UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INSTITUTO MÉTROPOLE DIGITAL

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

DIM 0119 – ESTRUTURAS DE DADOS BÁSICAS I

FERNANDO FERREIRA DE LIMA FILHO

VÍTOR HUGO ARAÚJO PINTO

ANÁLISE EMPÍRICA COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

NATAL/RN

2020

FERNANDO FERREIRA DE LIMA FILHO

VÍTOR HUGO ARAÚJO PINTO

ANÁLISE EMPÍRICA COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

Trabalho realizado para a Universidade Federal do Rio Grande do Norte, exigência como parte da avaliação da disciplina de Estruturas de Dados Básicos I.

Docente: Prof. Dr. Selan Rodrigues dos Santos

NATAL/RN

2020

SUMÁRIO

1. INTRODUÇÃO

Sedgewick e Wayne, em Algorithms 4ª edição, descrevem algoritmos e sua importância como sendo métodos para a resolução de problemas que melhor se adequam a implementação computacional. Desta forma, algoritmos estão intimamente ligados a estruturas de dados, então os autores concluem que as estruras de dados são esquemas com a finalidade de organizar dados, deixando-os próprios para o processamento de um algoritmo. Por conseguinte, fica claro que na produção de um algoritmo, resolver o problema não é mais suficiente, é necessário se preocupar com a eficiência da resolução do problema.

A partir dessa definição, como no trabalho anterior da mesma matéria, vamos estudar a etapa seguinte aos algoritmos de pesquisa, estudo de algoritmos de ordenação. Os sete algortimos experimentados foram:

- insertion sort;
- selection sort;
- bubble sort;
- shell sort;
- quick sort;
- radix sort.

O trabalho tem como objetivo gerar dados e, a partir destes, gráficos (tamanho máximo do array versus tempo gasto para ordenação) em 6 cenários distintos, que são:

1 - arranjos com elementos em ordem não decrescente;

2 - arranjos com elementos em ordem não crescente;

3 – arranjos com elementos 100% aleatórios

4 – arranjos com 75% de seus elementos em sua posição definitiva;

5 - arranjos com 25% de seus elementos em sua posição definitiva;

6 - arranjos com 50% de seus elementos em sua posição definitiva.

2. METODOLOGIA

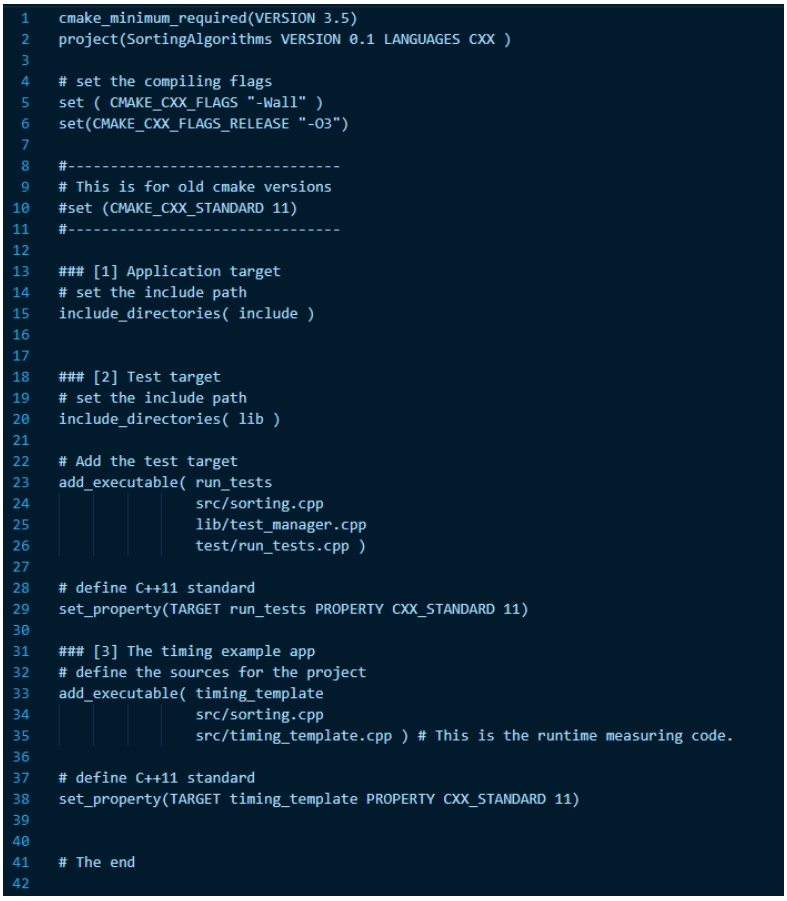
2.1. COMPUTADOR

O equipamento utilizado para obtenção das análises tem as seguintes especificações: Processador AMD Ryzen 5 3600X 3.79GHz (6-Core Processor), 1x8GB de memória RAM, Placa de Vídeo Radeon RX 580 Series, SSD XPG SX8100 512GB, M.2, Leitura 3500MB/s, Gravação 1900MB/s. O sistema operacional utilizado foi o Linux Mint 20 (Ulyana).

2.2. FERRAMENTAS

Todos os algoritmos utilizados para esse relatório foram codificados em c++ e compilados em c++ 11 através do arquivo make gerado pelo cmake (Figura 1) e as configurações utilizadas pelos autores, o passa a passo encontra-se no arquivo README.md do repositório (https://projetos.imd.ufrn.br/fernandoff/edb1-empirical-analysis).

Figura 1: configuração do cmakefile.txt



```
cmake_minimum_required(VERSION 3.5)
project(SortingAlgorithms VERSION 0.1 LANGUAGES CXX )

# set the compiling flags
set ( CMAKE_CXX_FLAGS "-Wall" )
set(CMAKE_CXX_FLAGS_RELEASE "-O3")

#--------------------------------
# This is for old cmake versions
#set (CMAKE_CXX_STANDARD 11)
#--------------------------------

### [1] Application target
# set the include path
include_directories( include )


### [2] Test target
# set the include path
include_directories( lib )

# Add the test target
add_executable( run_tests
                src/sorting.cpp
                lib/test_manager.cpp
                test/run_tests.cpp )

# define C++11 standard
set_property(TARGET run_tests PROPERTY CXX_STANDARD 11)

### [3] The timing example app
# define the sources for the project
add_executable( timing_template
                src/sorting.cpp
                src/timing_template.cpp ) # This is the runtime measuring code.

# define C++11 standard
set_property(TARGET timing_template PROPERTY CXX_STANDARD 11)


# The end

```

E para a geração dos gráficos foi utilizado o gnuplot, como orientado pelo docente do curso. O gnuplot é um programa de linha de comando que pode plotar os

gráficos de funções matemáticas em duas ou três dimensões, e outros conjuntos de dados. O programa pode ser executado na grande maioria dos computadores e sistemas operacionais. Ele é um programa com uma longa história, datando de antes de 1986 (homepage do gnuplot.org – data out/20).

Para a geração dos arranjos foi utilizado os métodos indicados pelo docente nas notas de orientação do trabalho, seguindo a o seguinte passo a passo como base (texto retirado das orientações do trabalho do docente):

Para gerar um vetor com, digamos 75%, dos elemento em sua ordem definitiva você podeproceder da seguinte forma (sugestão):

1) Comece com o arranjo A de dados já em ordem não decrescente.

2) Crie um arranjo I com valores de 0 até n−1, onde n é o tamanho do arranjo A.

3) Embaralhe (shuffle) os elementos de I.

4) Como desejamos, no exemplo75% dos elementos na ordem correta, precisamos modificar apenas 25% dos elementos originais.

Faça um laço de i = 0 até p = d(0.25∗ n) , com i sendo incrementado de 2 em 2, trocando A[I [i]] com A[I [i+ 1]].

3. ALGORITMOS

Aqui listaremos os algoritmos utilizados para o trabalho em código c++ com suas respectivas descrições retiradas do site: https://www.geeksforgeeks.org/sorting-algorithms/.

