UNIVERSIDADE FEDERAL DO CEARÁ

RELATÓRIO DO PROJETO 03

Dupla: João Victor de Almeida Félix
Rayrisson Vinicius Alves de Lima

Curso: Ciência da Computação

Orientador: Atílio Gomes

Quixadá - 2020

Conteúdo

1 Introdução

O objetivo do projeto é utilizar algoritmos de ordenação tanto em sua versão iterativa, quanto recursiva para mostrar qual apresenta melhor desempenho no quesito do tempo médio de execução.

O projeto foi realizado em dupla. Como foram pedidos os seguintes algoritmos: **InsertionSort**, **SelectionSort**, **MergeSort** e **QuickSort**. Um dos membros ficava com os algoritmos pares, enquanto que o outro ficava com os ímpares.

Subdividimos o projeto em 3 arquivos, sendo a main.cpp, ordenacaoVetor.cpp e ordenacaoVetor.h . Desenvolvemos o projeto usando a linguagem C++.

2 Listagem dos Programas

2.1 main.cpp

O arquivo main.cpp irá chamar os algoritmos a fim de testarmos e observar o tempo médio de execução de cada um baseado num vetor gerado com valores aleatórios 5 vezes.

2.2 ordenacaoVetor.cpp

Este é o arquivo onde iremos implementar todos os algoritmos de ordenação e métodos que geram números aleatórios com tamanhos de variados, retirados de um vetor, além de gravar esses valores em arquivos para depois serem lidos.

2.3 ordenacaoVetor.h

Contém os protótipos das funções que serão implementadas no aquivo "ordenacaoVetor.cpp" em formato estruturado.

3 Algoritmos de Ordenação

Aqui iremos mostrar como funcionam os algoritmos iterativos e recursivos.

OBS: Todos os método estão devidamente comentados no arquivo "ordenacaoVetor.cpp".

3.1 InsertionSort

No algoritmo Insertion usamos o primeiro laço para n(tamanho do vetor) interações enquanto que o segundo será utilizado para descolar á direita todos os elementos maiores que a chave atual, onde está chave é o valor do array na posição atual do laço externo. Ao final da iteração mais interna, colocamos a key na devida posição. Isso se repete até todos os valores do vetor estejam devidamente ordenados.

3.1.1 Insertion Iterativo

```
1  void insertionSort(int array[], int n){
2    int i, j, key;
3    for(j = 1; j < n; j++){
4      key = array[j];
5      i = j-1;
6      while(i >= 0 && array[i] > key){
7        array[i+1] = array[i];
8        i--;
9      }
10     array[i+1] = key;
11   }
12 }
13
```

3.1.2 Insertion Recursivo

```
1  void insertionSort_recursive(int array[], int n){
2    int key, i, j;
3    j = n;
4    if(n == 0){
5      return;
6    }
7    insertionSort_recursive(array,n-1);
8
9    key = array[j-1];
10   i = j-2;
11   while(i >= 0 && array[i] > key){
12     array[i+1] = array[i];
13     i--;
14   }
15   array[i+1] = key;
16
17 }
18
```

3.2 SelectionSort

Compara o primeiro elemento com o menor elemento do vetor que estivesse na sua frente, se fosse menor ele realizava a troca, logo após faz a mesma coisa com o segundo, depois com o terceiro e assim sucessivamente.

Na versão iterativa o menor elemento era achada por meio de um laço com um teste de caso, na versão recursiva ele era achado por meio de varias chamadas e comparavam com o elemento da chamada anterior.

