Relatório técnico

Sumário

Introdução

Sobre o projeto

O projeto de análise empírica tem como objetivo analisar o desempenho de um conjunto de algoritmos de ordenação baseado no seu tempo de execução e número de interações em arranjos de tamanho `n` crescente. O projeto trabalha com seis cenários de amostras: ordem completamente aleatória, ordem não decrescente, ordem não crescente, arranjo 75% ordenado, arranjo 50% ordenado e arranjo 25% ordenado.

Requisitos para reprodução e execução dos testes

- Sistema operacional: `linux`
- Aplicação auxiliar: `cmake`
- Compilador: `g++`
- Versionador: `git`
- Gerar gráficos: `RStudio` . Caso deseje utilizar outro método atente-se apenas a formatação dos arquivos.

Especificações da máquina utilizada

Componente	Modelo
Processador	AMD Ryzen 7 1800X, Oito Núcleos, Cache 20MB, 3.6GHz
Placa mãe	ASUS EX-A320M-Gaming Socket AM4 Chipset AMD A320
Memória	Memória RAM 24GB 3000 - 3200 MHz DDR4
Sistema	Ubuntu 18.04
Linguaguem	C++ 11
Versão do compilador	g++ 7.4.0

Compilação e execução

Para baixar o código na sua máquina:

```
$ git clone https://github.com/OnofreTZK/Empirical_Analysis.git
```

Para compilar:

```
$ cmake -S . -Bbuild
$ cd build/
$ make
```

Para executar:

```
$ ./ea_driver <array_max_size> <initial_sample_size> <number_of_samples>
<algorithm_ID>
```

- `<array_max_size>`: Tamanho máximo da amostra.
- `<initial_sample_size>`: Tamanho da amostra inicial.
- `<number_of_samples>`: Número de amostras

- `<algorithm_ID>`: código do algoritmo de ordenação
  - **[0]** -- *Quick Sort*
  - **[1]** -- *Selection Sort*
  - **[2]** -- *Insertion Sort*
  - **[3]** -- *Bubble Sort*
  - **[4]** -- *Radix Sort*
  - **[5]** -- *Shell Sort*
  - **[6]** -- *Quick + Insertion*
  - **[7]** -- *Merge Sort*

O programa aceita até três algoritmos para análise em uma única execução, exemplo:

```
$ ./ea_drive 10000 5000 50 0 1 2
```

Análise

Processando as linhas de comando

Após a leitura e o processamento de argumentos, todos os dados são passados para a função principal `execute_analysis` :

```
// Exemplo de uma execução com 3 algoritmos.

std::cout << "\n>>>Starting " << data.sort_ID[algo1] << " analysis\n";

execute_analysis( array[algo1], algo1, MAX, samples, init_sample, data );

std::cout << "\n>>>Starting " << data.sort_ID[algo2] << " analysis\n";

execute_analysis( array[algo2], algo2, MAX, samples, init_sample, data );

std::cout << "\n>>>Starting " << data.sort_ID[algo3] << " analysis\n";

execute_analysis( array[algo3], algo3, MAX, samples, init_sample, data );

return EXIT_SUCCESS;
```

A partir dela os dados serão distribuidos para que o programa possa medir e guardar o que for necessário:

```
void execute_analysis( algorithms func, int algorithm_ID, long int max,
                       long int samples, long int init_sample, DATA data )
{
    // Gerando um vetor completamente aleatório.
    long int * array = generate_array( max );

    // Variável para calcular a média do tempo
    //=========================================================================
    double arithmetic_mean;// será utilizada a média aritmética progressiva
    //=========================================================================

    data.alocate_vectors( samples );

    for( int type = 0; type < 6; type++ )
    {

        for( int iter_samples = 0; iter_samples < samples; iter_samples++)
        {

            arithmetic_mean = 0.0; // zerando a média

            for( int time_control = 0; time_control < 10; time_control++ )
            {
                type_array( array, type, data, max );

                // -- CONTADOR COMEÇA AQUI --
                std::chrono::steady_clock::time_point START =
std::chrono::steady_clock::now();
                func( array, array + increase_array_control( max, samples,
init_sample, iter_samples + 1 ) );
                std::chrono::steady_clock::time_point STOP =
std::chrono::steady_clock::now();
                // -- CONTADOR TERMINA AQUI --

                auto timer = (STOP - START);

                //Cálculo da média progressiva.
                arithmetic_mean = arithmetic_mean + ( ( std::chrono::duration<
double, std::milli > (timer).count() - arithmetic_mean )/(time_control+1) );

            }
            // Aloca de acordo com o número de amostras.
            long int sample = increase_array_control( max, samples,
init_sample, iter_samples + 1 );

            // Guarda os valores para gerar o arquivo.
            data.set_values( sample, arithmetic_mean, iter_samples );
        }
```

```
        // Cria o arquivo.
        create_data_file( data, algorithm_ID, type );

    }

    delete[] array; // libera a memória.
}
```

Funções principais

- `generate_array`

```
//============================================================================
===========
// aloca e preenche o vetor com números em um intervalo de [0,
<array_max_size>)
//============================================================================
===========
long int generate_numbers( long int range )
{
      std::random_device seed;

      std::mt19937_64 gen( seed() );
      // aleatoriedade com probabilidade linear.
      std::uniform_int_distribution<long int> distr(0, range);

      return distr( gen );
}


long int * generate_array( long int max )
{
    // alocação utilizando a entrada do usuário.
    long int * array = new long int [max];

    for ( int i = 0; i < max; i++ )
    {
        array[i] = generate_numbers(max);
    }

    return array;
}
//============================================================================
========
```

- `increase_array_control`

```
//================================================================================
========
// Crescimento linear das amostras utilizando P.A. ( <init_sample>,
<array_max_size> )
//================================================================================
========
long int increase_array_control( long int max, long int samples, long int
init_sample, int index )
{
    //calculando o 'r'( razão) progressão aritmética.
    long int reason = ( max - init_sample )/ samples;

    return init_sample + (( index - 1 ) * reason);

}
//================================================================================
========
```

O método de crescimento da `amostra inicial` ao `tamanho máximo` é linear

Fórmula :

$amostra_n = amostra_1 + (n-1)reason$

`n` = índice da amostra.

`reason` = $\frac{amostra_m - amostra_0}{S}$

`S` = número de amostras.