Insertion sort: um simples algoritmo de ordenação o qual funciona de forma similar à forma como ordenamos cartas de baralho na mão. O *array* é virtualmente dividido em uma parte ordenada e outra não, valores da parte não ordenada são escolhidos e colocados na ordem correta na região ordenada. Segue na Figura 2 o código utilizado.

Figura 2: código em c++ para Insertion Sort

```cpp
void insertion( value_type * first, value_type * last)
{
    long int size { last - first};
    for (long int i{1}; i < size; i++)
    {
        value_type key = *(first+i);
        long int j{i-1};
        for (; j>= 0 and *(first+j)> key; j --)
        {
            *(first+j+1) = *(first+j);
        }
        *(first+j+1) = key;
    }


}
```

Selection sort: o selection sort ordena um vetor achando repetidamente o menor elemento (considerando ordem ascendente) de uma região não ordenada e colocando no começo, mantendo duas regiões de *subarrays* uma já ordenada e o restante a ser ordenado. Em cada iteração do algoritmo o menor elemento é encontrado e posto para o próximo elemento da região ordenada, vide Figura 3 com o código.

Figura 3: código em c++ para Selection Sort

```cpp
void selection( value_type * first, value_type * last)
{
    for ( auto i = first; i != last - 1; i++) // this first loop go through all elements
    {
        auto min{i}; //the min variable will receive the address and value of each value in the array
        *min = *i;

        for ( auto j = i + 1; j != last; j++) //this second loop gonna go through the array skipping the first one (we assumed this is the lower)
        {
            //if ( cmp( *j, *min ) ) // using the compare function if the value j (remainder of the array) is less than the min
            if ( *j < *min )
            {
                min = j; //the min receveis j (possible min)
            }
        }//after loop the last min is the lower value of the remainder of the arry

        if (min != i) //if min == i means the element i is the lower in the ramainder array
            std::iter_swap(min, i); //now we swap the min with the first element
    }
}
```

Bubble sort: é dito como o mais simples dos algortimos de ordenação que funciona repetidamente trocando os elementos adjacentes se estes estão fora da condição de ordenação, a Figura 4 mostra o código utilizado.

Figura 4: código em c++ para Bubble Sort

```
void bubble( value_type * first, value_type * last)
{
    for ( auto i = first; i != last; i++ ) //go through all elements
    {
        for ( auto j = i + 1; j != last; j++) //compares the element i with all remainders
        {
            //if ( cmp(*j, *i) ) //if the value with the lowset postion in the array is greater swap them
            if (*j < *i)
            {
                std::swap( *i, *j );
            }
        }
    }
}
```

Shell sort: é primordialmente uma variação da *Insertion Sort,* a diferença é que no Insertion o elemento é movido apenas uma posição por vez e quando um elemento precisa ser movido varias posições isso é feito um movimento por vez, já no Shell sort o “avanço” do elemento é determinado pelo seu tamanho, ou seja, o elemento caminha, primeiramente, com uma distância que é metade do tamanho, isso para todos os elementos do meio até o fim, e isso se repete reduzindo essa distância pela metade até chegar em 1. A Figura 5 mostra o código utilizado.

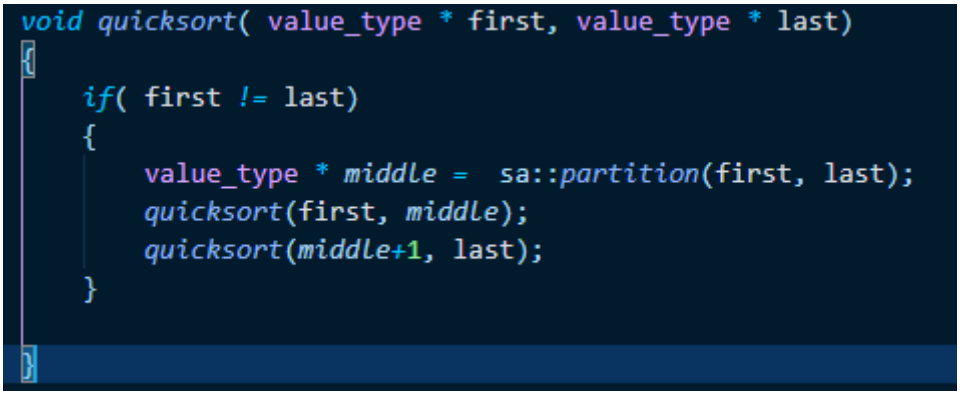
Figura 5: código em c++ para Bubble Sort

```
void shell( value_type * first, value_type * last)
{
    value_type aux;
    for (value_type gap = (last-first)/2;  gap > 0; gap/=2)
    {
        for (value_type i = gap; i < (last-first); i++)
        {

            aux = *(first+i);
            value_type j = i;
            while(j >= gap and *(first+(j-gap)) > aux )
            {
                *(first+j) = *(first+(j-gap));
                j-=gap;
            }
            *(first+j) = aux;
        }
    }
}
```

Quick sort: um algoritmo do tipo dividir e conquistar. Este escolhe um elemento como ponto de rotação e particiona o resto do vetor ao redor do ponto de rotação.

Figura 6: código em c++ para Quick Sort



Radix sort: A ideia do Radix Sort é fazer a classificação dígito por dígito, começando do dígito menos significativo ao dígito mais significativo. A classificação Radix usa a classificação por contagem como uma sub-rotina para classificar.

Figura 7: código em c++ para Radix Sort

```cpp
void radix(value_type *first, value_type *last)
{
    value_type  max { *first };
    for (short i{1}; i < (last-first); i++)
    {
        // std::cout << *(first+i) << std::endl;
        if(*(first+i) > max)
            max = *(first+i);
    }

    for (value_type exp = 1; (max/exp) > 0 and exp > 0; exp*=10)
    {
        value_type output[last-first];
        value_type count[10]{0,0,0,0,0,0,0,0,0,0};
        value_type i;
        for ( i = 0; i < last - first; i++)
        {
            count[(*(first+i)/exp)%10]++;
        }

        for ( i = 1; i < 10; i++)
        {
            count[i] += count[i -1];
        }
        for (value_type i = last - first -1; i >= 0; i--)
        {
            if(i > last - first)
                break;
            int index= (*(first+i)/ exp)%10 ;
            output[count[index] -1] = *(first+i);
            count[index] --;
        }

        for ( i = 0; i < last-first; i++)
        {
            *(first+i) = output[i];
        }
    }
}
```

4. RESULTADOS E GRÁFICOS

Aqui apresentaremos os gráficos gerados pelo gnuplot, optamos por separar aqueles de mesma complexidade juntos e dentro dos de mesma complexidade, os que foram mais lineares criamos um a parte.

Cenário 1:

Figura 8: gráfico do cenário 1 para insertion, quick, radix e shell.

