[**Introdução**](#_hq2oq3mcqsy6) **2**

[**Revisão Bibliográfica sobre os métodos**](#_k6vyrzj6m6a7) **3**

[2.1. Bubble Sort](#_46t7ssk1d5os) 3

[2.2. Inserção Direta](#_hhvgzlf238aw) 4

[2.3. Seleção Direta](#_rweqpl2diebl) 4

[2.4. Shake Sort](#_nv1nxxc1zrws) 5

[2.5. Quick Sort](#_ihprjeiy9me6) 6

[2.6. Merge Sort](#_ap8r86p4hpj3) 8

[2.7. Heap Sort](#_z8h7mdclbw2w) 10

[**Descrição do Experimento**](#_i7epiwt5za66) **12**

[**Resultados Gráficos e Tabelas de Valores**](#_9qnxnhxkqk1j) **12**

[**Discussão da Eficácia dos Métodos de Ordenação**](#_yysjepszauyq) **12**

[**Conclusão acerca dos Métodos Analisados**](#_oddbi6bk5orh) **12**

[**Bibliografia**](#_s2urk0nrkdza) **12**

# **Introdução**

Estrutura de Dados, de acordo com o professor da Unicamp, Ivan Luiz Marques Ricarte, em seu livro *Estrutura de Dados,* é *“o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento”*. Nesses mecanismos de organização de dados são utilizados tanto técnicas de armazenamento quanto de ordenação desses dados, sendo utilizados para as mais diversas finalidades, onde são definidos a organização, métodos de acesso e opções de processamento para os dados a serem utilizados dentro do programa.

Com diferentes tipos de métodos de ordenação, é natural que haja diferenças quanto às suas capacidades de desempenho, podendo serem melhor aplicadas em situações específicas, aumentando ainda mais as possibilidades de manipulação de dados e ganho de desempenho computacional em sistemas. Sobre essa questão, o objetivo principal desse trabalho é a análise de desempenho dos principais métodos de ordenação com relação aos tamanhos variados de vetores.

# **Revisão Bibliográfica sobre os métodos**

## **2.1. Bubble Sort**

Sendo considerado um dos algoritmos mais simples de ordenação, o Bubble Sort é um algoritmo recomendado para vetores pequenos, por ser uma ordenação de O(n²) para a quantidade de comparações de valores dentro do vetor, e M(n²) para a quantidade de movimentações entre os valores, ou registros, dentro de um vetor. (Daniel Viana, ***Conheça os principais algoritmos de ordenação***)

Nesse algoritmo, é feito a comparação inicial de valores entre um elemento e seu sucessor dentro vetor: caso o valor do elemento seja menor, ocorre a permutação do mesmo com o valor do elemento analisado; senão, o valor permanece na posição, ocorrendo a comparação com o próximo valor do próximo elemento. Esse processo finaliza somente caso, ao percorrer o vetor, não haja mais nenhuma permutação a ser realizada.

**Algoritmo utilizado no Experimento:**

*for( i = 0; i < 10; i++ ) {*

*for( j = i + 1; j < 10; j++ ) {*

*if ( vetor[i] > vetor[j] ) {*

*aux = vetor[i];*

*vetor[i] = vetor[j];*

*vetor[j] = aux;*

*}*

*contador++;*

*}*

*}*

## **2.2. Inserção Direta**

Essa ordenação ocorre com o percurso do vetor da esquerda para a direita, ordenando os elementos à esquerda desse vetor à medida que avança o percurso. Sendo também de ordenação O(n²), seu algoritmo tem um bom funcionamento desde que a ordem no qual os elementos estão não seja alterada durante a ordenação por meios externos à ela.

Seu funcionamento ocorre com a escolha do primeiro elemento do vetor, que será utilizado para comparação com o segundo elemento, onde, caso o mesmo seja menor que o primeiro, ocorre o descolamento para o primeiro elemento, ficando o primeiro e o segundo elemento ordenados. Depois, é feito essa comparação do primeiro e o segundo com o terceiro elemento, ordenando os três, para que esse processo possa ser feito para o quarto com relação aos três anteriores e vice-e-versa, até que todos os valores dos devidos elementos estejam corretamente ordenados.

