UFOP – UNIVERSIDADE FEDERAL DE OURO PRETO
ICEB – INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DECOM – DEPARTAMENTO DE COMPUTAÇÃO

LUCAS GOMES DOS SANTOS
VITOR HUGO LELES FONSECA

TRABALHO PRÁTICO – AVALIAÇÃO EMPÍRICA

OURO PRETO
2023

LUCAS GOMES DOS SANTOS - 20.1.4108

VITOR HUGO LELES FONSECA - 17.2.4324

TRABALHO PRÁTICO – AVALIAÇÃO EMPÍRICA

Relatório do trabalho prático da disciplina BCC241 - Projeto e Análise de Algoritmos, que consiste em uma avaliação empírica de três algoritmos de ordenação

OURO PRETO

2023

SUMÁRIO

OBJETIVO

Este trabalho tem como objetivo avaliar empiricamente algoritmos de ordenação, descrevendo a análise de complexidade quanto ao tempo de execução, em relação à entradas idênticas para diferentes métodos. Serão estudados e comparados os seguintes algoritmos: Merge Sort, Selection Sort e Radix Sort. Para análise, os algoritmos são implementados, executados e os resultados comparados. Após esta análise, será possível concluir qual método é o mais recomendado para cada tipo de situação.

INTRODUÇÃO

A ordenação pode ser definida como o processo de rearranjar um conjunto de elementos em uma ordem ascendente ou descendente. Esses elementos devem apresentar valores de um domínio em comum, de modo que é possível estabelecer uma relação entre eles e, consequentemente, ordená-los.

A ordenação pode ser classificada como interna ou externa. Os algoritmos que adotam a estratégia de **ordenação interna** têm os elementos a serem ordenados e seus dados sempre contidos em memória. Já os algoritmos baseados na estratégia de **ordenação externa** manipulam elementos contidos em arquivos de texto, binários e outros dispositivos de armazenamento, ou seja, na memória externa.

Os algoritmos baseados em ordenação interna utilizam de comparações ou de contagem para definir uma ordem entre os elementos, o Merge Sort, Insertion Sort e RadixSort são algoritmos de ordenação interna. O Merge Sort e Insertion Sort utilizam a estratégia de comparação e o Radix Sort utiliza a estratégia de contagem para ordenar os elementos.

No desenvolvimento do trabalho os algoritmos foram testados de modo que dado um vetor com n valores inteiros como entrada, é retornado um vetor contendo esses valores, de modo que o valor de cada posição i é menor ou igual ao da posição *i+1*, com $1 \leq i < n$.

ANÁLISE DE COMPLEXIDADE

De acordo com a proposta apresentada para o trabalho: dado um vetor com n valores inteiros como entrada, os algoritmos devem retornar um vetor com os valores do vetor de entrada, porém o valor de cada posição i deve ser menor ou igual ao da posição i+1, com $1 \leq i < n$. Foram implementados os 3 algoritmos a serem avaliados, sendo eles: Merge Sort, Insertion Sort e Radix Sort.

Abaixo estão descritas a análise e as complexidades encontradas para cada algoritmo testado no desenvolvimento do trabalho.

MergeSort – O(n log n)

A ideia do MergeSort é dividir o vetor em dois subvetores, cada um com metade dos elementos do vetor original. Esse procedimento é então reaplicado aos dois subvetores recursivamente.

Quando os subvetores têm apenas um elemento (caso base), a recursão para. Então, os subvetores ordenados são fundidos (ou intercalados) num único vetor ordenado.