3.2.1 Selection Iterativo

```
1  void selectionSort(int array[], int n){
2    for(int i = 0; i < n - 1; i++){
3      int min = i;
4      for(int j = i + 1; j < n; j++){
5        if(array[j] < array[min]){
6          min = j;
7        }
8      }
9      std::swap(array[i], array[min]);
10   }
11 }
12
```

3.2.2 Selection Recursivo

```
1  void selectionSortRecursive(int array[], int n, int i){
2    if(i == n - 1){
3      return;
4    }else{
5      std::swap(array[min(array, i + 1, n, i)], array[i]);
6      selectionSortRecursive(array, n, i + 1);
7    }
8  }
9
```

3.3 MergeSort

No MergeSort utilizando uma função auxiliar chamada intercala que recebe como parâmetros o vetor as posições inicial, a do meio do vetor e a final. No algoritmo iterativo, inicialmente iremos intercalar apenas dois elementos por vez enquanto que a posição final não seja superior ao tamanho do vetor. Em seguida iremos duplicar a quantidade de elementos que irã ser intercalados. Tais operações irão ocorrer enquanto forem menores que o tamanho do vetor.

O algoritmo recursivo possui uma ideia similar já que irá empilhar os dados até que ele possa ordenar apenas dois elementos e em seguida o dobro até que estejam todos ordenados.

3.3.1 Merge Iterativo

```
void mergeSort_interative(int array[], int r){
  int inicio, meio, fim;
  int novo_meio;
  for(meio = 1; meio < r; meio *= 2){
    inicio = 0;
    for(inicio = 0; inicio+meio < r; inicio += 2*meio){
      fim = inicio + (2*meio) - 1;
      novo_meio = (fim+inicio) / 2;
      if(fim > r){
        fim = r;
      }
      intercala(array,inicio,novo_meio,fim);
    }
  }
}

```

3.3.2 Merge Recursivo

```
void mergeSort(int A[] , int p , int r) {
  if(p < r) {
    int q = (p + r) / 2;
    mergeSort (A , p , q);
    mergeSort (A , q+1, r);
    intercala (A, p, q, r);
  }
}

```

3.4 QuickSort

O algoritmo executa enquanto ainda existem elementos a serem comparados, ele chama a função separa que usa o último elemento como pivot e vai colocando todos os elementos menores que ele no começo do vetor e por ultimo move o pivot para a posição seguinte ao último elemento movido.

Na versão recursiva ele fica fazendo a chamada enquanto a posição do elemento inicial é menor que a posição do elemento final, ou seja, quando existem mais de um elementos. A versão iterativa segue a mesma lógica, só que usamos laço e pilha ao invés da chamada recursiva

3.4.1 Quick Iterativo

```
void quickSortIterativo(int array[], int p, int r){
  std::stack<int> pilha;
  pilha.push(p);
  pilha.push(r);
  while(!pilha.empty()){
```

```
      r = pilha.top();
      pilha.pop();
      p = pilha.top();
      pilha.pop();
      int indice = separa(array, p, r);
      if(indice - 1 > p){
        pilha.push(p);
        pilha.push(indice - 1);
      }
      if(indice + 1 < r){
        pilha.push(indice + 1);
        pilha.push(r);
      }
    }
  }

```

3.4.2 Quick Recursivo

```
  void quickSortRecursive(int array[], int p, int r){
    if(p < r){
      int i = separa(array, p, r);
      quickSortRecursive(array, p, i - 1);
      quickSortRecursive(array, i + 1, r);
    }
  }

```

4 Gráficos

Aqui iremos mostrar o desempenho de cada algoritmo separadamente e em seguida um gráfico comparando o desempenho entre a versão iterativa e recursiva dos algoritmos de ordenação. Nas últimas seções irá conter os gráficos que compara os algoritmos iterativos e recursivos do **InsertionSort**, **SelectionSort**, **MergeSort** e **QuickSort**.

Ao observar os gráficos podemos perceber que o algoritmo iterativo irá apresentar um melhor desempenho inicialmente do que o recursivo e no momento em que o vetor passa a ter um tamanho maior é o recursivo que apresenta melhor desempenho. Haverá certos gráficos em que o desempenho de ambos pode apresentar uma diferença mínima.

