de ordenação: Uma análise quantitativa para o pior caso.

SILVA, P., SCHANTZ, D., VILNECK, I., SILVEIRA, F., and CHICON, P. M. M. Análise do desempenho computacional dos métodos inserção direta, bolha, shellsort e combo-sort.

Souza, J. E., Ricarte, J. V. G., and de Almeida Lima, N. C. (2017). Algoritmos de ordenação: Um estudo comparativo. *Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574)*, 1(1).

Viana, G. V. R., Cintra, G. F., and Nobre, R. H. (2015). Ordenação de dados.