- **Cálculo da média aritmética progressiva**

```
arithmetic_mean = arithmetic_mean + ( ( std::chrono::duration< double,
std::milli > (timer).count() - arithmetic_mean )/(time_control+1) );
```

Buscando o máximo de precisão cada amostra é analisada 10 vezes e o tempo guardado é a média de todas as execuções. Com o intuito de evitar erros de arredondamento o programa utiliza a seguinte fórmula:

$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k}$

- **`type_array`**

```
//================================================================================
```

```
// Seleciona baseado no laço qual amostra deve ser gerada para análise.
//========================================================================
void type_array( long int * array ,int type, DATA data, long int max )
{

        switch( type )
        {
            case 0 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                random_array( array, array + max, max );
                break;
            case 1 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                ascending_sorting( array, array + max );
                break;
            case 2 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                descending_sorting( array, array + max );
                break;
            case 3 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                ascending_sorting( array, array + max );
                partial_sorting( array, array + max, 25 );
                break;
            case 4 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                ascending_sorting( array, array + max );
                partial_sorting( array, array + max, 50);
                break;
            case 5 :
                std::cout << "\nTEST " << type+1 << " -- " <<
data.typesample[type] << "\n";
                ascending_sorting( array, array + max );
                partial_sorting( array, array + max, 75 );
                break;
            default:
                std::cout << "\nHow?\n";
                break;
        }
}
//========================================================================
```

Algoritmos

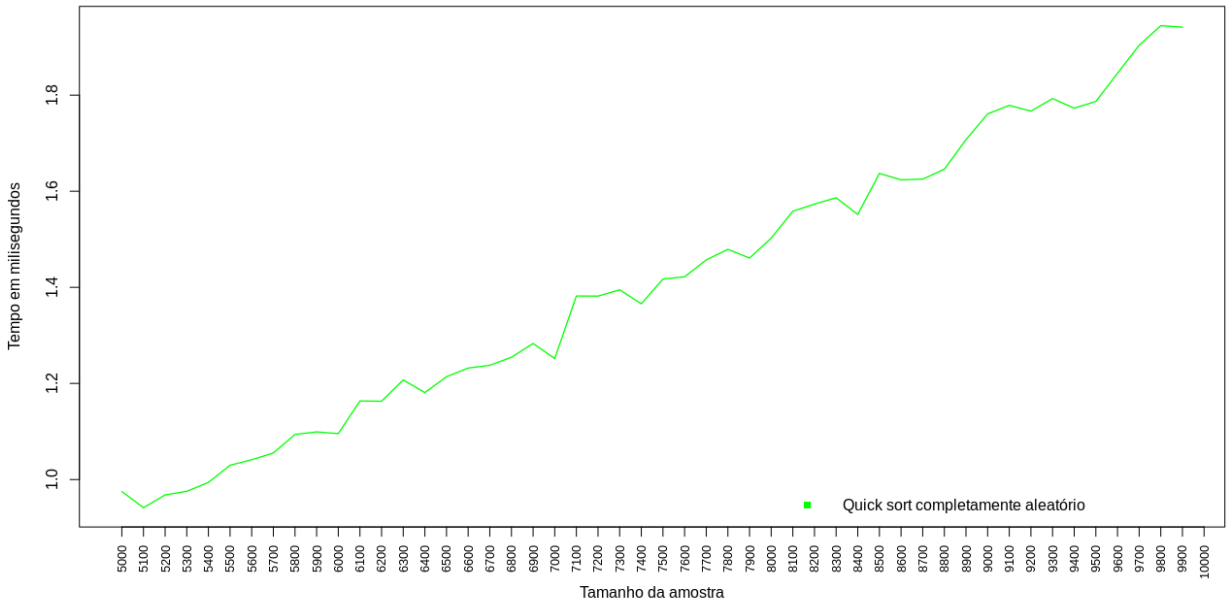
Quick Sort

- É o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Complexidade no pior caso: $O(n^2)$
- Complexidade no melhor e médio caso: $O(nlogn)$
- Não é estável.
- Estratégia: ***dividir e conquistar***
- Funcionamento:

```
procedimento QuickSort(X[], IniVet, FimVet)
var
   i, j, pivo, aux
início
   i <- IniVet
   j <- FimVet
   pivo <- X[(IniVet + FimVet) div 2]
   enquanto(i <= j)
    |      enquanto (X[i] < pivo) faça
    |       |   i <- i + 1
    |      fimEnquanto
    |      enquanto (X[j] > pivo) faça
    |       |   j <- j - 1
    |      fimEnquanto
    |      se (i <= j) então
    |       |   aux  <- X[i]
    |       |   X[i] <- X[j]
    |       |   X[j] <- aux
    |       |   i <- i + 1
    |       |   j <- j - 1
    |      fimSe
   fimEnquanto
   se (IniVet < j) então
    |  QuickSort(X, IniVet, j)
   fimSe
   se (i < FimVet) então
    |  QuickSort(X, i, FimVet)
   fimse
fimprocedimento
```

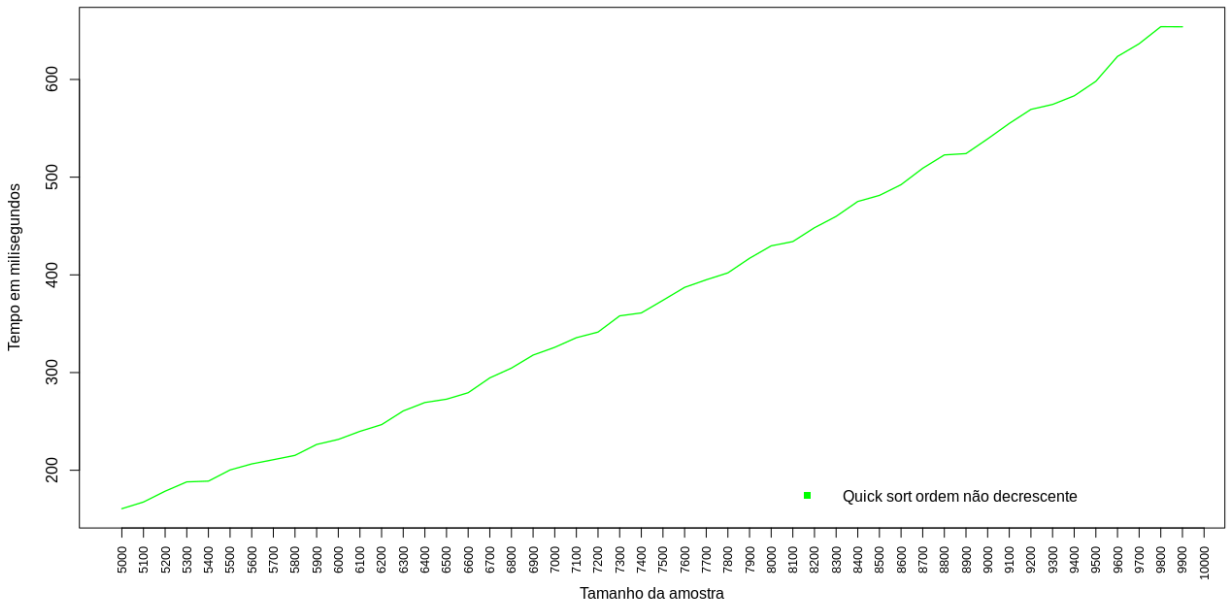
Gráficos:

`Cenário`: *Arranjo completamente aleatório*



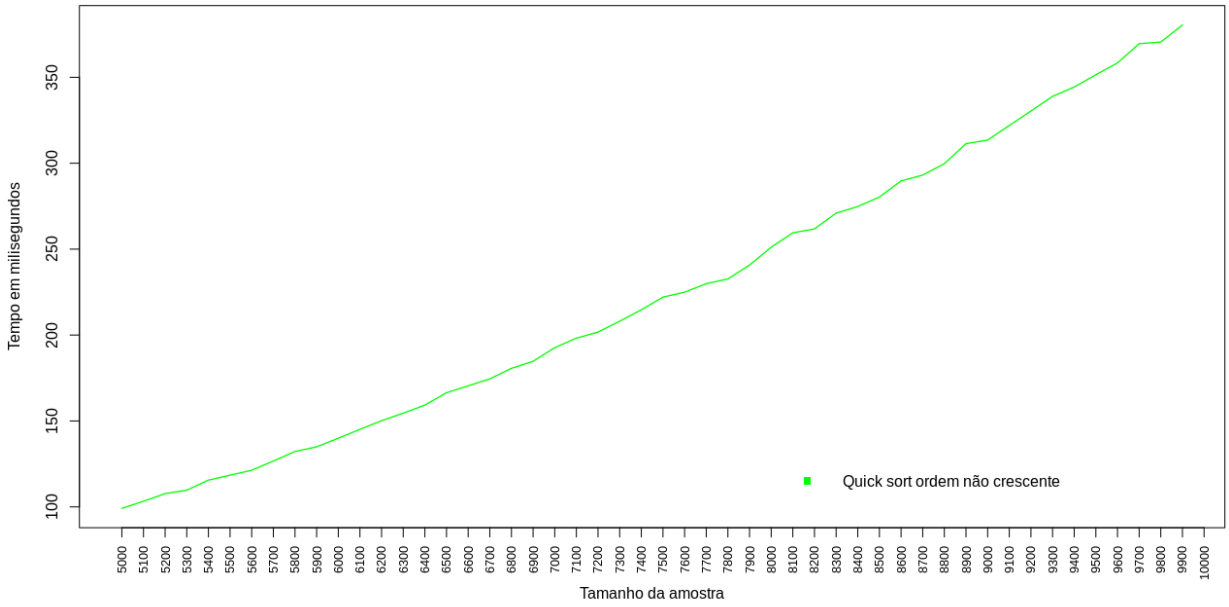
Os picos representam os momentos em que as amostras tendem ao pior caso, porém a eficiência do algoritmo é excelente em um arranjo sem ordem total.

`Cenário`: *Arranjo em ordem não decrescente*



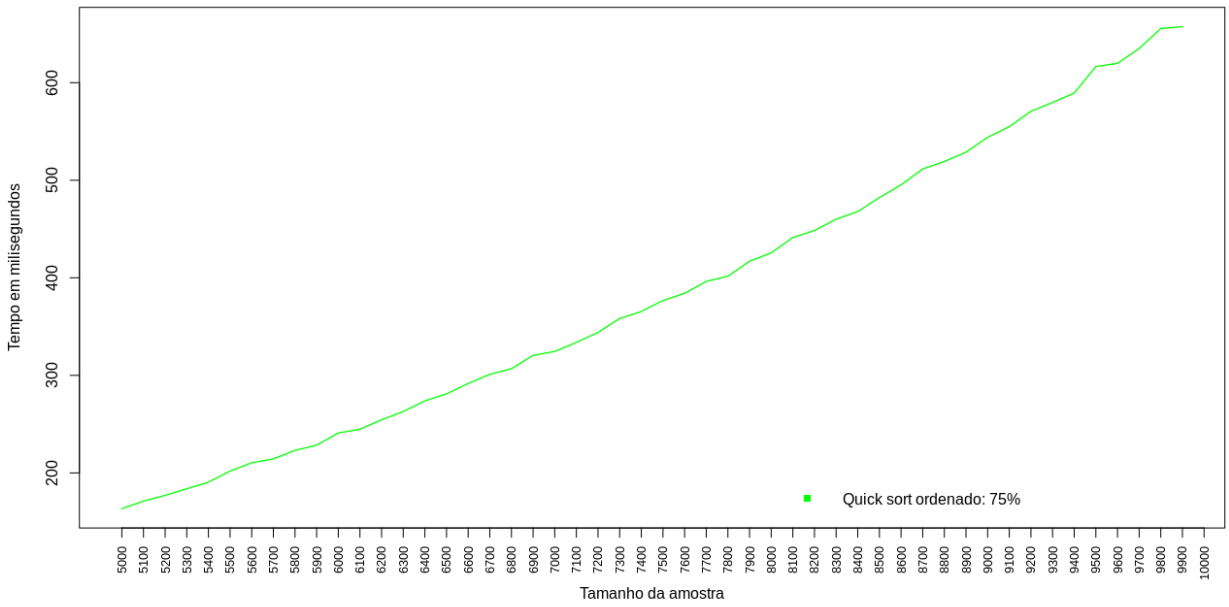
Aqui temos um exemplo de pior caso demonstrado com o pivô igual a `last - 1` e todos os elementos menores à sua esquerda. Com a limitação da máquina não foi possível analisar amostras muito maiores para melhor visualização da curva da parábola.

