PROJETO E ANÁLISE DE ALGORITMOS TRABALHO PRÁTICO I

Bárbara Letícia Rodrigues Milagres

Carlos Gabriel De Freitas

Laura Martins da Costa Coura Marinho

SUMÁRIO

1. InsertionSort
2. MergeSort
3. RadixSort
4. Resultados
5. Referências Bibliográficas

INSERTIONSORT

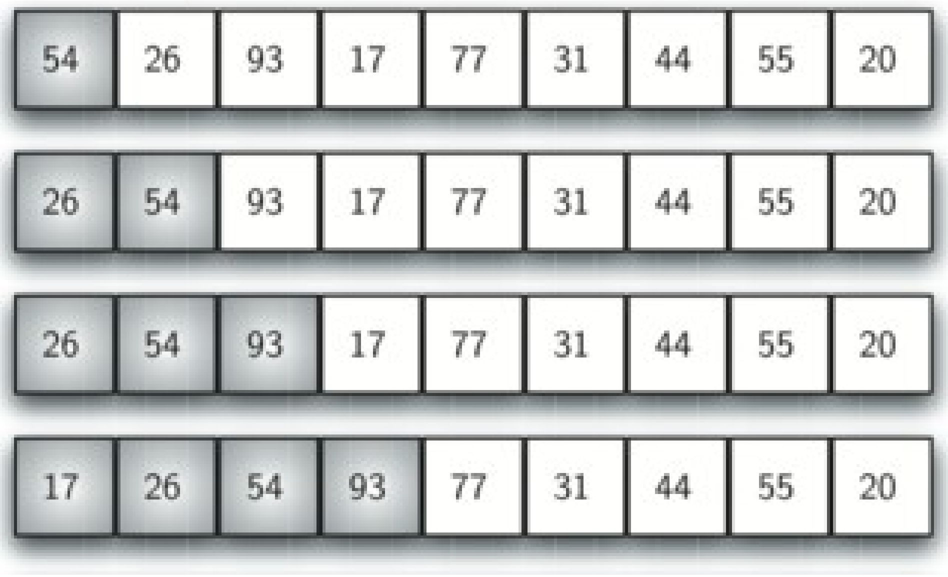
CÓDIGO

```c
// Realiza o InsertionSort em um vetor de inteiros v, de tamanho n
void insertionSort(int* v, int n){
    int j;
    int carta; // chave que sera inserida a cada iteração do loop for

    for(int i = 1; i < n; i++) {
        carta = v[i];
        j = i - 1;

        while(j >= 0 && v[j] > carta){
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = carta;
    }
}
```

INSERTIONSORT

COMPLEXIDADE

Primeiramente observa-se o loop de repetição *for*, que resulta em uma complexidade O(n), e, em seguida, observa-se o loop de repetição *while* dentro do *for*, cuja complexidade em seu pior tempo é O(n). Desse modo, o cálculo do custo total é feito através de T(n) = O(n) * O(n), o que resulta em T(n) = O(n^2).



MERGESORT

CÓDIGO

```
// Realiza o MergeSort em um vetor de inteiros v, com o indice inicial "comeco" e indice final "fim"
void mergeSort(int* v, int comeco, int fim){
    if(comeco < fim){
        int meio = (comeco+fim)/2;

        mergeSort(v, comeco, meio);
        mergeSort(v, meio+1, fim);
        merge(v, comeco, meio, fim);
    }
}
```

MERGESORT

CÓDIGO

```
// Funcao auxiliar de mergeSort()
void merge(int* v, int comeco, int meio, int fim) {
    int i = comeco; // Variável de controle do primeiro "sub-vetor"
    int j = meio+1; // Variável de controle do segundo "sub-vetor"
    int k = 0;      // Variável de controle do vetor auxiliar
    int n = fim-comeco+1;

    int *vAux = malloc(n * sizeof(int)); // Vetor auxiliar

    while(i <= meio && j <= fim){
        if(v[i] < v[j]) {
            vAux[k] = v[i];
            i++;
        } else {
            vAux[k] = v[j];
            j++;
        }
        k++;
    }
```

MERGESORT

CÓDIGO

```
    //Caso ainda haja elementos na primeira metade
    while(i <= meio){
        vAux[k] = v[i];
        k++;
        i++;
    }

    //Caso ainda haja elementos na segunda metade
    while(j <= fim) {
        vAux[k] = v[j];
        k++;
        j++;
    }

    //Move os elementos de volta para o vetor original
    for(k = comeco; k <= fim; k++){
        v[k] = vAux[k-comeco];
    }

    free(vAux);
}
```

COMPLEXIDADE

Caso base: $T(1) = O(1)$

Chama recursivamente para as duas metades do vetor duas vezes: $T(n) = 2 * T(n/2)$.