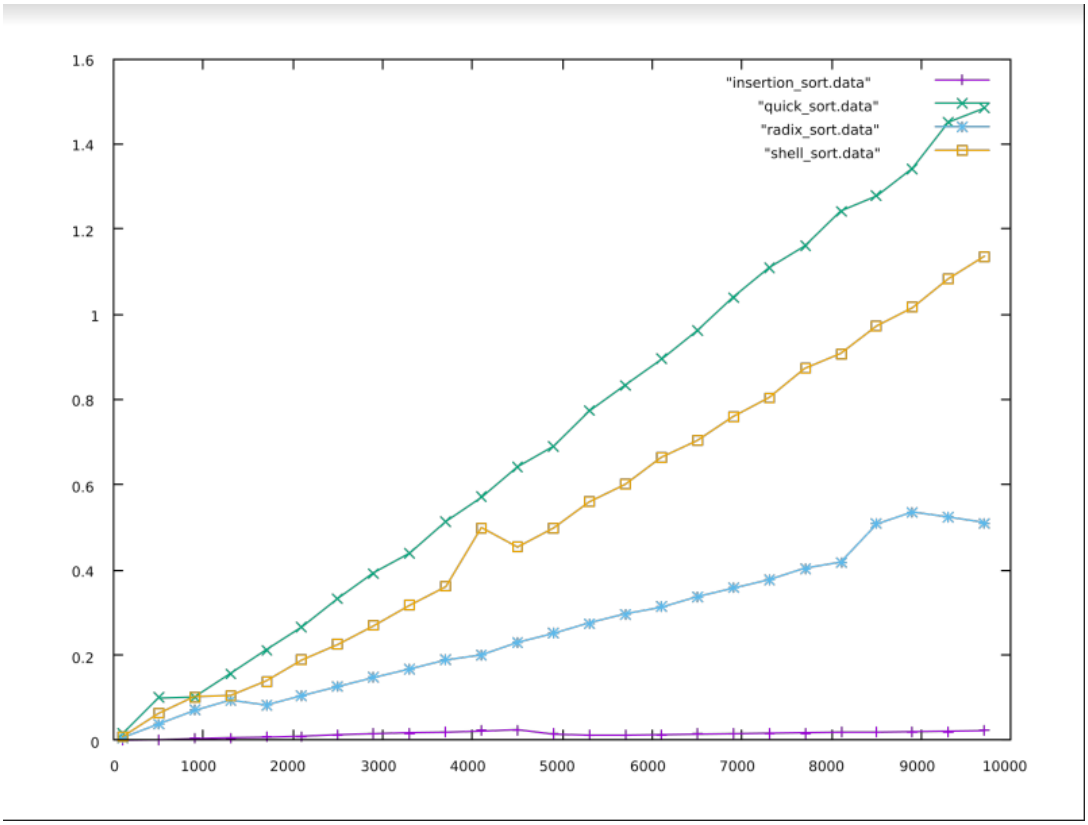


Figura 9: gráfico do cenário 1 merge

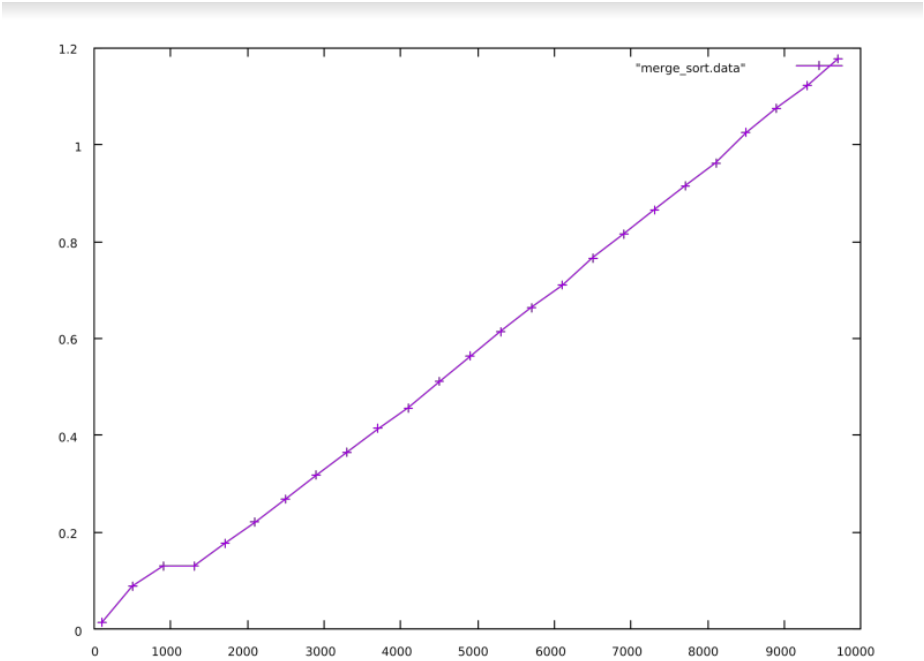
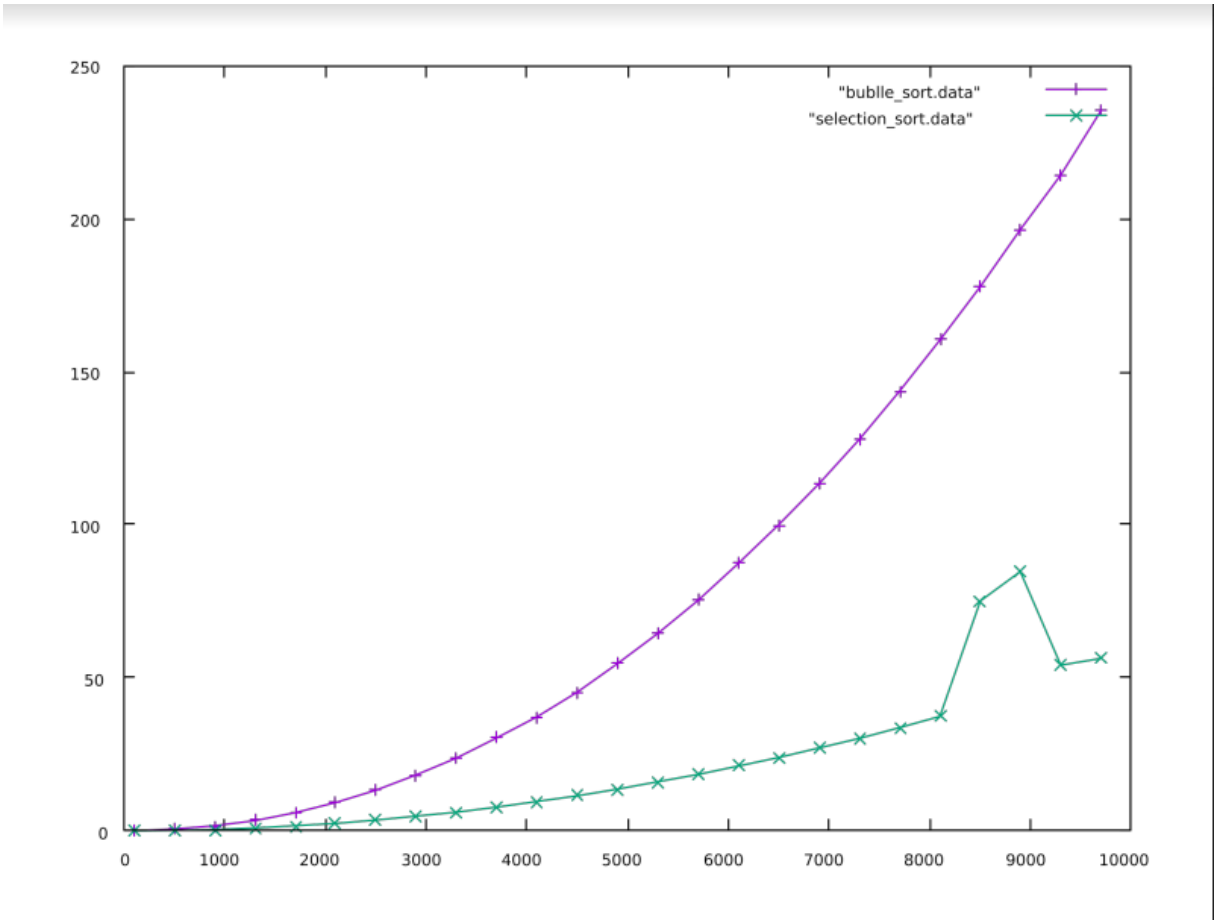


Figura 10: gráfico do cenário 1 bubble e selection



Cenário 2:

Figura 11: gráfico do cenário 2 para insertion, quick, radix e shell

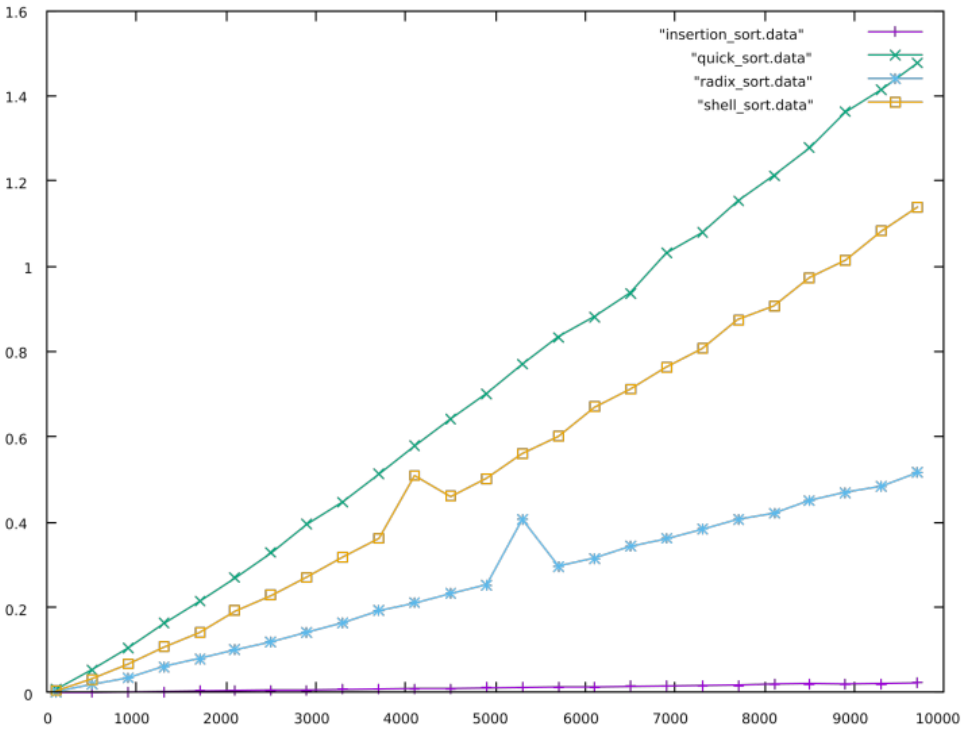


Figura 12: gráfico do cenário 2 merge

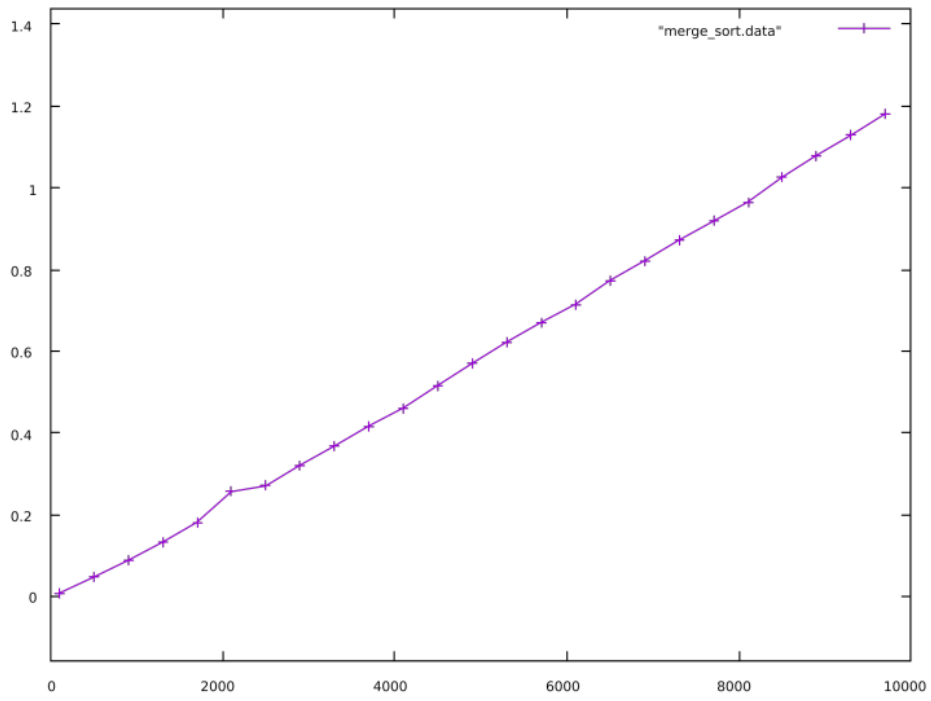
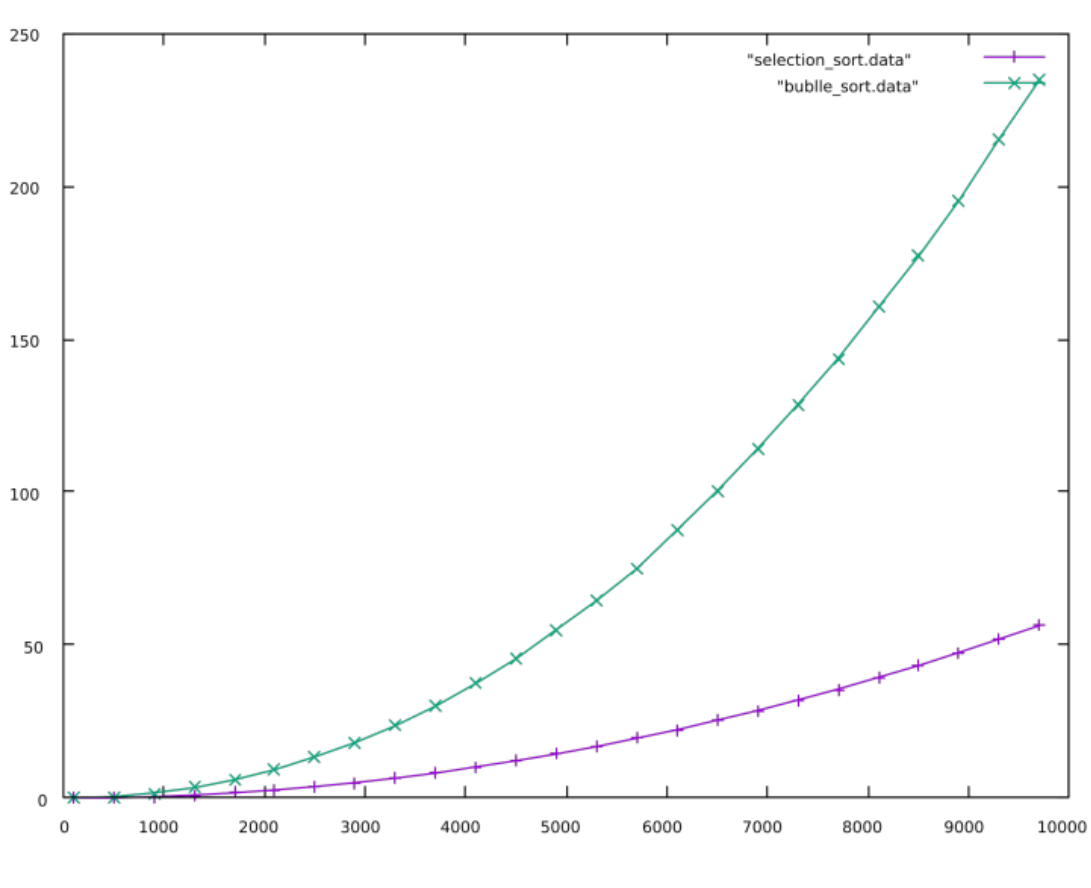


Figura 13: gráfico do cenário 2 bubble e selection



Cenário 3:

Figura 14: gráfico do cenário 3 para insertion, quick, radix e shell.