A complexidade deste algoritmo é dada pelas operações de divisão em sub vetores menores até que exista somente um único elemento, conquistar - ordenar os subvetores provenientes do vetor original, e por fim, combinar as soluções do subproblema. O passo seguinte também influencia na ordem de complexidade do algoritmo - a combinação. Cada item na lista irá ser processado em algum momento e colocado na lista ordenada. Então a operação de combinar resultará em uma lista de tamanho n requerendo n operações. A análise resulta em log(n) divisões, cada qual custando n, totalizando n log (n) operações. Logo,podemos concluir que o MergeSort é um algoritmo da ordem de complexidade O(n log(n)).

```
void mergeSort (int* v , int n) {
   mergeSort_ordena (v , 0, n -1) ;
}

// Ordena o vetor v[ esq .. dir ]
void mergeSort_ordena (int *v, int esq, int dir) {
   if ( esq >= dir )
   return ;

   int meio = ( esq + dir ) / 2;
   mergeSort_ordena (v,esq,meio);
   mergeSort_ordena (v,meio+1,dir);
   mergeSort_intercala (v,esq,meio,dir);
}
```

```
// Intercala os vetores v[esq .. meio ] e v[ meio +1.. dir ]
void mergeSort_intercala (int* v , int esq , int meio , int dir) {
   int i , j , k ;
   int a_tam = meio - esq +1;
   int b_tam = dir - meio ;
   int* a = new int [a_tam];
   int* b = new int [b_tam];
   for (i=0; i< a_tam ; i ++){
       a[i] = v[i+ esq];
   }
   for (i = 0; i < b_tam ; i ++){
       b[i] = v[i+meio+1];
   }
   i = 0; // Índice para o vetor 'a'
   j = 0; // Índice para o vetor 'b'
   k = esq; // Índice para o vetor 'v'

   while (i < a_tam && j < b_tam) {
       if (a[i] < b[j]) {
           v[k++] = a[i++];
       } else {
           v[k++] = b[j++];
       }
   }

   // Copia os elementos restantes do vetor 'a', se houver
   while (i < a_tam) {
       v[k++] = a[i++];
   }
```

```
   // Copia os elementos restantes do vetor 'b', se houver
   while (j < b_tam) {
       v[k++] = b[j++];
   }
}
```

InsertionSort – O(n ²)

O InsertionSort pode ser entendido como o algoritmo utilizado pelo jogador de cartas, as cartas são ordenadas uma por uma da esquerda da direita. A segunda carta é verificada, analisando se deve ficar antes ou na posição que está, depois a terceira carta é classificada, sendo deslocada até sua posição correta e assim por diante.

Consiste em trocar o menor elemento do conjunto pelo elemento que está no início da lista, depois o segundo menor elemento pelo que está na segunda posição e assim sucessivamente, até os dois últimos elementos do conjunto. Desse modo, temos que o Insertion Sort é *O(n²).* Esse método é vantajoso para um conjunto pequeno de dados, uma vez que sua complexidade é quadrática.

```
void insertionsort ( int* vector, int n) {
 int i,j;                              // O(1)

 for (i = n -2; i >= 0; i --) {        // O(n) -> n - 1 iterações
   vector[n] = vector[i];              // O(1)
   j = i + 1;                          // O(1)
   while (vector[n] > vector[j] ) {    // O(n) -> No pior caso n - 1
iterações
     vector[j - 1] = vector[j];        // O(1)
     j ++;                             // O(1)
   }

 vector[j - 1] = vector[n];            // O(1)
```

```
}
}                                           // = O(n^2)
```

RadixSort – O(n)

O Radix Sort é um algoritmo baseado no conceito de ordenação por contagem, que consiste em criar uma tabela de N entradas, onde N é o número de elementos diferentes que pode ser inserido no vetor de entrada. E nesta tabela são inseridos contadores, incrementados se os mesmos forem correspondentes a chave de valor i, após realizado este processo, é conhecido a quantidade de posições necessárias para cada valor de chave, assim os elementos são transferidos para as posições corretas no novo vetor ordenado. A análise de complexidade do RadixSort, leva em consideração a quantidade de dígitos que compõem a chave pesquisada. Neste caso do trabalho em questão devemos considerar que para as instâncias sugeridas o tamanho independente da potência de 10 terá a mesma quantidade de dígitos. Consideramos uma constante “k”, e como sua vez o algoritmo deverá ser capaz de fazer a leitura destes “k” dígitos nas “n” chaves que comporão cada instância. A ordem de complexidade é dada pela multiplicação do número de dígitos pela quantidade de elementos, k*n e de notação Θ (nk) no seu pior caso que seria em um vetor totalmente desordenado.

```
int getMax(int* vector, int n)
{
   int max = vector[0];
   for (int i=1; i<n; i++)
       if (vector[i] > max)
           max = vector[i];
   return max;
}
// Using counting sort to sort the elements in the basis of
significant places
void countingSort(int* array, int size, int place) {
 const int max = 10;
 int output[size];
 int count[max];
```

```
 for (int i = 0; i < max; ++i)
   count[i] = 0;

 // Calculate count of elements
 for (int i = 0; i < size; i++)
   count[(array[i] / place) % 10]++;

 // Calculate cumulative count
 for (int i = 1; i < max; i++)
   count[i] += count[i - 1];

 // Place the elements in sorted order
 for (int i = size - 1; i >= 0; i--) {
   output[count[(array[i] / place) % 10] - 1] = array[i];
   count[(array[i] / place) % 10]--;
 }

 for (int i = 0; i < size; i++)
   array[i] = output[i];
}

// Main function to implement radix sort
void radixsort(int* array, int size) {
 // Get maximum element
 int max = getMax(array, size);

 // Apply counting sort to sort elements based on place value.
 for (int place = 1; max / place > 0; place *= 10)
```

```
    countingSort(array, size, place);

}
```