Custo local: $T(n) = O(n)$

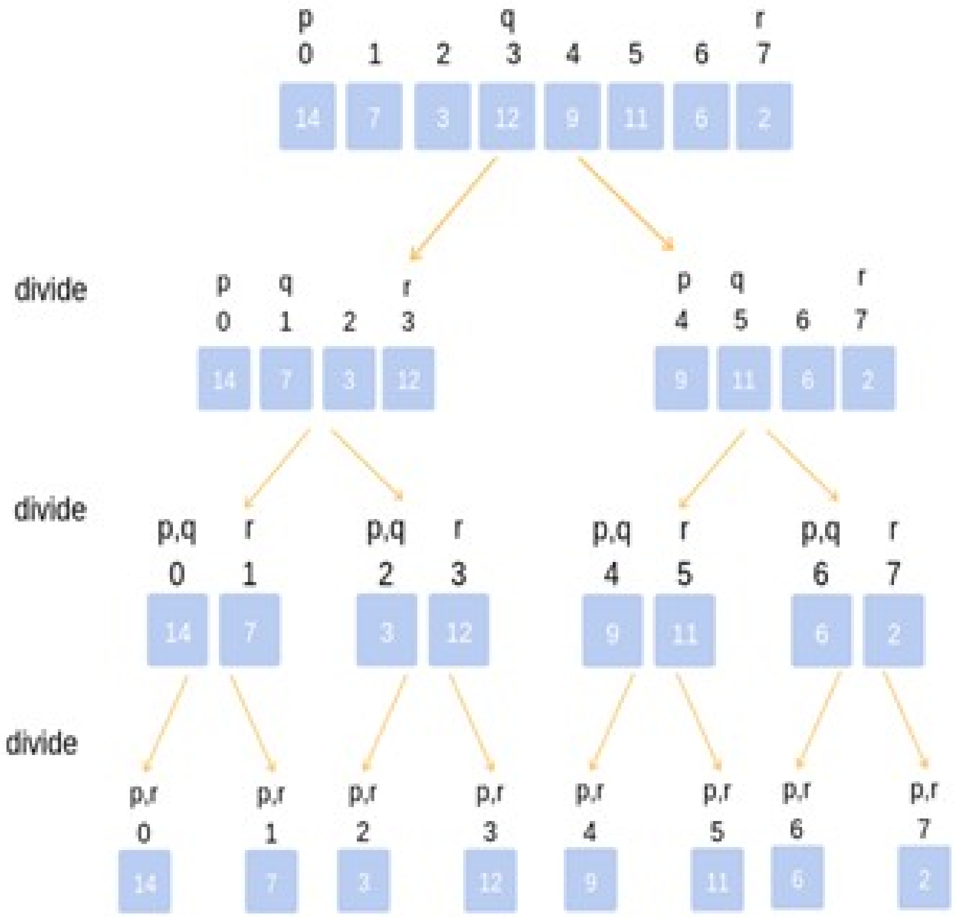
Logo:

$T(1) = O(1)$

$T(n) = 2 * T(n/2) + O(n)$

A partir do Teorema Mestre, tem-se que para um $T(n) = aT([n/b]) + O(n^d)$ com as constantes $a > 0$, $b > 1$ e $d >= 0$, $T(n) = O(n^d * \log n)$ se d =.

Neste caso, , o que implica que = d. Portanto, conclui-se que a complexidade do tempo do *MergeSort* é $T(n) = O(n * \log n)$.