`Cenário`: *Arranjo em ordem não crescente*

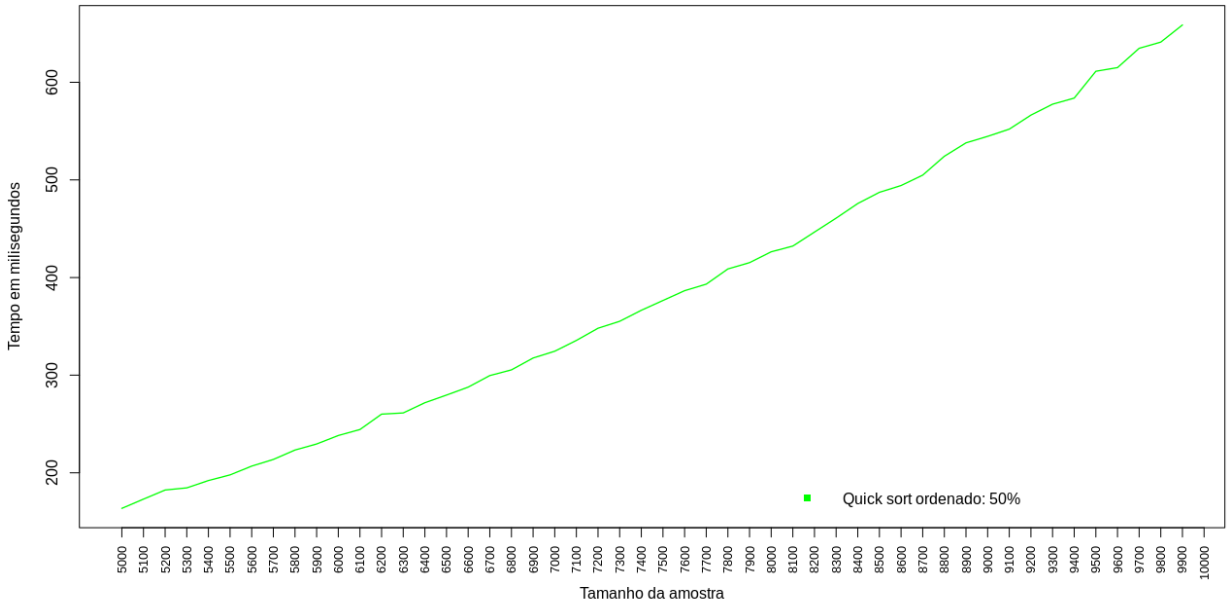


Assim como o gráfico anterior, esse demonstra outro exemplo de pior caso com o pivô igual a `first` e todos os elementos maiores à sua direita.

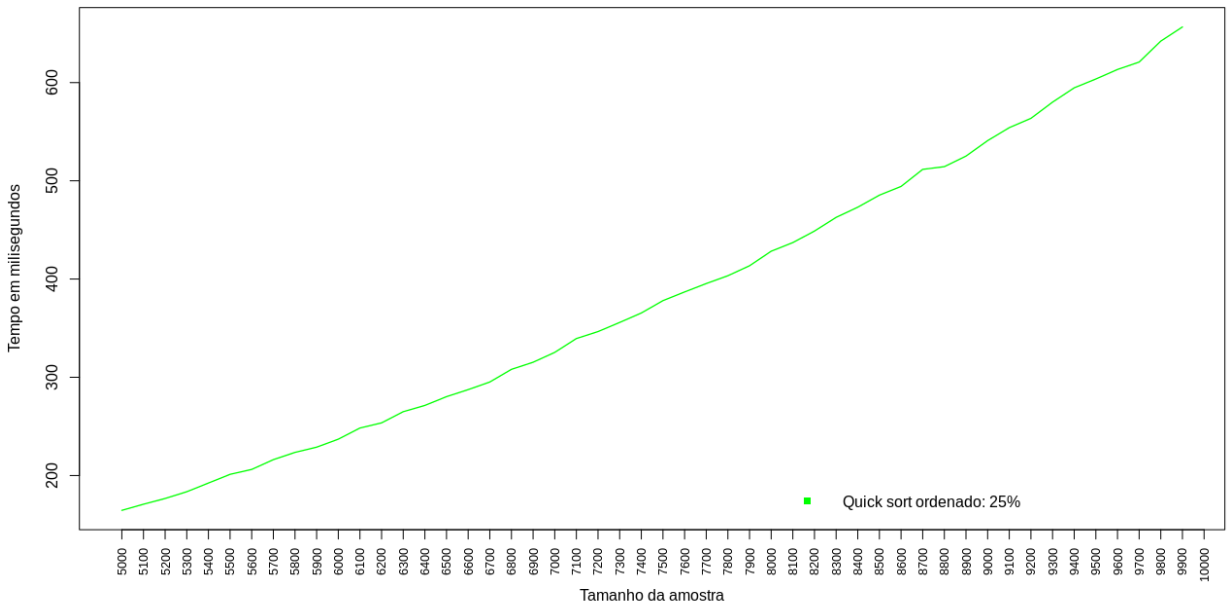
`Cenário`: *Arranjo 75% ordenado*



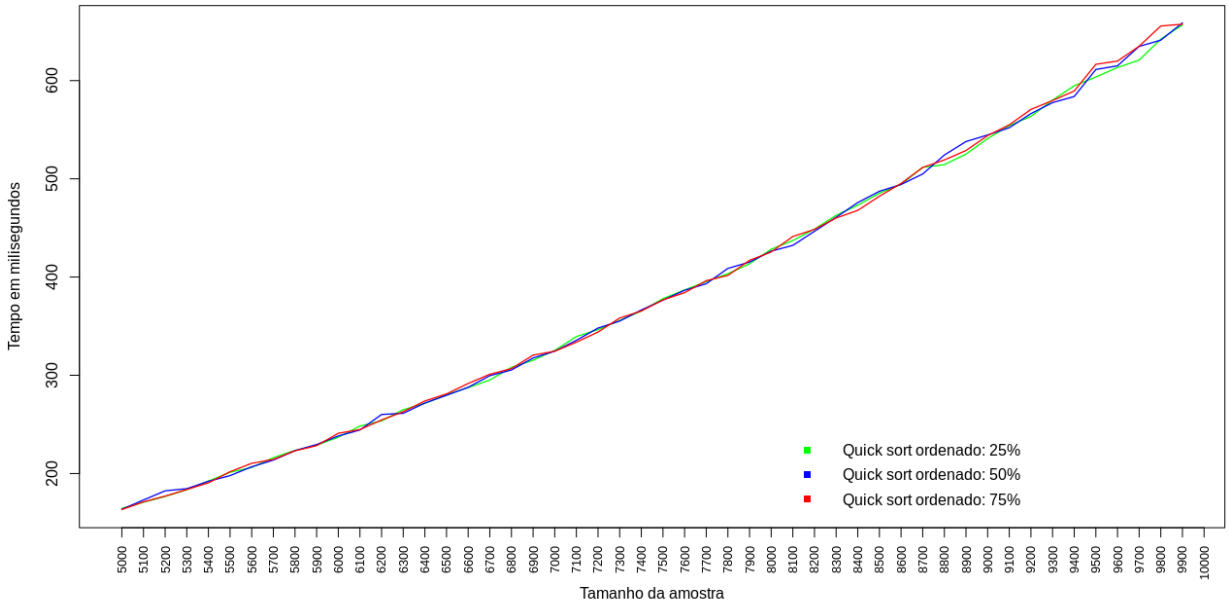
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjos parcialmente ordenados*