DESENVOLVIMENTO

A análise foi realizada de forma empírica, testando o tempo de execução de cada um para instâncias de tamanhos diferentes. Foi proposta a geração de 20 instâncias para cada algoritmo e a avaliação de sua complexidade de tempo. As instâncias são de tamanho n = 100, 200, 400, 800, 1.600, 3.200, …, ou seja, multiplicando por 2.

Os algoritmos de ordenação tiveram suas chamadas inseridas no driver da aplicação, função *main( )*, sendo contado o tempo do início até o final da execução com a função *clock( )*, que conta os milissegundos que passam até do início ao final da execução do bloco de código que encontra entre a função.

RESULTADOS

Merge Sort

Com o Merge Sort, de complexidade *O(n log n)* foi possível realizar testes com instâncias maiores, uma vez que a complexidade é logarítmica, de modo que é adequado para instâncias maiores.

Merge Sort	
100	0 ms
200	0 ms
400	0 ms
800	0 ms
1.600	0 ms
3.200	0 ms
6.400	1 ms
12.800	3 ms
25.600	6 ms
51.200	19 ms
102.400	28 ms
204.800	65 ms

409.600	124 ms
819.200	267 ms
1.638.400	577 ms
3.276.800	1150 ms
6.553.600	2426 ms
13.107.200	5002 ms
26.214.400	10434 ms
52.428.800	86663 ms