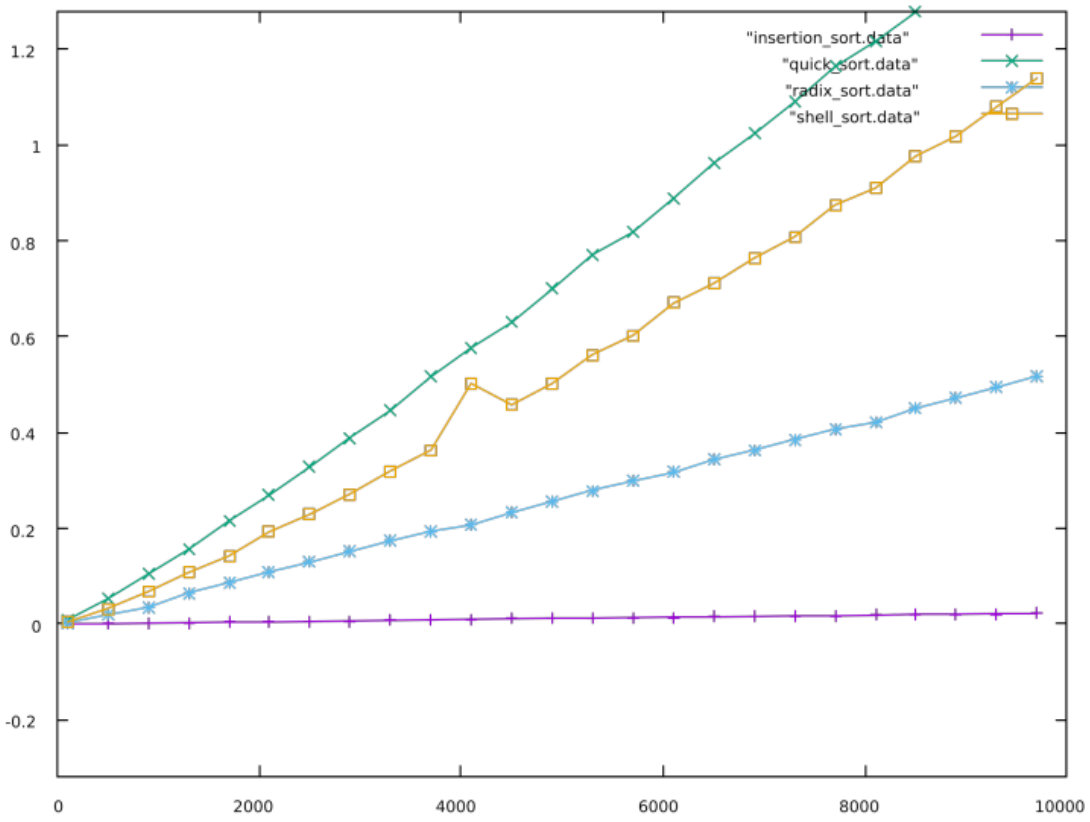


Figura 15: gráfico do cenário 3 merge

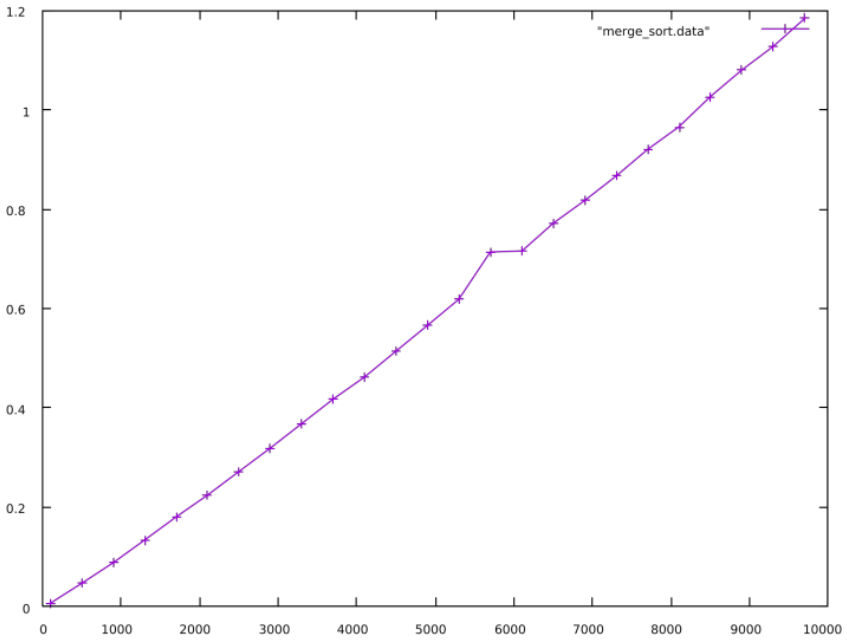
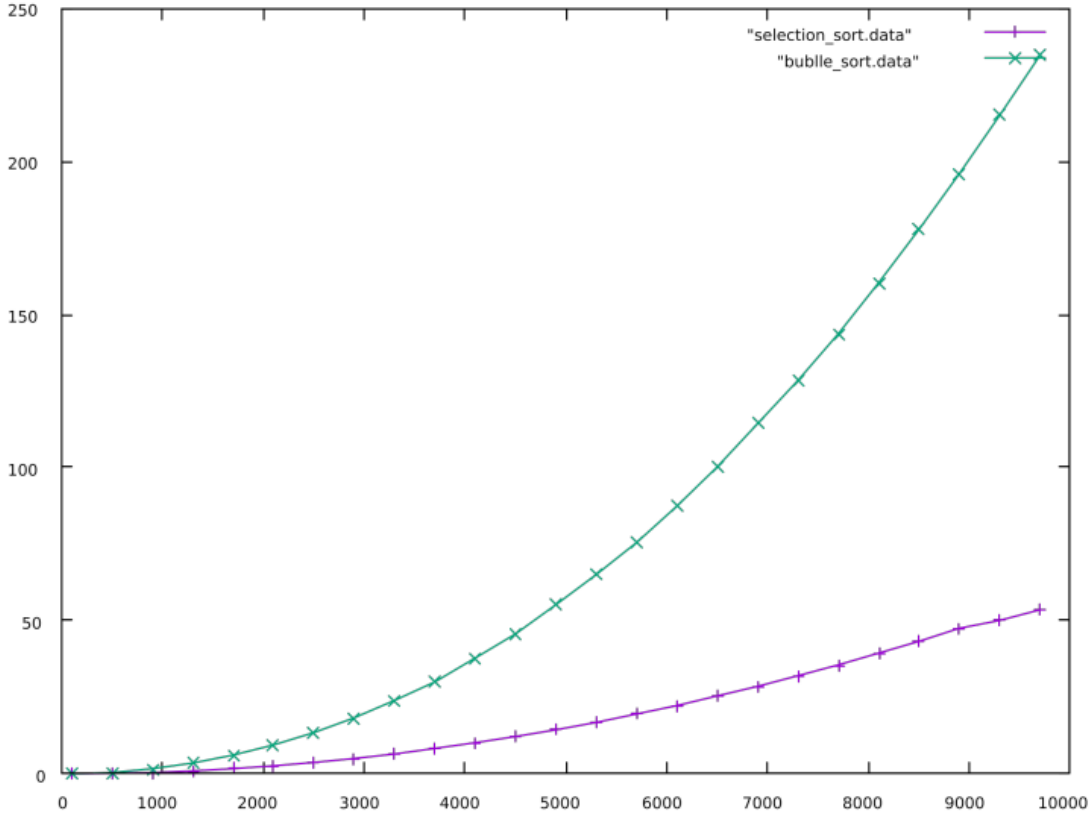


Figura 16: gráfico do cenário 3 bubble e selection



Cenário 4:

Figura 17: gráfico do cenário 4 para insertion, quick, radix e shell.

