Algoritmos de Ordenação

Jean Carlos Ricken, Lucas Orestes Fabris, Marcos Machado.

Universidade do Extremo Sul Catarinense, Curso Ciência da Computação

Resumo: O uso de algoritmos de inserção é essencial para que possam ser organizados. Buscamos através deste estudo analisar o uso de algoritmos de ordenação seguintes: Inserção Direta, Selection Sort, Bubble Sort, Comb Sort, Merge Sort, Heap Sort, Shell Sort, Tim sort, Quick Sort. Realizando testes com o objetivo de comparar o desempenho e a função de cada um dos algoritmos.

1.Introdução

Por meio deste artigo, será apresentado os algoritmos de Inserção Direta, Selection Sort, Bubble Sort, Comb Sort, Merge Sort, Heap Sort, Shell Sort, Tim sort e Quick Sort. Expondo o pseudocódigo (baseado na linguagem C) realizado na execução, e comparar os tempos baseados na quantidade de valores de entrada, e com as seguintes métricas: número de trocas, comparações, tempo de execução, para um vetor com tamanho 1000.

2.0 Algoritmo de ordenação

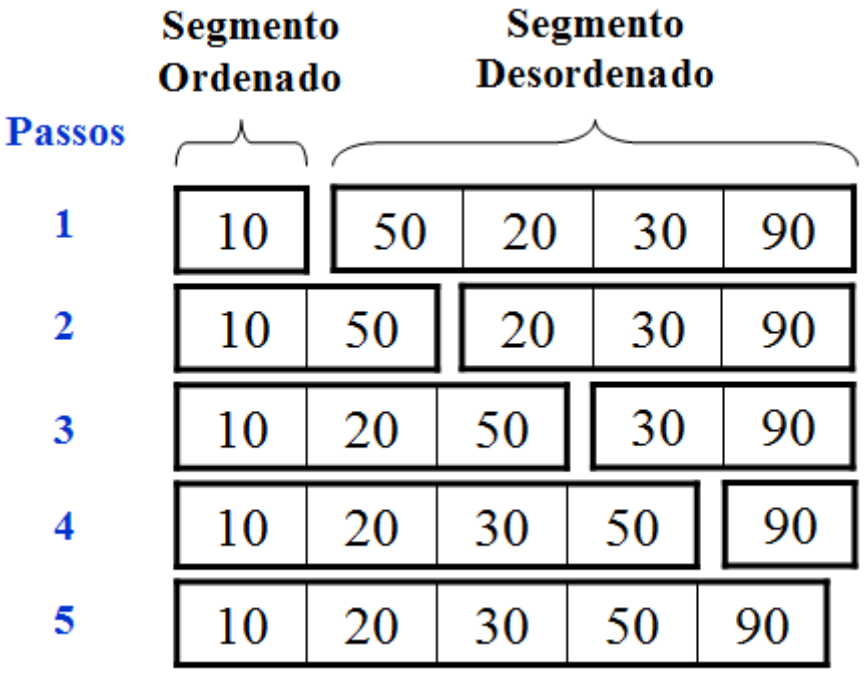
Estes algoritmos organizam os elementos baseados nas informações dos usuários, onde este pode realizar a escolha no formato de saída, para a sequência de entrada.

Ele efetua sua ordenação completa ou parcial. A ordenação tem o abjetivo de facilitar a recuperação dos dados de uma lista.

Para este estudo foram escolhidos alguns algoritmos de ordenação para serem estudados que são: Inserção Direta, Selection Sort, Bubble Sort, Comb Sort, Merge Sort, Heap Sort, Shell Sort, Tim sort, Quick Sort.

2.1 Inserção direta

É um algoritmo que varre a lista de elementos, o mesmo os organiza, um a um, em sua posição mais correta, onde o elemento a ser alocado terá, a sua esquerda um valor menor e à sua direita um valor maior.