**Algoritmo utilizado no** **Experimento:**

*for(i=1;i<100;i++) {*

*aux=vetor10\_2[i];*

*j=i-1;*

*while((j>=0)&&(aux<vetor10\_2[j])) {*

*vetor10\_2[j+1]=vetor10\_2[j];*

*j--;*

*contador++;*

*}*

*vetor10\_2[j+1] = aux;*

*}*

## **2.3. Seleção Direta**

Enquanto o método de ordenação anterior consistia em usar o valor do primeiro elemento para que fosse realizada as comparações e deslocamentos necessários, na Seleção Direta é inicialmente feito uma busca dentro do vetor pelo menor valor, para que este seja deslocado para o primeiro elemento. Depois, é capturado o próximo valor menor dentro desse vetor, descartando o valor capturado anteriormente, e é inserido no segundo elemento, deslocando seu valor anterior ao elemento que anteriormente armazenada o segundo menor valor do vetor, realizando todo esse processo até o final do vetor, quando todos os valores estão ordenados. Esse método, sendo de ordenação O(n²), não é considerado um algoritmo estável.

**Algoritmo utilizado no Experimento:**

*for(i=0;i<10;i++) {*

*menor=vetor10\_2[i];*

*posmenor=i;*

*for(j=i+1;j<10;j++) {*

*if(vetor10\_2[j]<menor) {*

*menor=vetor10\_2[j];*

*posmenor=j;*

*}*

*contador++;*

*}*

*vetor10\_2[posmenor]=vetor10\_2[i];*

*vetor10\_2[i]=menor;*

*}*

## **2.4. Shake Sort**

Sendo considerado um aprimoramento do método *Bubble Sort*, essa ordenação realiza duas varreduras, em ambos os casos realizando a permutação. Assim como seu antecessor, a comparação ocorre de forma linear entre os elementos, ou seja, a comparação do valor do primeiro elemento com o segundo, e assim por diante; porém, ao final da varredura padrão, da esquerda para a direita, é percorrido o vetor de forma inversa, sendo da direita para a esquerda, a fim de ordenar o menor valor para a posição inicial, tendo em vista que a primeira varredura garante que o maior valor assuma a posição do último elemento do vetor. Essa varredura direita e inversa ocorre até que todos os valores sejam comparados e alocados em seus respectivos elementos.

**Algoritmo utilizado no Experimento**

*do {*

*for(j=r;j>=posInicial;j--){*

*if(vetor10\_2[j-1]>vetor10\_2[j]) {*

*aux=vetor10\_2[j-1];*

*vetor10\_2[j-1]=vetor10\_2[j];*

*vetor10\_2[j]=aux;*

*k=j;*

*}*

*contador++;*

*}*

*posInicial=k+1;*

*for(j=posInicial;j<=r;j++){*

*if(vetor10\_2[j-1]>vetor10\_2[j]){*

*aux=vetor10\_2[j-1];*

*vetor10\_2[j-1]=vetor10\_2[j];*

*vetor10\_2[j]=aux;*

*k=j;*

*}*

*contador++;*

*}*

*r=k-1;*

*} while(posInicial<=r);*

## **2.5. Quick Sort**

O algoritmo de *Quick Sort* é considerado o método de ordenação interna mais rápido conhecido atualmente. De acordo com o autor Daniel Viana, *“é um algoritmo de comparação, que emprega a estratégia de ‘divisão e conquista’ “*, pois esse algoritmo inicialmente particiona o vetor em duas partes, comparando seus elementos e ordenando-os independente da outra parte.