Pela observação comparativa dos gráficos há uma queda de desempenho do algoritmo quando a divisão do vetor é desbalanceada(pivô tendendo as pontas), com isso pode-se concluir que o quick sort não é a melhor opção para arranjos parcialmente ordenados.

Insertion Sort

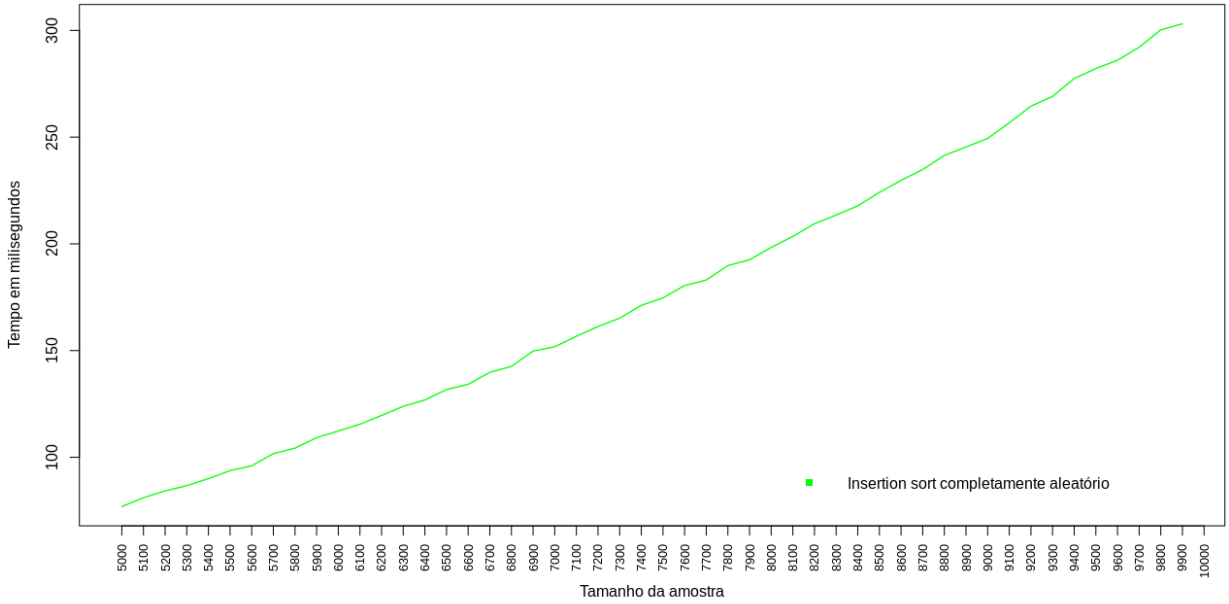
- Ordenação por inserção
- Percorre um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda
- Complexidade no melhor caso: $O(n)$
- Complexidade no pior caso: $O(n^2)$
- Método de ordenação estável.
- Funcionamento:

```
FUNÇÃO INSERTION_SORT (A[], tamanho)
        VARIÁVEIS
                i, j, eleito
        PARA i <- 1 ATÉ (tamanho-1) FAÇA
                eleito <- A[i];
                j <- i-1;
                ENQUANTO ((j>=0) E (eleito < A[j])) FAÇA
                        A[j+1]:= A[j];
# Elemento de lista numerada
                        j:=j-1;
                FIM_ENQUANTO
                A[j+1] <- eleito;
```

```
        FIM_PARA
FIM
```

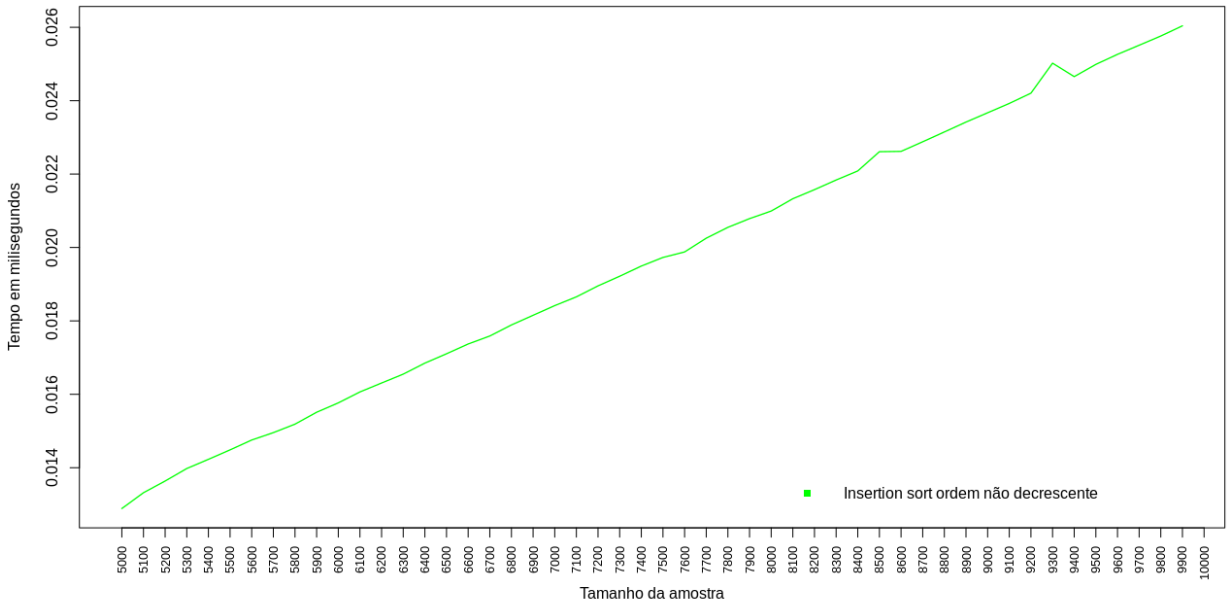
Gráficos :