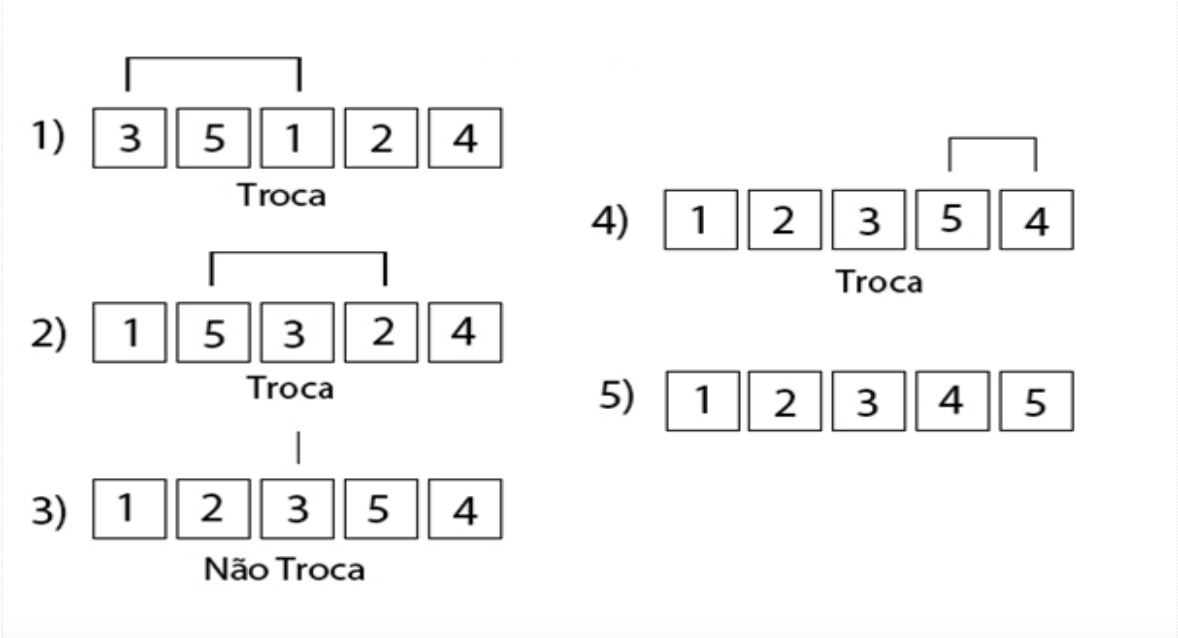


Código usado na execução do algoritmo:

```
void insertionSort(int vetor[], int beg, int end){
    int temp, i, j;
    for (i = beg + 1; i <= end; i++)
    {
        countVerify++;
        temp = vetor[i];
        j = i - 1;
        while (vetor[j] > temp && j >= beg)
        {
            countVerify++;
            vetor[j+1] = vetor[j];
            countChanges++;
            j--;
        }
        vetor[j+1] = temp;
    }
}
```

2.2 Selection sort

Algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição, depois o segundo menor valor para a segunda posição, e assim sucessivamente com os elementos restantes, até os últimos dois elementos.