RADIXSORT

CÓDIGO

```c
//Realiza o Radixsort com o vetor v de tamanho n
void radixSort(int* v, int n) {
    int i;
    int *b;
    int maior = v[0];
    int exp = 1;

    b = malloc(n * sizeof(int));

    for (i = 0; i < n; i++) {
        if (v[i] > maior)
            maior = v[i];
    }
```

```c
    while (maior/exp > 0) {
        int bucket[10] = { 0 };
        for (i = 0; i < n; i++)
            bucket[(v[i] / exp) % 10]++;
        for (i = 1; i < 10; i++)
            bucket[i] += bucket[i - 1];
        for (i = n - 1; i >= 0; i--)
            b[--bucket[(v[i] / exp) % 10]] = v[i];
        for (i = 0; i < n; i++)
            v[i] = b[i];
        exp *= 10;
    }

    free(b);
}
```

RADIXSORT

COMPLEXIDADE

- Pior caso: linha “maior = v[i]”, complexidade O(n).
- Loop de repetição *while:* repete k vezes (O(k)) e tem possuindo quatro *loops* de repetição *for*, três de complexidade O (n) e um de O(10).
- Linha “exp *= 10” possui complexidade O (1)

Logo:
T(n) = O(k) * [O(n)+ O(10) + O(n) + O(n)]
Aplicando regra da soma:
T(n) = O(k) * O(n)
Aplicando regra da multiplicação e considerando k uma constante:
T(n) = O(n)

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**
1**4**2	1**3**2	1**2**3	2**6**3	2**3**3	0**1**4	0**8**7
014	**1**23	**1**32	**2**33	**1**42	**2**63	**0**87
014	087	123	132	142	233	263

RESULTADOS

Tamanho (n)	100	1000	10000	100000	1000000
InsertionSort	0.0000	0.0008	0.1672	15.8664	1385.0812
MergeSort	0.0000	0.0016	0.0031	0.0328	0.3273
RadixSort	0.0000	0.0000	0.0031	0.0156	0.2141

Média dos tempos para ordenação dos vetores em relação ao seu tamanho.

Tamanho (n)	100	1000	10000	100000	1000000
InsertionSort	(0.0000, 0.0000)	(-0.0009, 0.0024)	(0.1516, 0.1828)	(15.1994, 16.5334)	(1296.9020, 1473.2605)
MergeSort	(0.0000, 0.0000)	(-0.0007, 0.0038)	(0.0001, 0.0061)	(0.0288, 0.0369)	(0.2928, 0.3619)
RadixSort	(0.0000, 0.0000)	(0.0000, 0.0000)	(0.0001, 0.0061)	(0.0123, 0.0190)	(0.1951, 0.2331)

Intervalo de confiança dos métodos de ordenação em relação ao tamanho dos vetores.