4.1 InsertionSort

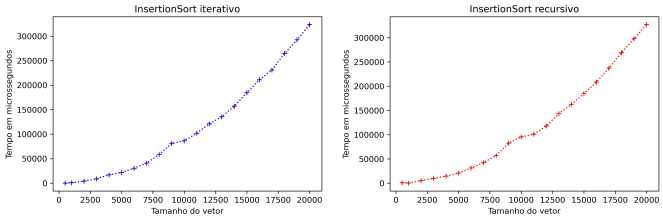

Figura 1: Gráficos do Insertion

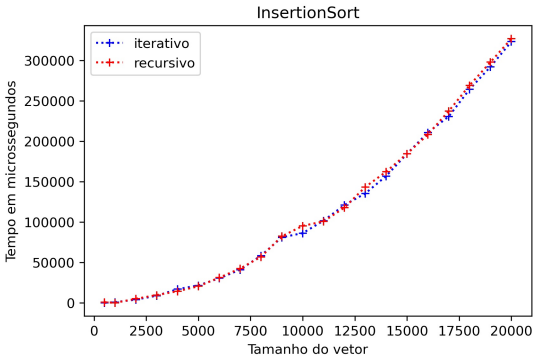

Figura 2: Desempenho do InsertionSort

4.2 SelectionSort

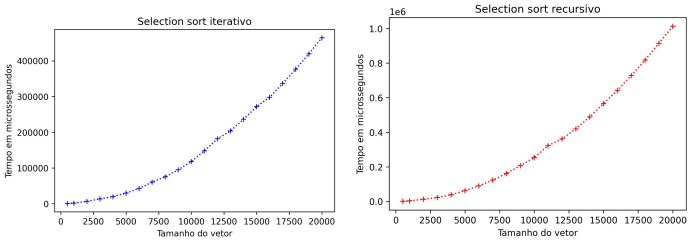

Figura 3: Gráficos do Selection

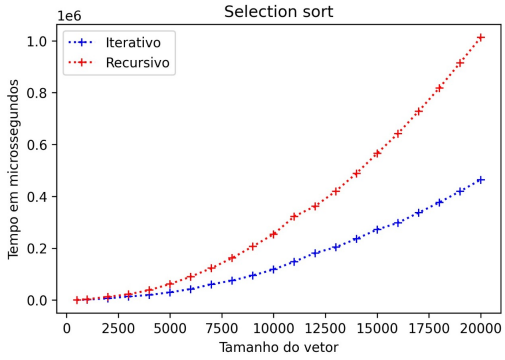

Figura 4: Desempenho do SelectionSort

4.3 MergeSort

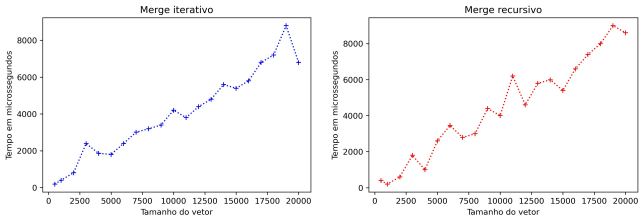

Figura 5: Gráficos do Merge

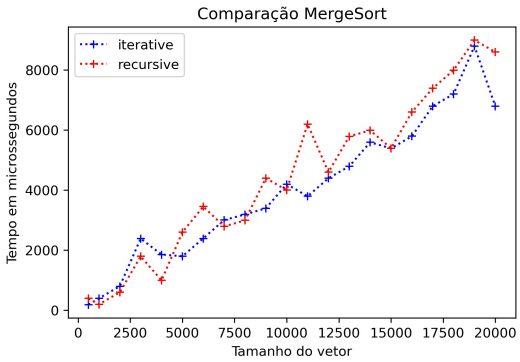

Figura 6: Desempenho do MergeSort

4.4 QuickSort

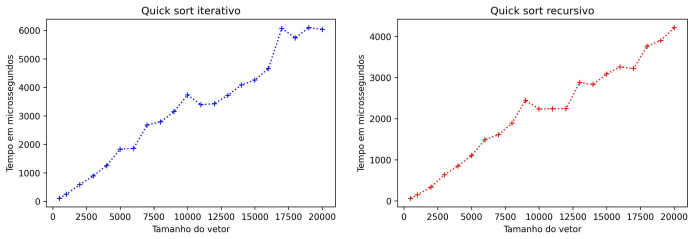

Figura 7: Gráficos do Quick

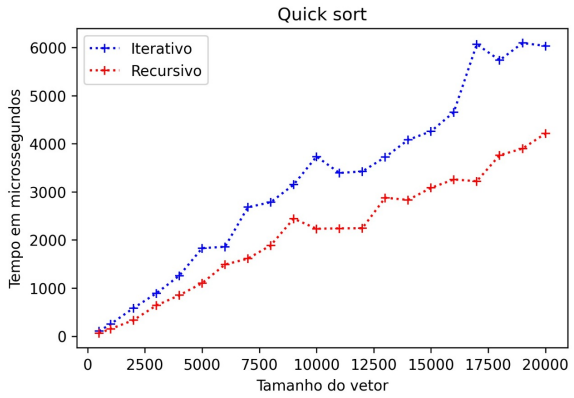

Figura 8: Desempenho do QuickSort

4.5 Algoritmos Iterativos

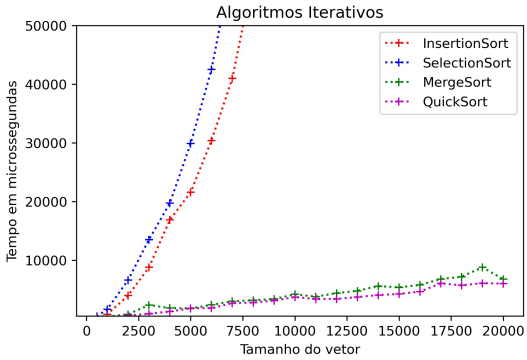

Figura 9: Desempenho dos algoritmos iterativos

4.6 Algoritmos Recursivos

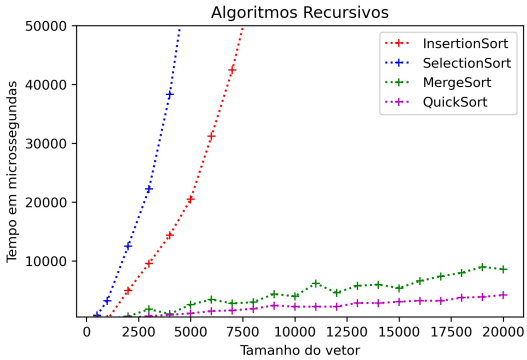

Figura 10: Desempenho dos algoritmos recursivos

5 Complexidade do Algoritmo

Todos os algoritmos foram avaliados no caso médio.

5.1 InsertionSort

O algoritmo Inserition apresenta complexidade $O(n^2)$.

5.2 SelectionSort

O algoritmo Selection apresenta complexidade $O(n^2)$.

5.3 MergeSort

O algoritmo Merge apresenta complexidade O(nlg n)

5.4 QuickSort

O Quick iterativo apresenta complexidade O(nlg n)

6 Referências

Feofiloff, P.Algoritmos em Linguagem C: 5º.ed. Link: `https://b-ok.lat/book/2281834/e16397`

GeeksforGeeks. Link: `https://www.geeksforgeeks.org/`

7 Dificuldade Apresentadas

A maior de dificuldade apresentada foi a de desenvolver a parte do algoritmo que faltava, tanto ele sendo recursivo quanto iterativo. Encontrar uma solução que apresentasse um vetor devidamente ordenado não foi a principal dificuldade e sim de um algoritmo que tivesse um melhor desempenho. Outra dificuldade que tivemos foi em encontrar a complexidade do algoritmo em questão.

8 Conclusão

Por meio de debates via google meet a fim de ajudar um ao outro com problemas na desenvolvimento dos algoritmo podemos compreender melhor como cada um funciona e de como desenvolver um que apresente o tempo de execução satisfatório, ou seja, o de melhor desempenho.