Tabela 1: Resultados obtidos para o tempo de execução do Merge Sort

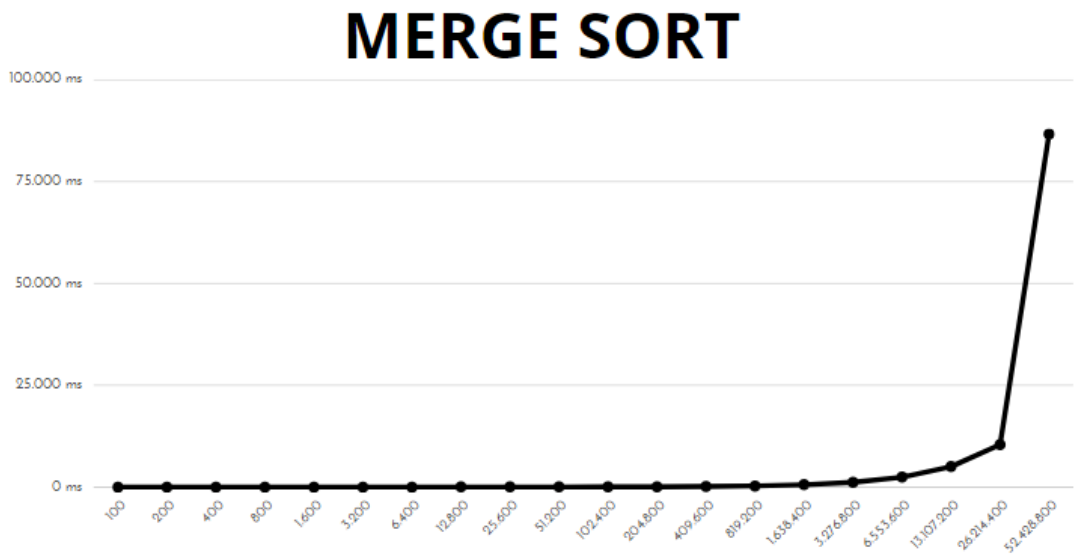

Gráfico 1: Resultados obtidos para o tempo de execução do Merge Sort

Radix Sort

Com o Radix Sort foi possível realizar testes para instâncias menores, mesmo que o algoritmo tenha complexidade linear não é adequado para conjuntos de dados muito grandes, uma vez que utiliza a estratégia de contagem, além disso, a entrada tem que apresentar uma configuração específica para que tenha execução rápida.

Radix Sort	
100	0 ms
200	0 ms

400	0 ms
800	0 ms
1.600	0 ms
3.200	0 ms
6.400	1 ms
12.800	1 ms
25.600	2 ms
51.200	8 ms
102.400	18 ms
204.800	35 ms
409.600	69 ms
819.200	104 ms
1.638.400	237 ms
3.276.800	-
6.553.600	-
13.107.200	-
26.214.400	-
52.428.800	-

Tabela 2: Resultados obtidos para o tempo de execução do Radix Sort

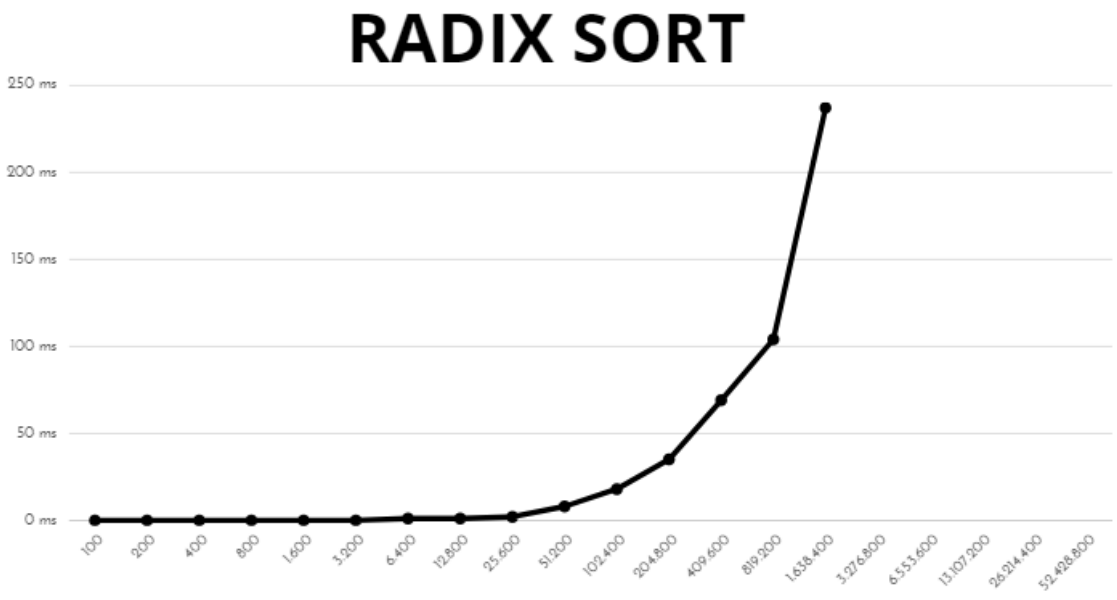

Gráfico 2: Resultados obtidos para o tempo de execução do Radix Sort

Insertion Sort

Com o Insertion Sort também foi possível realizar testes apenas para instâncias menores, uma vez que sua complexidade é $O(n^2)$. O custo de se classificar cada item é muito alto, uma vez que cada um é classificado fazendo comparações com todos os outros itens do conjunto. Desse modo, esse algoritmo não é adequado para instâncias muito grandes, sendo recomendado para conjuntos menores de dados, em casos que o conjunto está parcialmente ordenado ou quando se deseja inserir alguns dados em um conjunto.