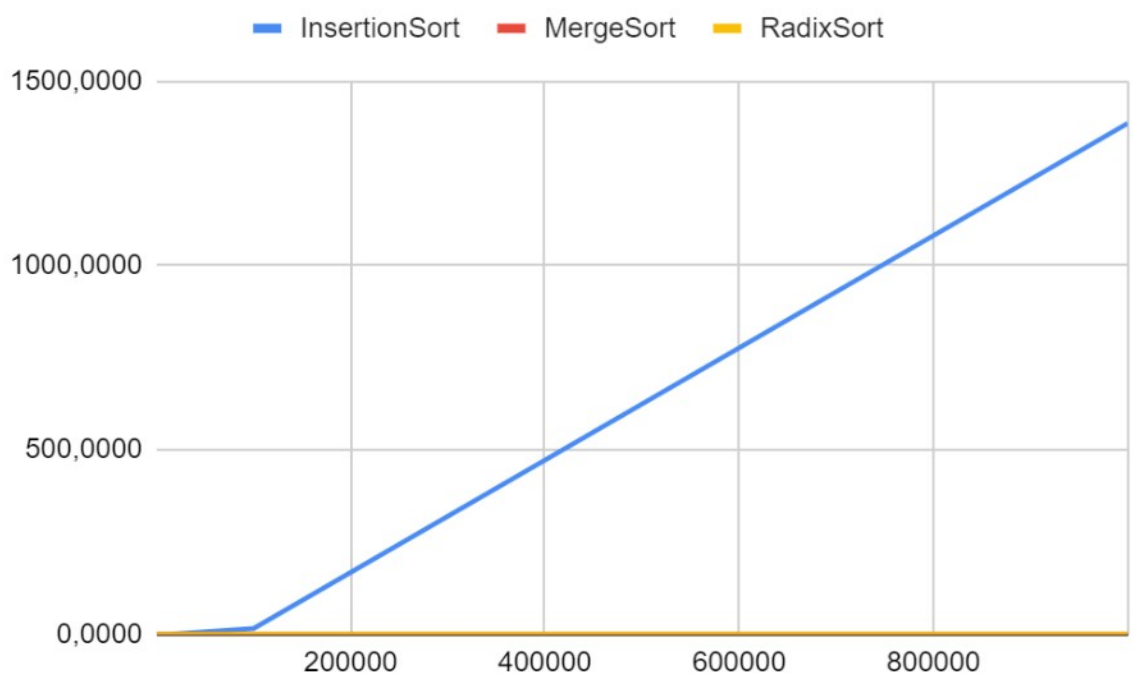
RESULTADOS

Tamanho (n)	100	1000	10000	100000	1000000
InsertionSort x MergeSort	(0.0000, 0.0000)	(-0.0037, 0.0021)	(0.1470, 0.1811)	(15.1682, 16.4990)	(1296.5949, 1472.9130)
InsertionSort x RadixSort	(0.0000, 0.0000)	(-0.0009, 0.0024)	(0.1486, 0.1795)	(15.1832, 16.5184)	(1296.6984, 1473.0360)
MergeSort x RadixSort	(0.0000, 0.0000)	(-0.0007, 0.0038)	(-0.0047, 0.0047)	(0.0114, 0.0230)	(0.0902, 0.1364)

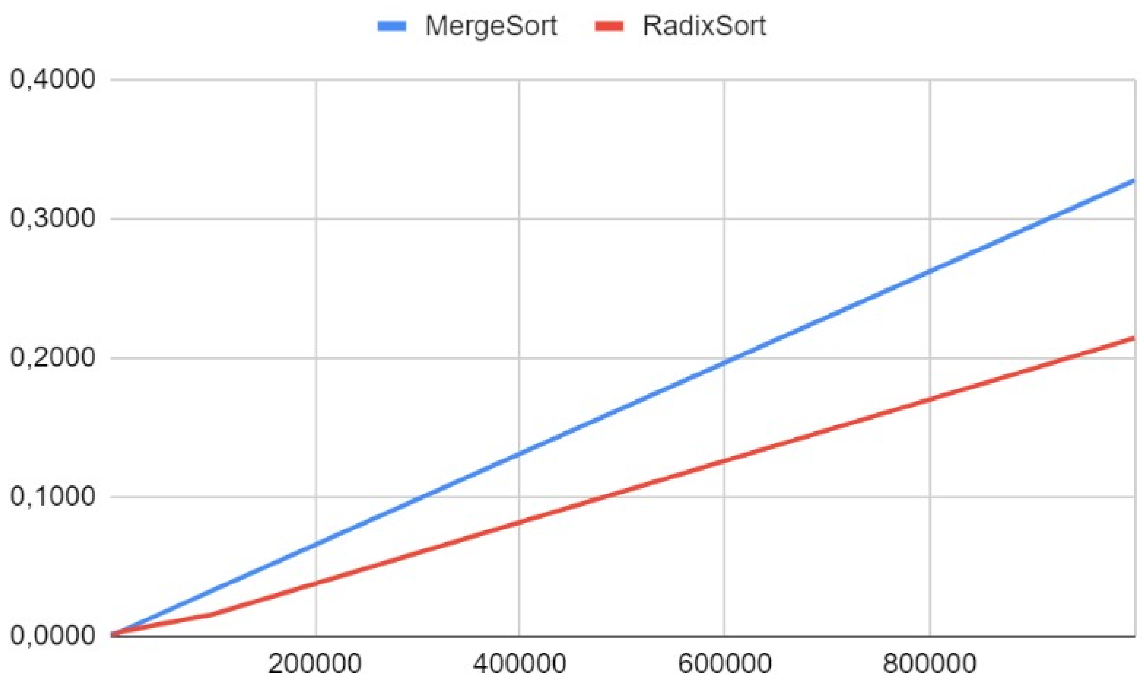
Teste estatístico t dos métodos de ordenação em relação ao tamanho dos vetores.