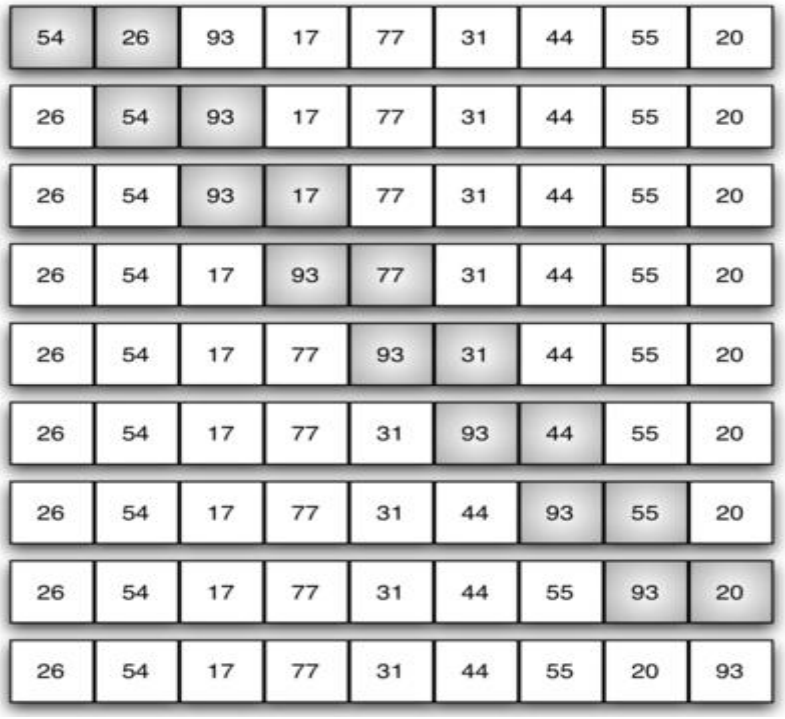


Código usado na execução do algoritmo:

```
void selectionSort(){
    int vetor[tamanho], i;
    clock_t begin = clock();
    for (i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    int j, min_idx;

    for (i = 0; i < tamanho-1; i++)
    {
        // Acha o elemento minimo no array embaralhado
        min_idx = i;
        for (j = i+1; j < tamanho; j++)
          if (vetor[j] < vetor[min_idx]){
            min_idx = j;
            countVerify++;
          }
        // Muda o elemento minimo com o primeiro do array
        muda(&vetor[min_idx], &vetor[i]);
    }
    clock_t end = clock();
    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,end);
}
```

2.3 Bubble sort

Utiliza o método onde faz a comparação entre dois elementos e troca sua posição, movendo o maior valor para a direita, realizando a passagem pelo algoritmo várias vezes até que ele o vetor esteja organizado.



Código usado na execução do algoritmo:

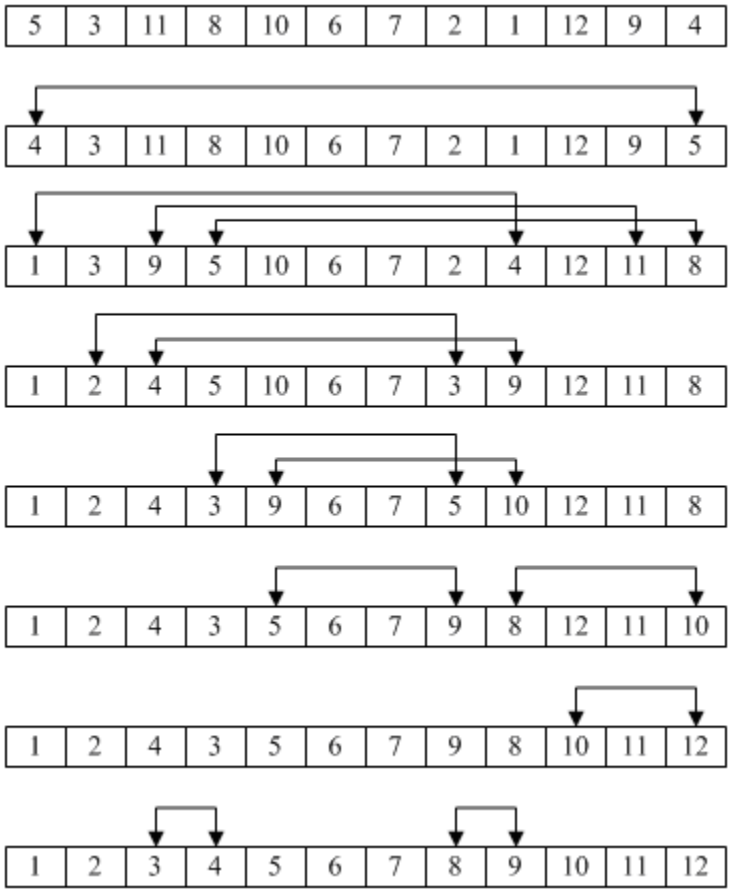
```
void bubbleSort(){
    int vetor[tamanho], i, j, temp;
    clock_t begin = clock();
    for (i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    for (i = 0; i < tamanho-1; i++)
       for (j = 0; j < tamanho-i-1; j++)
           if (vetor[j] > vetor[j+1]){
              muda(&vetor[j], &vetor[j+1]);
              countVerify++;
           }

    clock_t end = clock();
    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,end);
}
```

2.4 Comb Sort

A ideia do Comb Sort é eliminar os pequenos valores próximos do final da lista, já que eles atrapalham a classificação.

Este algoritmo é muito semelhante ao método Bubble Sort, porém ele é mais eficiente pois reordena diferentes pares de itens, separados por um salto que é calculado a cada passagem e quando quaisquer dois elementos são comparados, eles sempre têm um intervalo com 1 index de distância.

O intervalo começa como o comprimento da lista a ser ordenada, dividida pelo fator de encolhimento, e a lista é ordenada com este valor arredondando o intervalo quando for necessário. Então, a diferença é dividida pelo fator de encolhimento novamente, a lista é ordenada com este novo intervalo, e o processo se repete até que a diferença seja de 1.