Para que essa partição ocorra, é necessário que a mesma siga três definições: a primeira afirma que é escolhido um elemento pivô, de posição **d**, onde dele será utilizado a separação dos valores menores que o do elemento pivô, situados à sua esquerda, sendo essa a segunda definição, e os valores maiores que o dele, situados à sua direita, sendo essa a terceira definição. Essa separação ocorre com a varredura do vetor da esquerda para a direita, a fim de encontrar um valor maior que o do pivô, e depois realiza a varredura inversa, para encontrar um valor maior que o do mesmo, estando estes fora de posição, sendo por fim trocados, realizando esse processo até que seja feito a varredura e seja encontrado um valor igual ao do pivô, da mesma posição **d**, representando assim o final da separação.

Por fim, o método se encarrega de realizar as comparações entre as partições criadas, ordenando-as conforme for particionando.

**Algoritmo utilizado no Experimento:**

*void quicksort(int e, int d){*

*int i;*

*if(d > e){*

*i= particao(e,d);*

*quicksort(e,i-1);*

*quicksort(i+1,d);*

*}*

*}*

*int particao(int e, int d){*

*int v,i,j;*

*v = a[d];*

*i = e -1;*

*j = d;*

*do{*

*do{*

*i = i+1;*

*cont++;*

*}while((a[i] < v) && (i < d));*

*do{*

*j = j-1;*

*cont++;*

*}while((a[j] > v ) && (j > 0));*

*t = a[i];*

*a[i] =a[j];*

*a[j] = t;*

*}while(j > i);*

*a[j] = a[i];*

*a[i] = a[d];*

*a[d] = t;*

*return(i);*

*}*

## 

## 

## **2.6. Merge Sort**

Assim como o *Quick Sort*, ele realiza a partição do vetor, porém ao invés de utilizar essa partição para somente comparação de valores e trocar os valores para a posição correta, como ocorre nesse método, o *Merge Sort* literalmente particiona o vetor em outros sub-vetores, sendo divididos sempre em duas partes iguais, ou próximas de serem iguais, até que sejam formados sub-vetores de no máximo dois elementos para ser realizado uma comparação simples, e por fim é unido(*merge*) todos esses sub-vetores, agora formando o vetor original ordenado. A comparação e ordenação desse processo ocorre após a partição do vetor até os menores sub-vetores possíveis, onde ocorre a comparação dos sub-vetores de forma linear, e eles são unidos em sub-vetores maiores de forma ordenada, nos quais serão comparados linearmente entre si, e unidos na ordenação correta desses grupos, até por fim os dois maiores sub-vetores serem enfim comparados e unidos ordenadamente, formando o vetor originalmente inserido, desta vez devidamente ordenado.

**Algoritmo utilizado no Experimento:**

*void mergesort(int e, int d, int v[]) {*

*int meio;*

*if(e<d-1) {*

*meio = (e+d)/2;*

*mergesort(e,meio,v);*

*mergesort(meio,d,v);*

*intercala(e,meio,d,v);*

*}*

*}*

*void intercala (int e, int meio, int d, int v[]) {*

*int i,j,k,\*w;*

*w = (int\*) malloc ((d-e) \* sizeof(int));*

*i=e;*

*j=meio;*

*k=0;*

*while(i<meio && j<d) {*

*if (v[i] < v[j]) {*

*w[k++] = v[i++];*

*}*

*else {*

*w[k++] = v[j++];*

*}*

*contador++;*

*}*

*while (i<meio) {*

*w[k++] = v[i++];*

*contador++;*

*}*

*while (j<d) {*

*w[k++] = v[j++];*

*contador++;*

*}*

*for (i=e;i<d;++i) {*

*v[i] = w[i-e];*

*contador++;*

*}*

*free(w);*

*}*

## **2.7. Heap Sort**

Para que esse método de ordenação possa ser utilizado, é necessário que a estrutura de dados que os valores estão armazenados seja o **Heap**. Um Heap é uma estrutura de dados baseada em Árvores Binárias, com a principal definição: *Todos os nós pais devem ter seus valores maiores que os dos nós filhos*. Essa estrutura é representada do ponto de vista computacional como um vetor, onde no primeiro elemento do vetor se encontra a raiz, ou o primeiro elemento do heap, o nó filho da esquerda da raiz fica armazenado na segunda posição do vetor, enquanto que o nó da direita fica na terceira posição, e vice-e-versa.