Legenda:
Não há diferença
Primeiro método tem desempenho pior
Primeiro método tem desempenho melhor

RESULTADOS

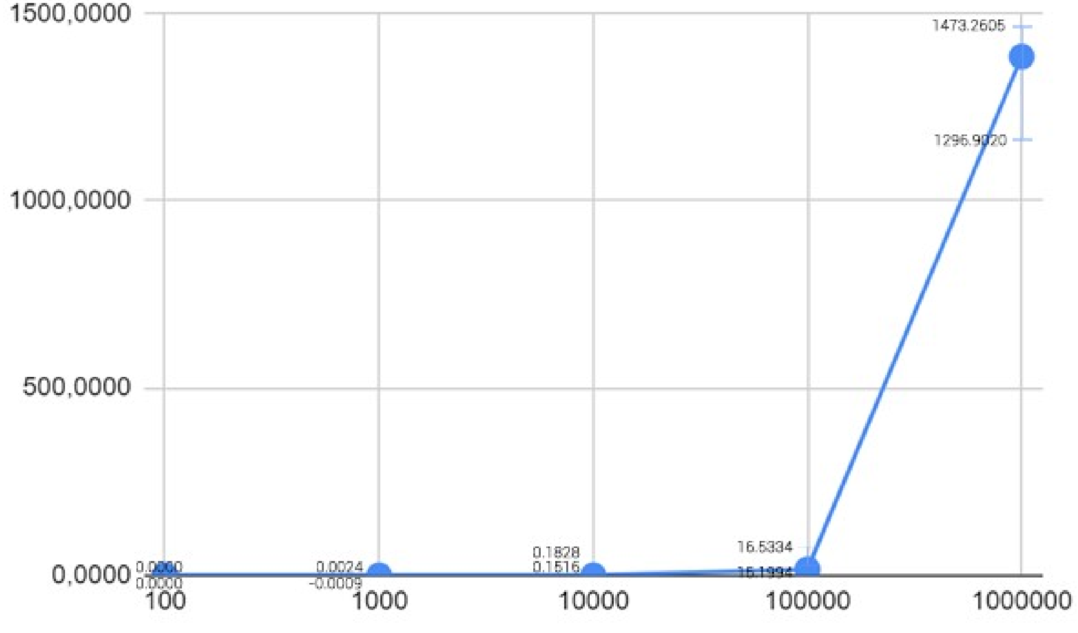


Média dos tempos para ordenação dos vetores em relação ao seu tamanho

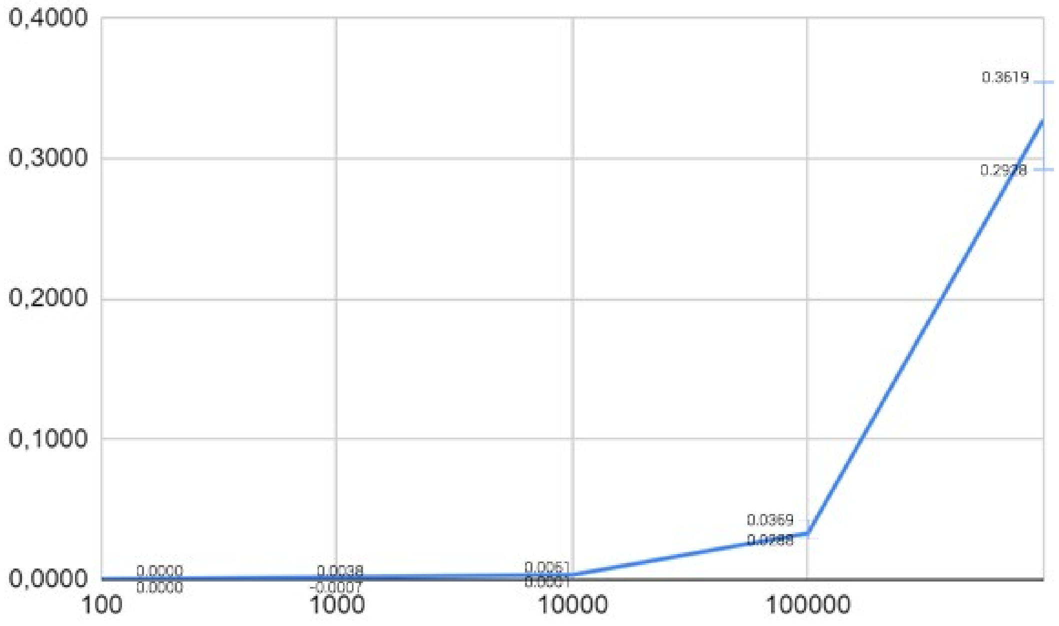


Média dos tempos para ordenação dos vetores em relação ao seu tamanho para os métodos MergeSort e RadixSort