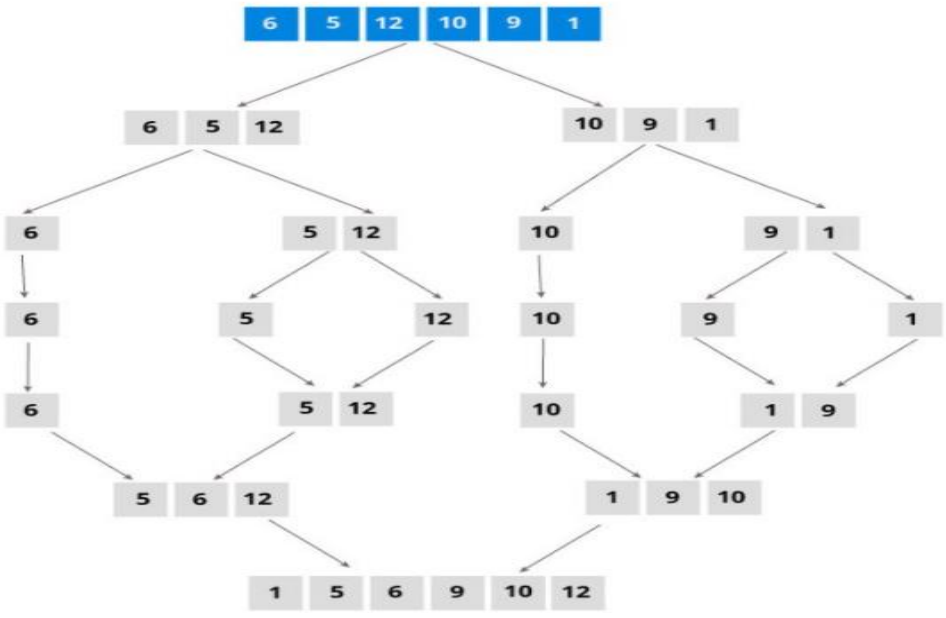


Código usado na execução do algoritmo:

```
void combSort(){
    int vetor[tamanho], i, j, temp;
    clock_t begin = clock();
    for (i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    printf("\n");

    bool swaps = true;
    int gap = tamanho;

    while (gap > 1 || swaps)
    {
        countVerify++;
        gap /= 1.247330950103979;

        if (gap < 1){
            gap = 1;
            countVerify++;
        }

        int i = 0;
        swaps = false;

        while (i + gap < tamanho)
        {
            countVerify++;
            int igap = i + (int)gap;

            if (vetor[i] > vetor[igap])
            {
                countVerify++;
                int temp = vetor[i];
                vetor[i] = vetor[igap];
                vetor[igap] = temp;
                countChanges+=2;
                swaps = true;
            }
            ++i;
        }
    }
    clock_t end = clock();
    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,end);
}
```

2.5 Merge Sort

A ideia dele é utilizar a estratégia de dividir para conquistar, dividindo em vários subproblemas e resolver estes de forma recursiva, e após resolvidos ocorre a conquista, que é a união das resoluções dos subproblemas.



Código usado na execução do algoritmo:

```
void mergeSort(){

void mergin(int vetor[], int l, int r){
    int temp, i;

    if(l < r){
        countVerify++;
        int m = l + (r - l) / 2;

        mergin(vetor, l, m);
        mergin(vetor, m + 1, r);

        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[n1], R[n2];

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

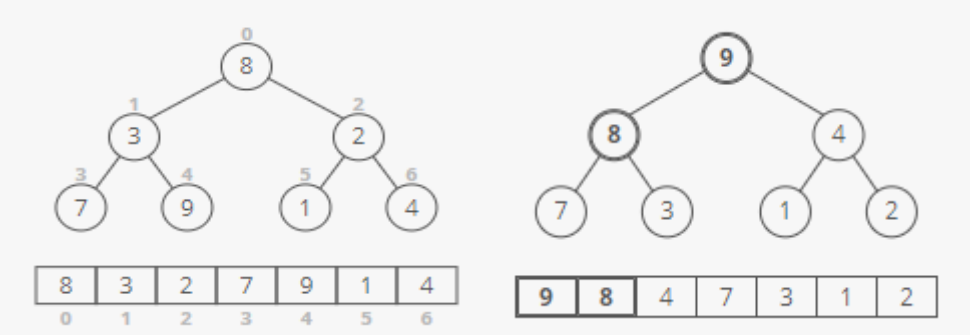
```
        i = 0;
        j = 0;
        k = l;
        while (i < n1 && j < n2) {
            countVerify++;
            if(L[i] <= R[j]) {
                countVerify++;
                vetor[k] = L[i];
                countChanges++;
                i++;
            }
            else{
         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