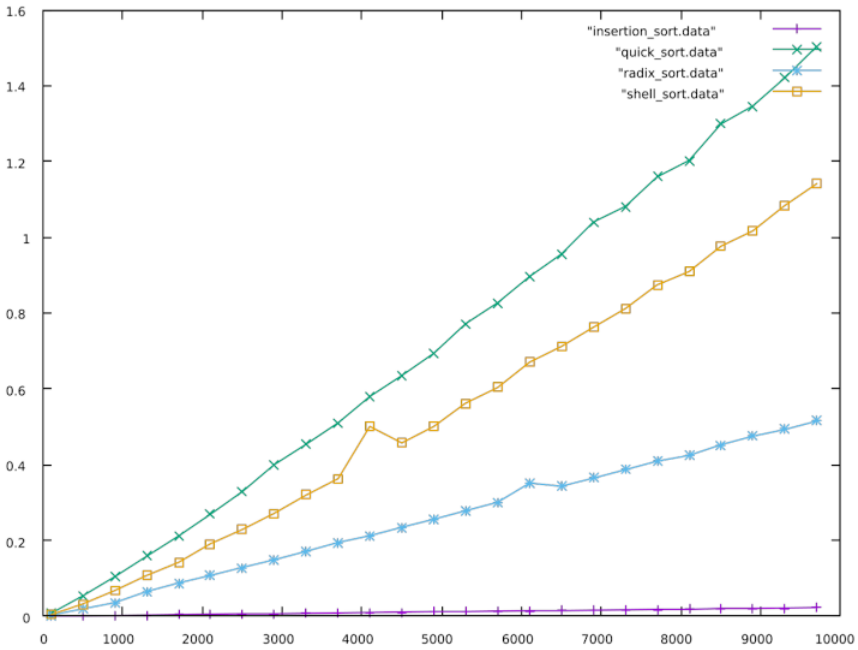


Figura 18: gráfico do cenário 4 merge

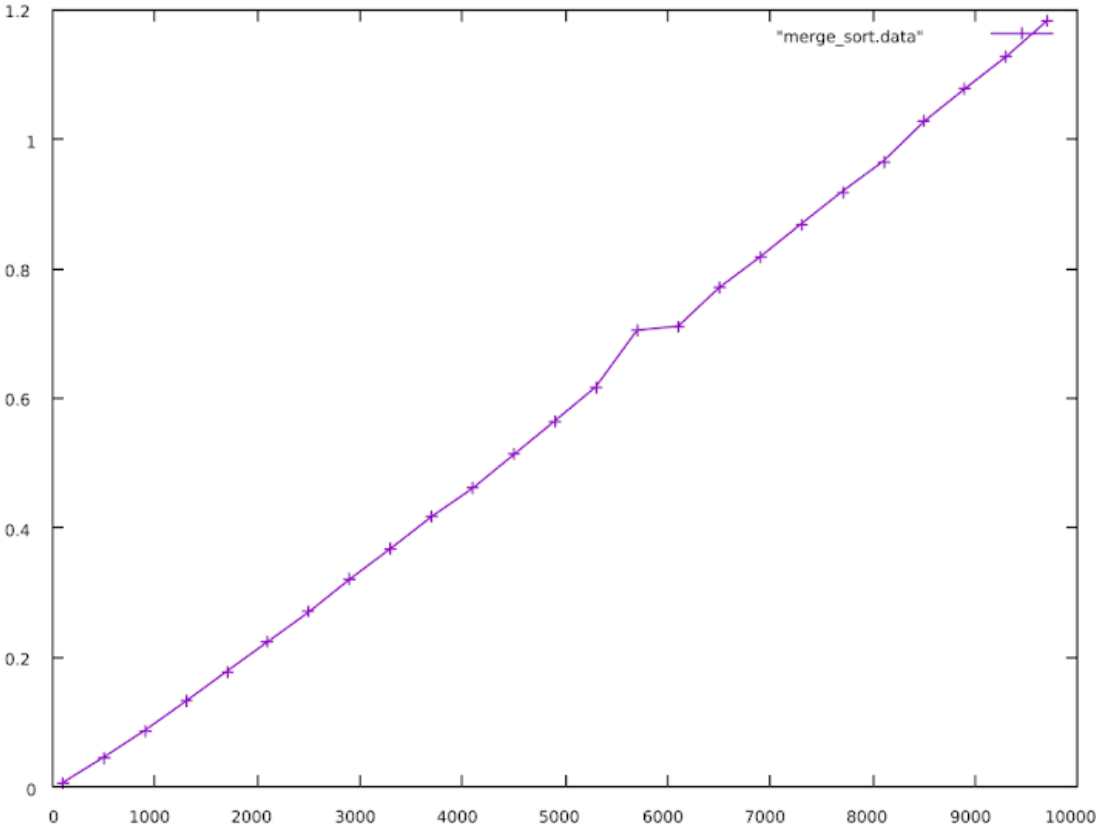
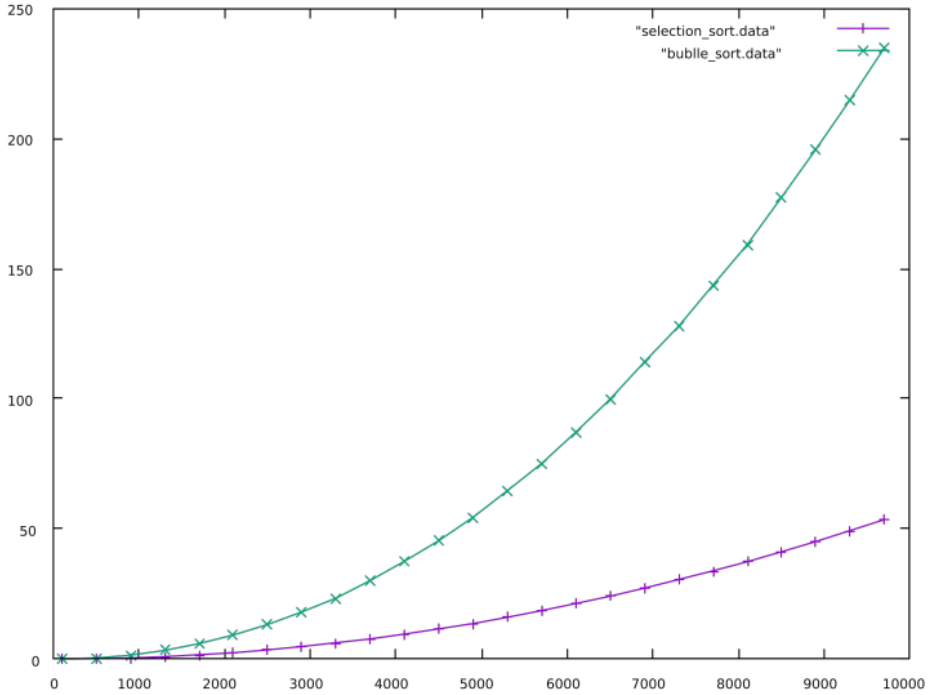


Figura 19: gráfico do cenário 4 bubble e selection



Cenário 5:

Figura 20: gráfico do cenário 5 para insertion, quick, radix e shell.