RESULTADOS

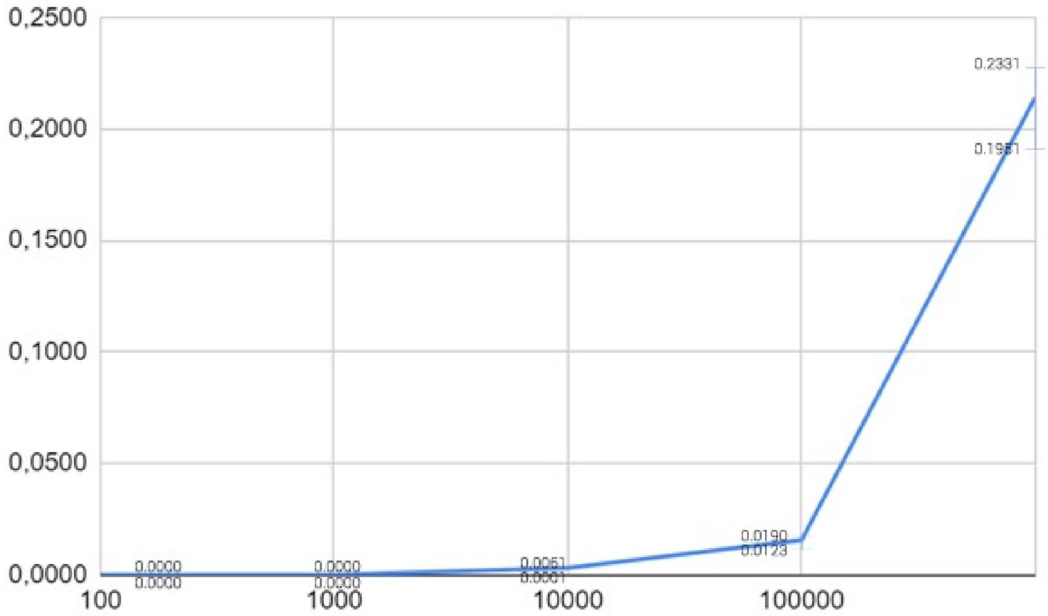


Média do tempo com o intervalo de confiança do método InsertionSort



Média do tempo com o intervalo de confiança do método MergeSort

RESULTADOS



Média do tempo com o intervalo de confiança do método RadixSort

Portanto, é possível concluir que entre os três métodos apresentados neste trabalho, o *InsertionSort* tem o pior desempenho e o *RadixSort* tem o melhor, com o *MergeSort* tendo um desempenho próximo ao do *RadixSort*, porém ainda pior que o dele.

Não obstante, verifica-se que os resultados obtidos através do teste estatístico t são comprovados pelas complexidades demonstradas na Seção 2:

Considerando que o *InsertionSort* tem complexidade $O(n^2)$, *MergeSort* $O(n * \log n)$ e o *RadixSort* $O(n)$, e que uma função n^a domina n^b se $a > b$, é evidente que o algoritmo do RadixSort terá o melhor desempenho e o InsertionSort terá o pior.

REFERÊNCIAS BIBLIOGRÁFICAS

ZIVIANI, Nivio et al. **Projeto de algoritmos: com implementações em Pascal e C**. Luton: Thomson, 2004.

SILVA, Eliezer de Souza da. **Estudo Comparativo de Algoritmos de Ordenação**. 2010.

FERREIRA, Anderson Almeida. **Aula 4: Dividir para Conquistar ou Divisão e Conquista (2.1-2.2)**. DECOM/UFOP, 2020.

M. BEDER, Delano. **Algoritmos de Ordenação: MergeSort**. 2008. Disponível em: <http://www.each.usp.br/digiampietri/ACH2002/notasdeaula/11-mergeSort.pdf>. Acesso em 18/10/2020.

WIRTH, N. **Algoritmos e estruturas de dados**. Tradução de Cheng Mei Lee. Livros técnicos e científicos, Editora S.A. Rio de Janeiro, 1999, 255p.

CHICON, Patricia Mariotto Mozzaquatro; TELOCKEN, Alex Vinícios; SCHUCH, Regis. **COMPARATIVO ENTRE OS MÉTODOS DE ORDENAÇÃO RADIXSORT E COUNTINGSORT**. XVIII Seminário Internacional de Educação no Mercosul, 2018.