Esse vetor criado a partir do heap é denominado **Max-heap**, representando o conceito de que os nós pais possuem valores maiores que os nós filhos através da seguinte equação:

*“Max-heap é um vetor de [1,...,M] elementos, tal que: v[f/2] > v[f], para f = 2,...,M.”* (Professor César, **MergeSort e HeapSort**)

Ou seja, um Max-heap ocorre somente se os valores de seus elementos sejam maiores que seus posteriores.

Para a manipulação do vetor Max-heap por parte do algoritmo Heap Sort, é necessário uma função que inicialmente receba um vetor qualquer e uma posição qualquer do mesmo, onde ocorrerá a análise do valor dessa posição com os demais elementos, deslocando-o até o elemento posterior correto, caso haja necessidade. Depois de gerado o Max-heap, é pego o valor do primeiro elemento, sendo a raiz do Heap, e troca-se pelo valor do último elemento desse vetor, estando o maior valor do vetor armazenado no final do mesmo, para que possa ser usado novamente a função de gerar Max-heap, denominado para esse experimento como *Peneira*, desta vez isolando o último elemento. Esse processo é realizado até que todos os elementos sejam alinhados, ou seja, quando é realizado a troca do elemento presente no meio do vetor pelo seu antecessor.

**Algoritmo utilizado no Experimento:**

*void peneira (int p, int m, int v[ ]) {*

*int x = v[p];*

*while (2\*p <= m) {*

*int f = 2\*p;*

*if (f < m && v[f] < v[f+1])*

*++f;*

*if (x >= v[f])*

*break;*

*v[p] = v[f]; p = f;*

*}*

*v[p] = x;*

*}*

*void heapsort (int n, int v[]) {*

*int p, m, x;*

*for (p = n/2; p >= 1; --p)*

*peneira (p, n, v);*

*for (m = n; m >= 2; --m) {*

*x = v[1];*

*v[1] = v[m];*

*v[m] = x;*

*peneira (1, m-1, v);*

*}*

*}*

# **Descrição do Experimento**

O experimento realizado teve início com o uso dos principais métodos de ordenação: Inserção Direta, Seleção Direta, Bubble Sort, Shake Sort, Quick Sort, Merge Sort e Heap Sort.

Para comparar o desempenho dos mesmos, foi utilizado em cada processo um contador, o qual era responsável por armazenar o número de ciclos que cada ordenação realizou. O valor armazenado pelo contador difere de acordo com o tipo de ordenação e o tamanho do vetor utilizados.

Por fim, os tamanhos utilizados dos vetores tem escala de grandeza científica, ou seja, todos eles são derivados de notações científicas: 10^1(10), 10^2(100), 10^4(10 000), 10^6(1 000 000), 10^8(100 000 000) e 10^9(1 000 000 000). Cada método ordenará um vetor com esse tipo de tamanho, sendo gerado um valor armazenado em um contador para cada um deles, sendo usado posteriormente para a construção dos gráficos e análise dos valores, através de um pequeno relatório acerca dos valores encontrados e uma conclusão dos mesmos.

# **Resultados Gráficos e Tabelas de Valores**

# **Discussão da Eficácia dos Métodos de Ordenação**

# **Conclusão acerca dos Métodos Analisados**

# **Bibliografia**

Ivan Luiz Marques Ricarte,  **Estrutura de Dados**. Em <<http://calhau.dca.fee.unicamp.br/wiki/images/0/01/EstruturasDados.pdf>> Acesso em 17 de Novembro de 2018.

Daniel Viana, **Conheça os Principais Algoritmos de Ordenação**.Em

<[https://www.treinaweb.com.br/blog/conheca-os-principais](https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/)

[-algoritmos-de-ordenacao/](https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/)> Acesso em 18 de Novembro de 2018.

Professor César, **MergeSort e HeapSort**. Acesso em 20 de Novembro de 2018.