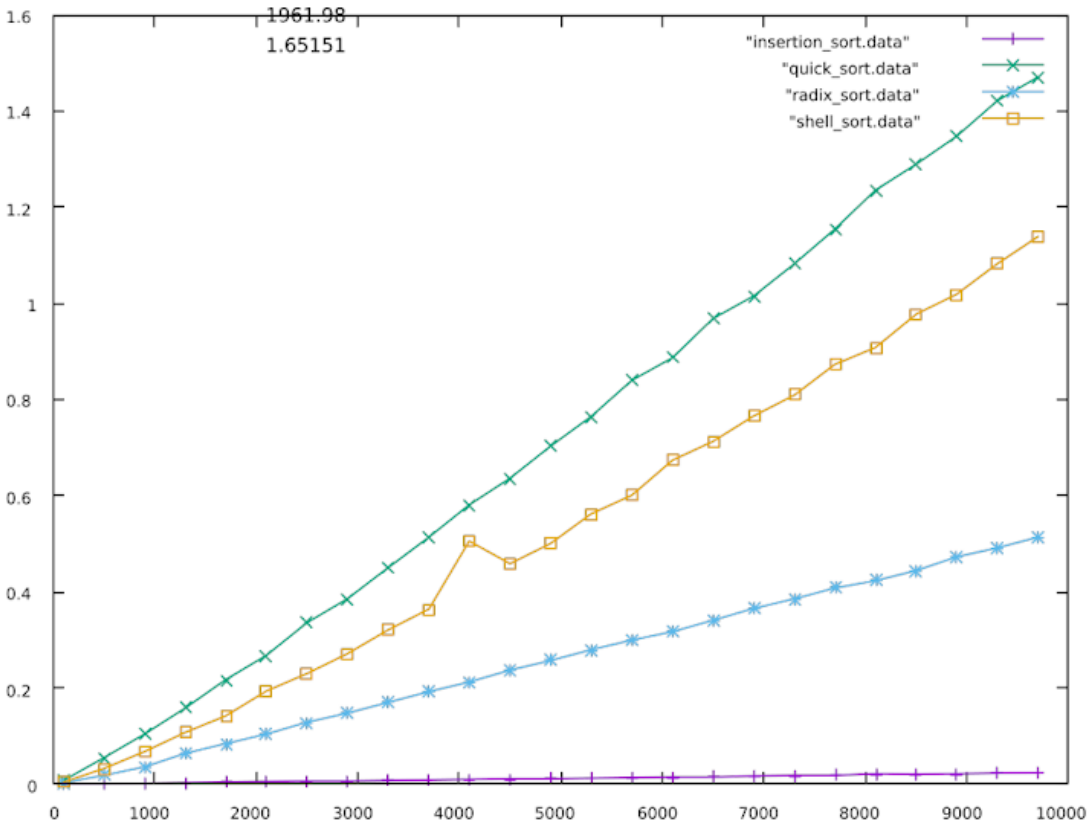


Figura 21: gráfico do cenário 5 merge

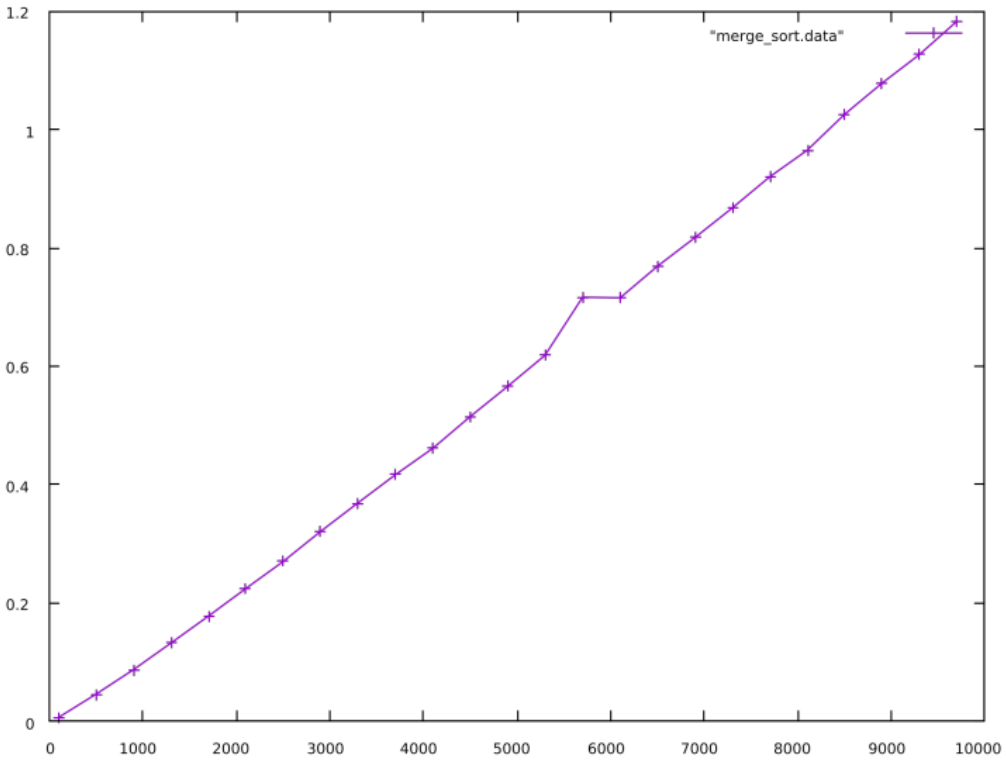
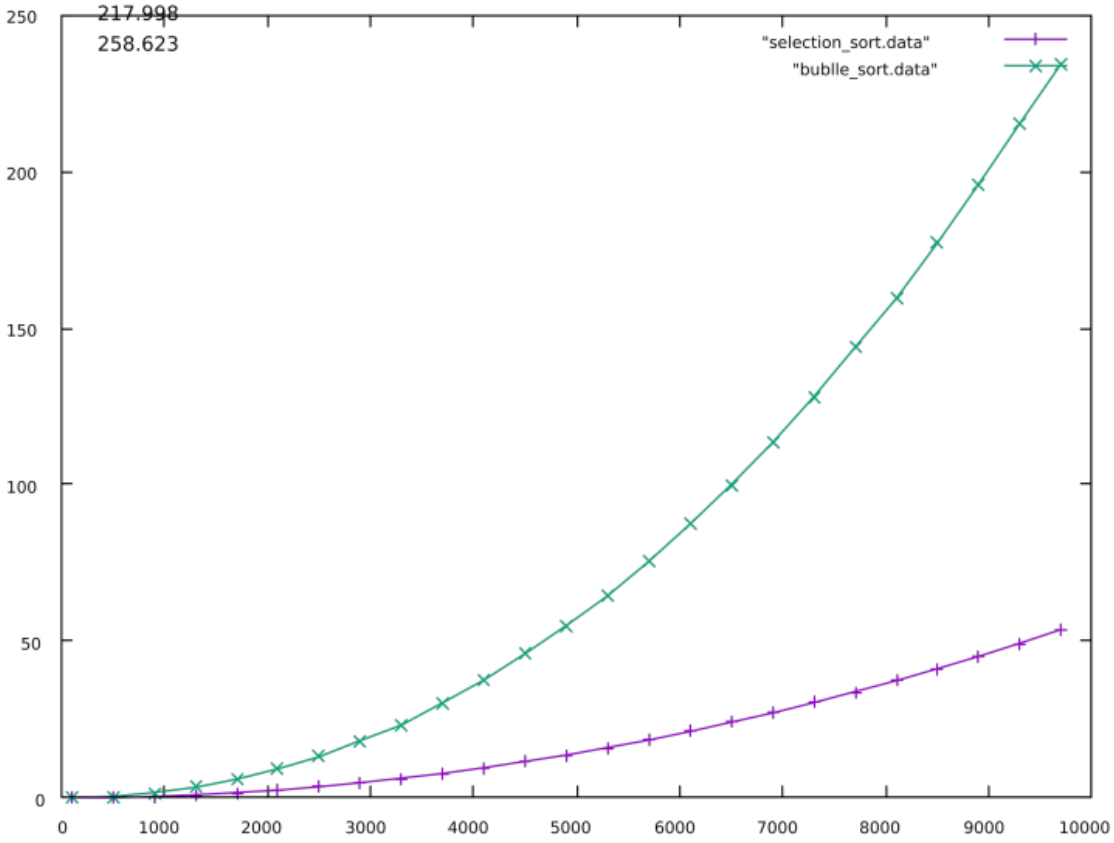


Figura 22: gráfico do cenário 5 bubble e selection



Cenário 6:

Figura 23: gráfico do cenário 6 para insertion, quick, radix e shell.