`Cenário`: *Arranjo completamente aleatório*



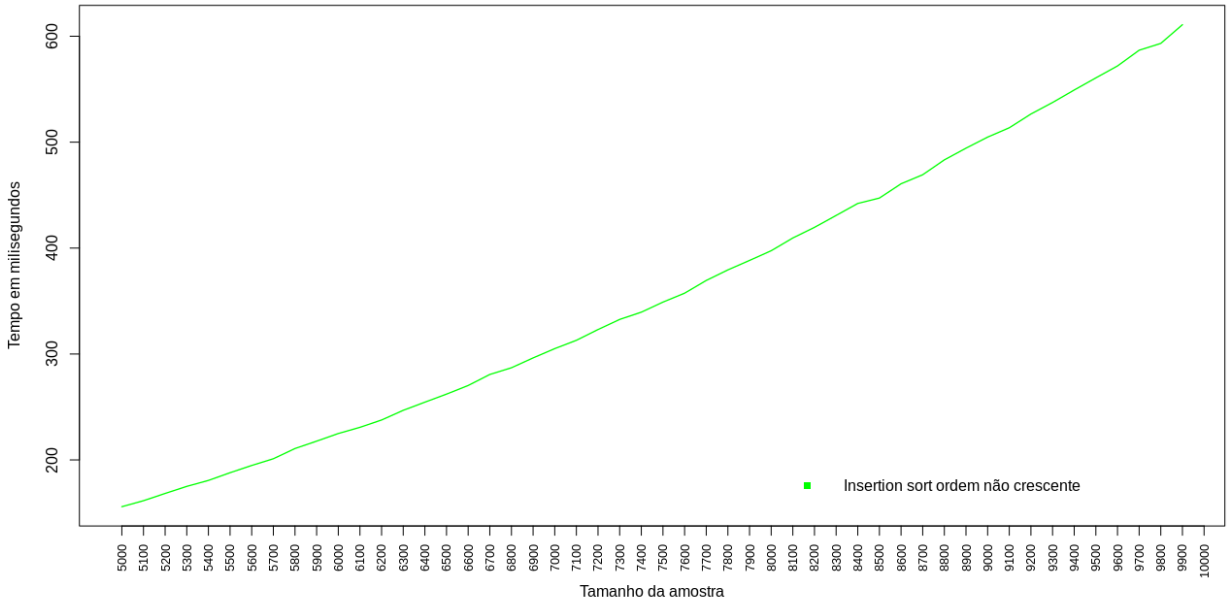
Em um arranjo não ordenado o algoritmo não tem uma boa eficiência e deve ser evitado para amostras muito grandes.

`Cenário`: *Arranjo em ordem não decrescente*



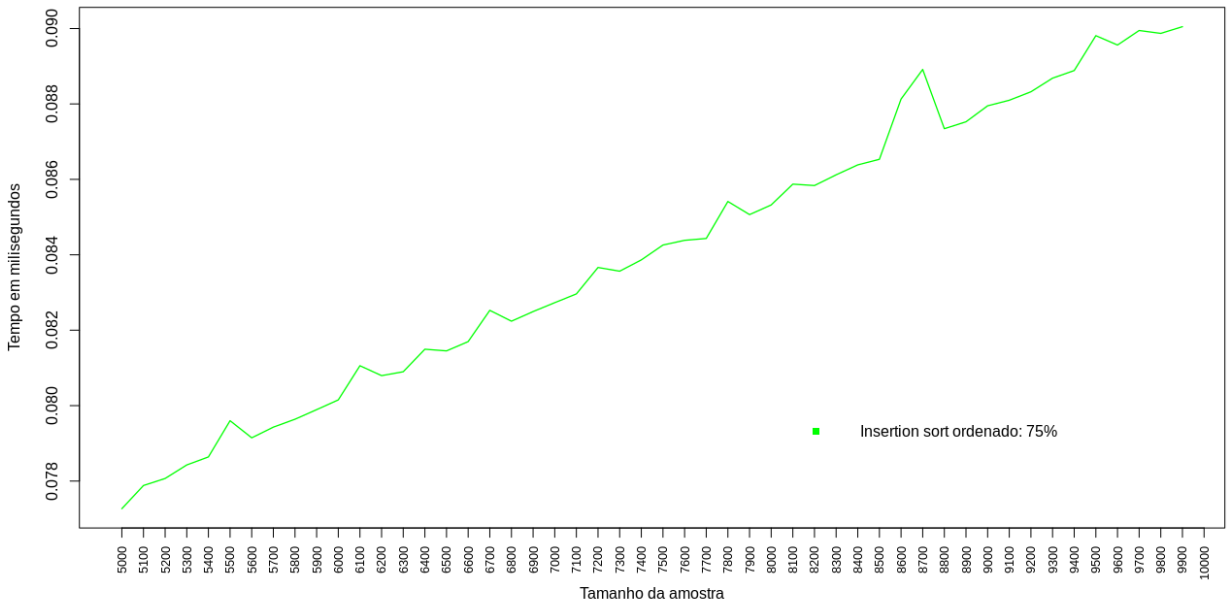
Visualização do insertion sort no seu melhor caso com o gráfico praticamente linear. Os picos se devem a variação de desempenho da máquina uma vez que o arranjo não possui repetição aumentando o número de interações.