Insertion Sort	
100	0 ms
200	0 ms
400	0 ms
800	1 ms
1.600	7 ms
3.200	11 ms
6.400	53 ms
12.800	122 ms
25.600	411 ms
51.200	1523 ms
102.400	5991 ms
204.800	23776 ms
409.600	96486 ms
819.200	418728 ms
1.638.400	-
3.276.800	-
6.553.600	-
13.107.200	-
26.214.400	-
52.428.800	-

Tabela 3: Resultados obtidos para o tempo de execução do Insertion Sort

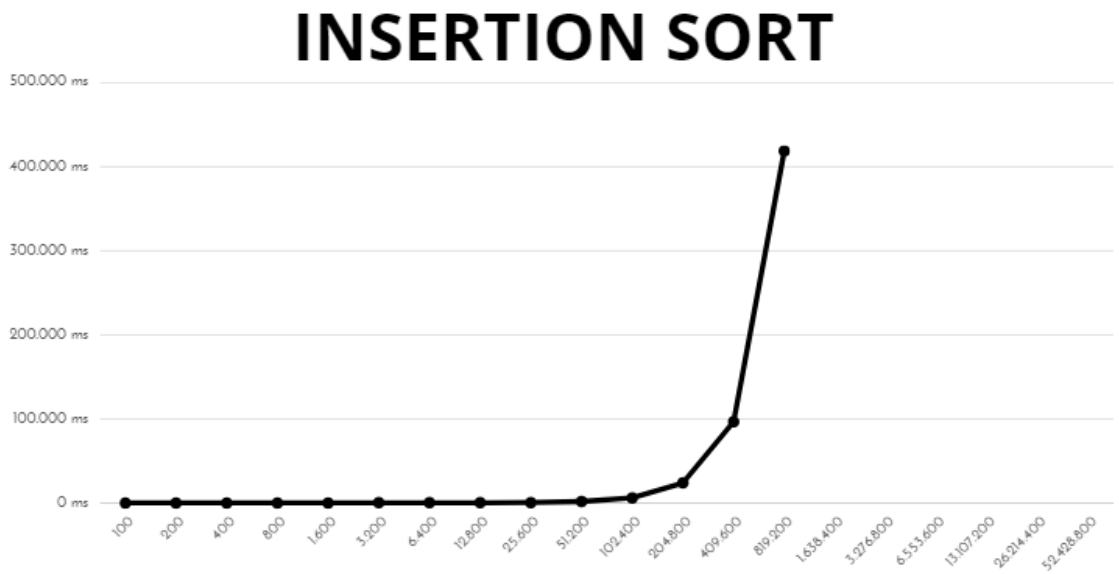

Gráfico 3: Resultados obtidos para o tempo de execução do Insertion Sort

CONCLUSÃO

O objetivo deste trabalho foi demonstrar uma avaliação empírica de três algoritmos de ordenação: Merge Sort, Insertion Sort e Radix Sort. Esses algoritmos são amplamente conhecidos e estudados, havendo diversos trabalhos científicos desenvolvidos sobre eles. Cada um desses algoritmos apresenta suas estratégias próprias de implementação, sua complexidade e os cenários para qual sua aplicação é vantajosa.

O Radix Sort é eficiente para ordenar inteiros com um número fixo de dígitos, mas pode exigir mais memória. O Merge Sort é eficiente para todas as instâncias, mantendo um bom desempenho. O Insertion Sort é simples e eficiente para pequenos conjuntos de dados, mas se torna menos prático para conjuntos maiores.

Desse modo, temos algoritmos que adotam diferentes estratégias de implementação e complexidades diferentes para diversos casos, de modo que é necessária uma ampla análise do contexto de aplicação, assim como o funcionamento e complexidade dos algoritmos para se utilizar aquele mais adequado.

REFERENCIAL BIBLIOGRÁFICO

LAUREANO, Marcos. “Estrutura de Dados com Algoritmos e C”, 2008.

RICARTE, Ivan Luiz Marques. “Estruturas de dados”, 2008. UNICAMP. São Paulo – SP

JÚNIOR, Edwar Salliba. “Estruturas de Dados – Notas de Aula”, 2007. Faculdade de Tecnologia INED. Belo Horizonte – MG

Material didático fornecido pela professora Amanda Sávio na disciplina de Estruturas de Dados I.

Material didático fornecido pelo professor Guilherme Tavares na disciplina de Estrutura de Dados II.