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

        while (j < n2) {
            countVerify++;
            vetor[k] = R[j];
            countChanges++;
            j++;
            k++;
        }
    }
}
```

2.6 Heap sort

O Heap Sort utiliza a estrutura de dados heap, para ordenar os elementos conforme são inseridos na estrutura. No final das inserções, os elementos podem ser removidos sucessivamente da sua raiz, na ordem que for desejado.

Para uma ordenação decrescente, deve ser construída uma heap mínima onde o menor elemento fica na raiz. Para uma ordenação crescente, deve ser construído uma máxima onde o maior elemento fica na raiz.



Código usado na execução do algoritmo:

```
void heapSort(){
    int vetor[tamanho], i, j, temp;
    clock_t begin = clock();
    for(i=0; i<tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    for (int i = tamanho / 2 - 1; i >= 0; i--)
        heapify(vetor, tamanho-1, i);


    for (int i = tamanho - 1; i >= 0; i--) {
        muda(&vetor[0], &vetor[i]);

        heapify(vetor, i, 0);
    }
    clock_t end = clock();
    printf("\n");
    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,end);
}
```

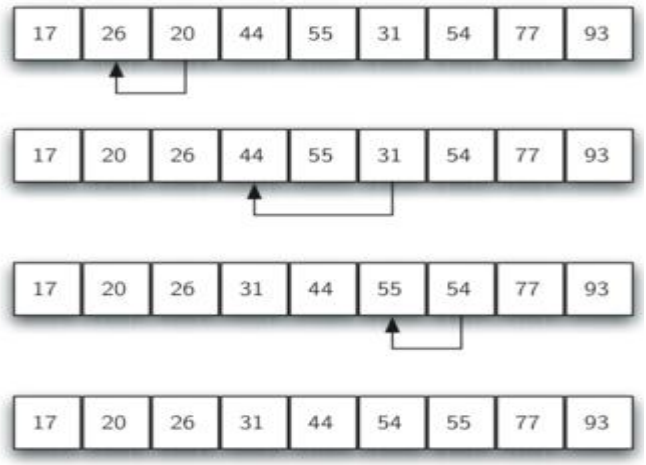
```
void heapify(int vetor[], int n, int i){
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && vetor[left] > vetor[largest]){
        countVerify++;
        largest = left;
    }

    if (right < n && vetor[right] > vetor[largest]){
        countVerify++;
        largest = right;
    }