`Cenário`: *Arranjo em ordem não crescente*

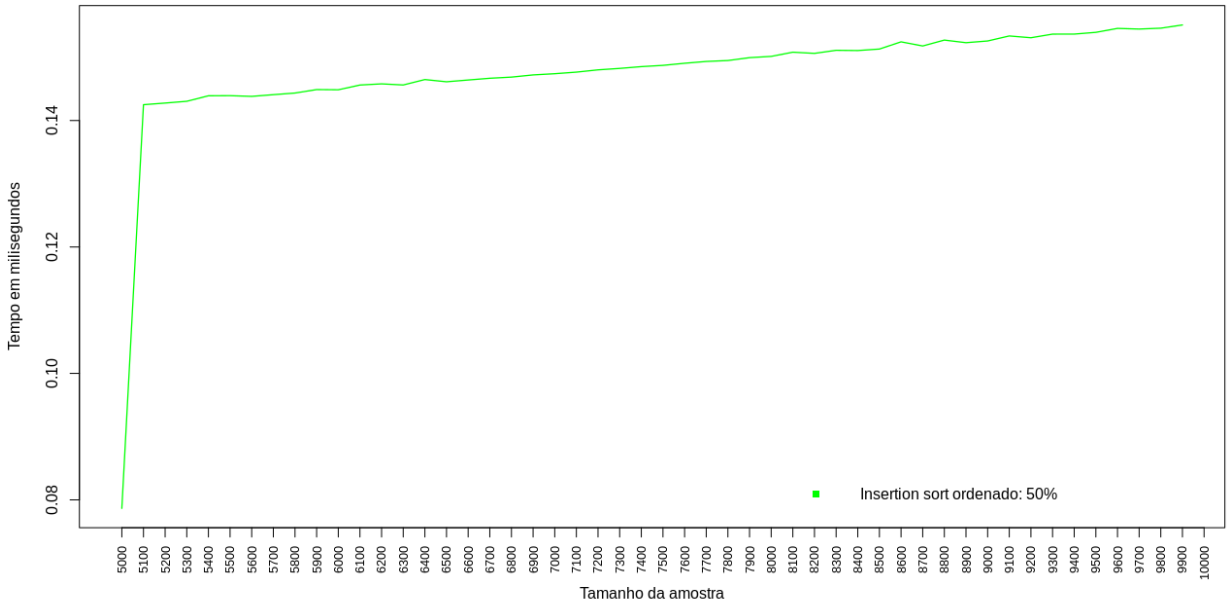


Visualização do pior caso, onde o algoritmo inverte completamente o vetor e sua complexidade é inevitavelmente quadrática.

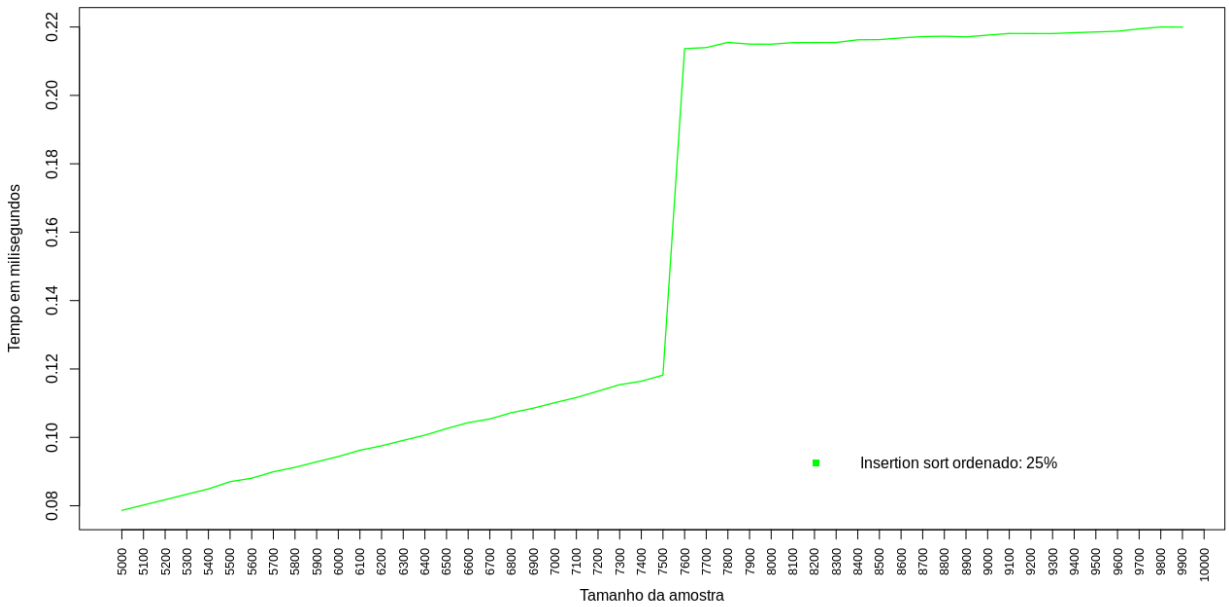
`Cenário`: *Arranjo 75% ordenado*



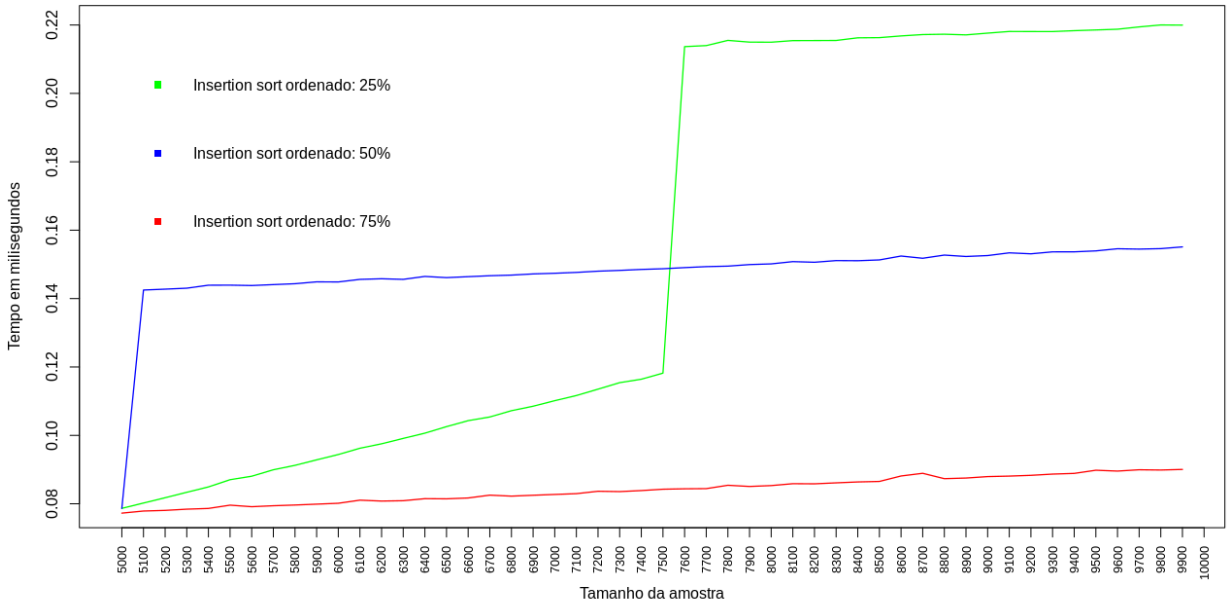
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*