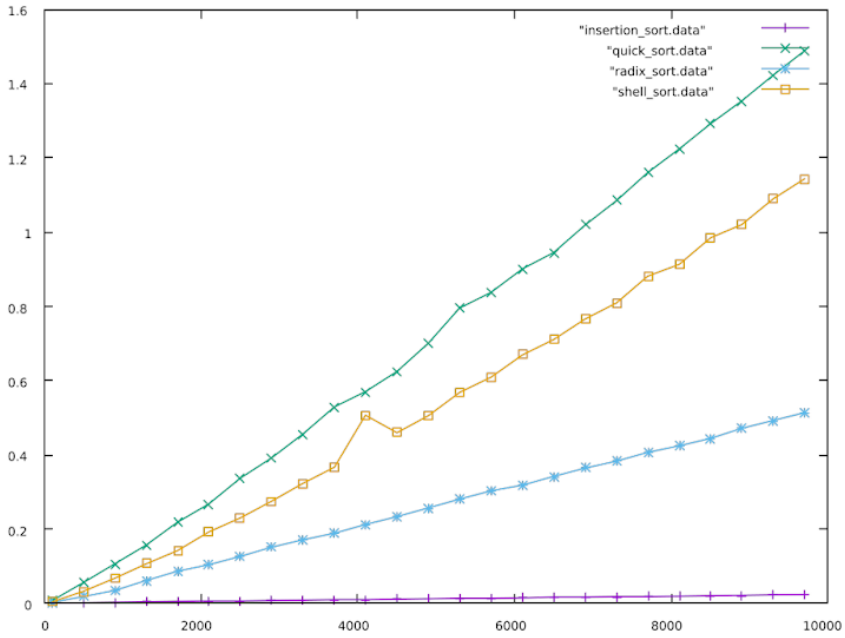


Figura 24: gráfico do cenário 6 merge

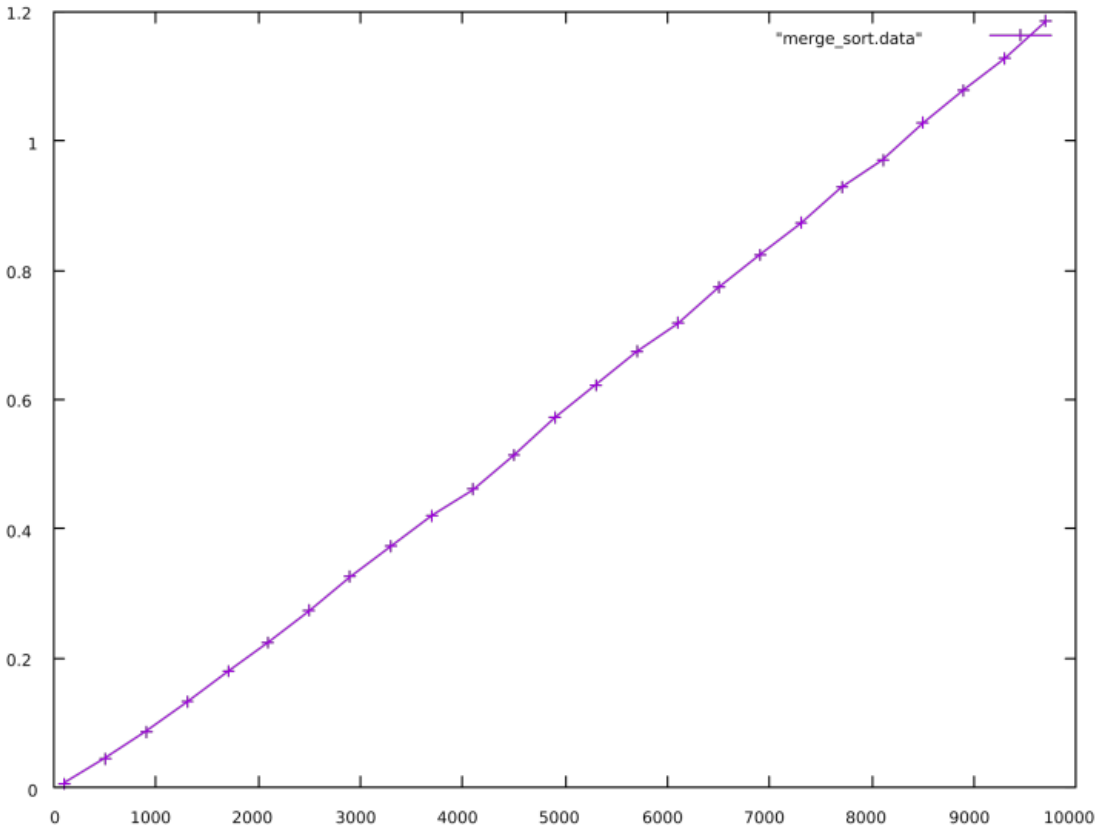
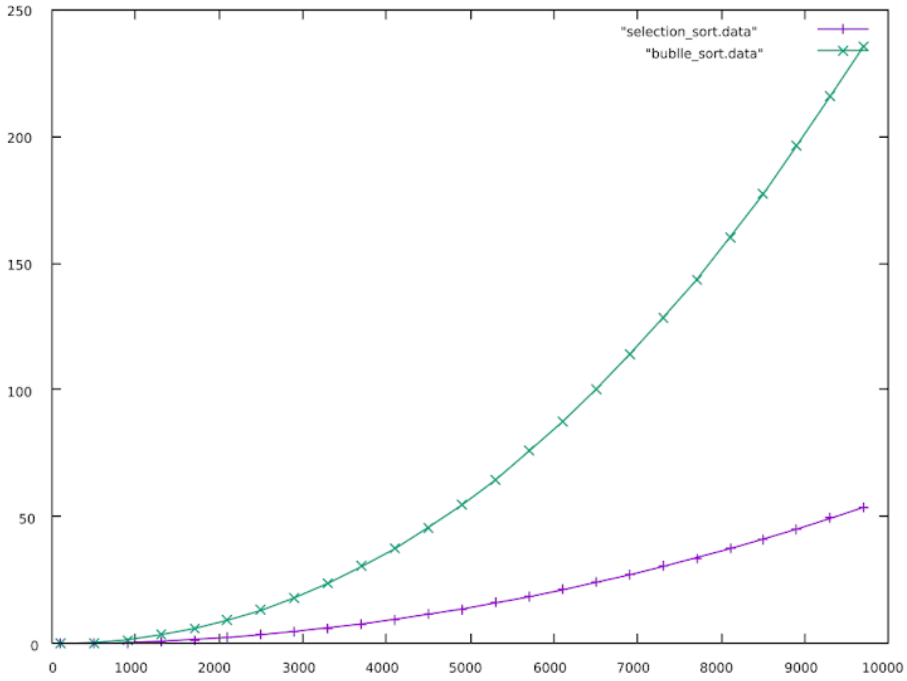


Figura 25: gráfico do cenário 6 bubble e selection



5. CONCLUSÃO E ANÁLISE

A abordagem escolhida durante a geração dos gráficos foi a sugerida pelo professor, ou seja, separamos os gráficos em quadráticos e logaritmos. Optamos por colocar o valor do tempo de execução em milisegundo no eixo Y e os valores das amostras no eixo X. Além disso foi gerado gráficos para cada um dos cenários de simulação dos algoritmos.

É válido ressaltar que durante a plotagem dos algoritmos quadráticos foi visualizado que os algoritmos quadráticos bubble sort e selection apresentam valores de tempo de execução desproporcionais quando comparados com os demais algoritmos quadráticos. Logo quando visualizados juntos isso impedia com que tivéssemos um bom discernimento sobre a eficiência e comportamento dos algoritmos de comportamento quadrático. Por conta disso optamos por gerar os gráficos do bubble e selection sort separados dos demais. Desse modo ficamos ao fim da plotagem com um total de 18 gráficos, ou seja, temos 3 gráficos para cada um dos 6 cenários.

Para uma melhor análise dos algoritmos quadráticos, como separamos eles em duas subcategorias, digamos os que tem um tempo de execução menor que 1.6 milissegundos e os que têm tempo de execução maior do que 1.6 milissegundos. Optamos por iniciar a análise pelos que têm tempo de execução menor que 1.6 milissegundos, para uma melhor síntese.

Como esperado pelo tempo de execucação o merge teve a melhor resposta e podemos ver isso pelo gráfico, que no pior caso se enquandra no tipo n log(n), o que era atencipado devido a literura visto em sala de aula.

Cada algoritmo mostra que tem funcionalidades adequadas para casos específicos, mostrando que não há uma forma genérica de melhor.

6. REFERÊNCIAS

Sedgewick and Wayne, 2002. Algorithms 4th edition.

Repositório: https://projetos.imd.ufrn.br/fernandoff/edb1-empirical-analysis

Geeks for geeks: https://www.geeksforgeeks.org/sorting-algorithms

GNUPLOT: http://www.gnuplot.info/