    if (largest != i) {
      countVerify++;
      muda(&vetor[i], &vetor[largest]);
      heapify(vetor, n, largest);
    }
}
```

2.7 Shell Sort

É o algoritmo mais eficiente entre os de complexidade quadrática. É uma melhoria da inserção direta.

Ele considera o array a ser ordenado como um único segmento, ele considera vários segmentos, sendo aplicado o método de inserção direta em cada um deles.

O algoritmo passa diversas vezes pela lista, dividindo o grupo maior em menores. Nos menores grupos é aplicado o método de ordenação por inserção.



Código usado na execução do algoritmo:

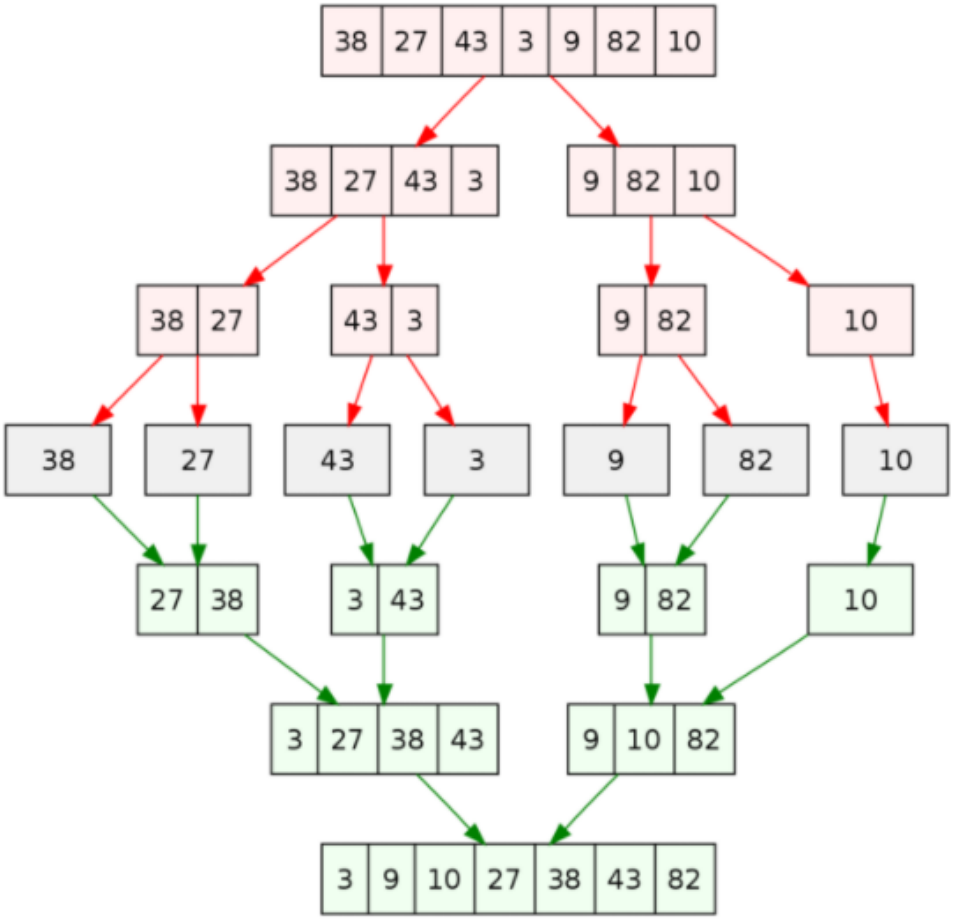
```
void shellSort(){
    int vetor[tamanho], i, j, temp;
    clock_t begin = clock();
    for (i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    printf("\n");

    for(int interval = tamanho / 2; interval > 0; interval /= 2) {
        for(int i = interval; i < tamanho; i += 1) {
            int temp = vetor[i];
            int j;
            for (j = i; j >= interval && vetor[j - interval] > temp; j -= interval) {
                vetor[j] = vetor[j - interval];
                countChanges++;
            }
            countChanges++;
            vetor[j] = temp;
        }
    }
    clock_t end = clock();
    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,end);
}
```

2.8Tim sort

Algoritmo híbrido de ordenação baseado no InsertionSort e no MergeSort. Ele se baseia na ideia de que um vetor de dados a ser ordenado contém sub-vetores já ordenados, não importando sua ordem.

O método ordena um segmento específico do vetor de entrada, incrementando da esquerda para a direita, buscando por elementos ordenados de forma consecutiva. Se esses segmentos não tiverem o tamanho necessário, eles são estendidos e ordenados utilizando o InsertionSort.

A posição de início e o tamanho gerados são armazenados em uma pilha. Durante a execução, alguns desses segmentos são combinados de acordo com condições analisadas dos elementos que estão no topo da pilha, garantindo que o comprimento dos segmentos gerados diminua e também que seja maior do que a soma dos próximos dois. Para finalizar entra o merge sort dos elementos restante, e assim temos como resultado um vetor ordenado.

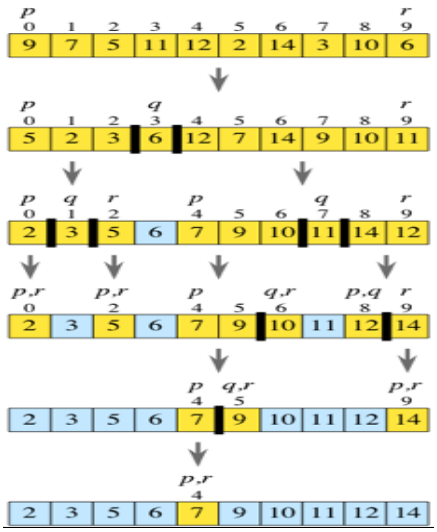


Código usado na execução do algoritmo:

```
void timSort(){
    int vetor[tamanho], i, j, temp;
    clock_t begin = clock();
    for (i = 0; i < tamanho; i++)
    {
        vetor[i] = rand() % 2000;
        printf("%d\t", vetor[i]);
    }
    int size,beg,mid,end;
    for (i = 0; i < tamanho; i+=run)
        insertionSort(vetor, i, minimo((i+31), (tamanho-1)));
    for (size = run; size < tamanho; size = 2*size)
    {
        for (beg = 0; beg < tamanho; beg += 2*size)
        {
            end = minimo((beg + 2*size - 1), (tamanho-1));

            mergin(vetor, beg, end);
        }
    }
    clock_t fim = clock();
    printf("\n");

    for(i=0;i<tamanho;i++){
        printf("%d\t", vetor[i]);
    }
    printf("\n");
    dadosExec(begin,fim);
}
```

2.9 Quick sort

Ele usa a estratégia de divisão e conquista que consiste em rearranjar as chaves de modo que as chaves menores antecedam as chaves maiores e em seguida ele ordena as duas sub listas de chaves menores e maiores de forma recursiva, até que a lista se encontre de forma ordenada. O processo não é infinito, pois para cada iteração, um elemento é posto em sua posição final e não será mais manipulado na próxima iteração.



Código usado na execução do algoritmo:

```
void quick(int vetor[],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        countVerify++;
        pivot=first;
        i=first;
        j=last;
        while(i<j){
            countVerify++;
            while(vetor[i]<=vetor[pivot]&&i<last){
                countVerify++;
                i++;
            }
            while(vetor[j]>vetor[pivot]){
                countVerify++;
                j--;
            }

            if(i<j){
                countVerify++;
                temp=vetor[i];
                vetor[i]=vetor[j];
                vetor[j]=temp;
                countChanges+=2;
            }
        }
        temp=vetor[pivot];
        vetor[pivot]=vetor[j];
        vetor[j]=temp;
        countChanges+=2;
        quick(vetor,first,j-1);
        quick(vetor,j+1,last);
    }
}
```

3.0 Resultado e discussões

A Linguagem C foi definida como meio de codificação para a realização das simulações e a IDE DevC++ V5.11 como ambiente de desenvolvimento. Já em hardware foram os testes foram feitas em uma máquina com as seguintes configurações.

- Fabricante Lenovo.
- Processador: Intel(R) Core(TM) i5-8250U CPU
- Frequência: 1.60GHz.
- Memoria Ram: 8,00 GB (DDR4 1.400MHz).
- Sistema operacional: Windows 10 Home(64 bits).

Teste em um vetor de 1000 elementos.

VETOR [1000]			
Lista	Ordem crescente		
Algoritmo	Tempo	Verificações	Mudanças
Inserção direta	0,188	249982	499964
Selection sort	0,528	5095	1998
Bubble sort	0,741	247587	495174
Comb sort	1,241	28293	8486
Merge sort	1,461	19696	19952
Heap sort	1,674	19736	18146
Shell sort	1,981	15403	15394
Tim sort	2,414	76523	87196
Quick sort	2,628	18581	4718

Conclusão

Concluímos nesse trabalho que cada método tem um jeito de realizar sua busca no vetor. Os mais sofisticados, como o Insertion sort que fez a execução em 0.18 segundos, foi o mais rápido a ser executado e também fez mais trocas, já o método Quick sort fez em 2,62 segundos, que fez um número menor de trocas que o Insertion sort.

Cada método possui sua maneira de organizar, assim como o seu tempo de execução, que varia do algoritmo que o mesmo é desenvolvido.

Referências

ZIVIANI, Nivio., **Projeto de algoritmos com implementações em JAVA e C++.** São Paulo Cengage Learning 2012 1 ISBN 9788522108213. http://www.bib.unesc.net/pergamum/biblioteca_s/acesso_login.php?cod_acervo_acessibilidade=5001540&acesso=aHR0cHM6Ly9pbnRlZ3JhZGEubWluaGFiaWJsaW90ZWNhLmNvbS5ici9ib29rcy85Nzg4NTIyMTA4MjEz&label=acesso%20restrito.

Honorato, B. (2013). "Algoritmos de ordenação: análise e comparação." http://www.devmedia.com.br/algoritmos-de-ordenacao-analise-ecomparacao/28261 , abril.

Szwarcfiter, J. L. and Markezon, L. (2015). “Estruturas de Dados e Seus Algoritmos.” 3ª edição. Rio de Janeiro. LTC.

Wikipedia, https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o , 16 de julho de 2021.

AED Algoritmos e Estruturas de Dados LEEC - 2003/2004, http://web.tecnico.ulisboa.pt/ana.freitas/AED/PDFs/SortA.pdf, 16 de julho de 2021.