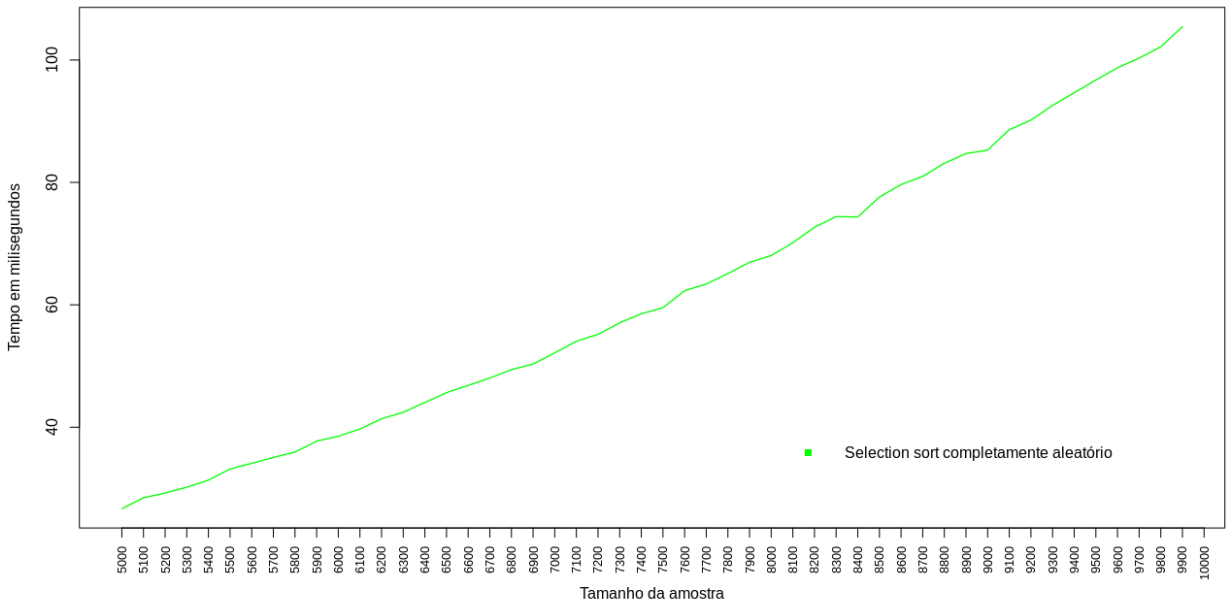
O melhor cenário para o insertion é um arranjo com ordem parcial tendendo a ordem total. Fica clara a redução e/ou inexistência de picos no tempo de execução quanto mais ordenado estiver o vetor.

Selection Sort

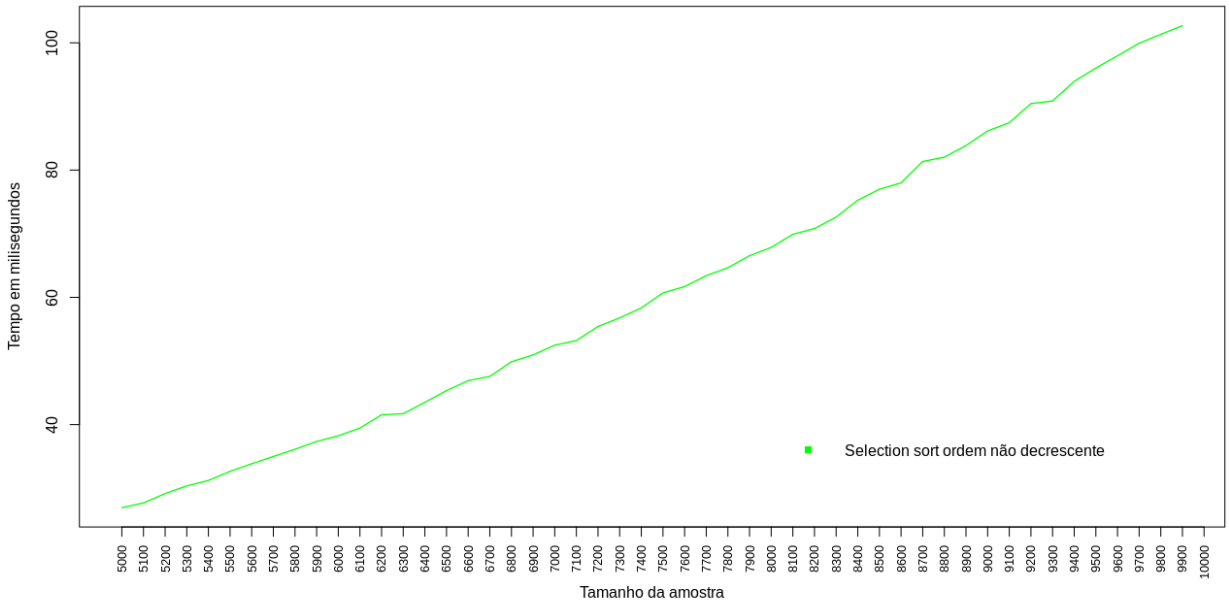
- Ordenação por seleção: seleciona o menor item e colocar na primeira posição, seleciona o segundo menor item e colocar na segunda posição, segue estes passos até que reste um único elemento.
- Complexidade em todos os casos: $O(n^2)$
- Não é estável.
- Funcionamento:

```
var i,j,aux,menor : INTEIRO
PARA i=0 ATE (vet.tamanho-1) PASSO 1
 menor = i
 PARA j=i+1 ATE vet.tamanho PASSO 1
  SE (vet[menor] > vet[j]) ENTAO
   menor = j
   FIMSE
   FIMPARA
  SE (menor != i) ENTAO
   aux = vet[menor]
   vet[menor] = vet[i]
   vet[i] = aux
 FIMSE
FIMPARA
```

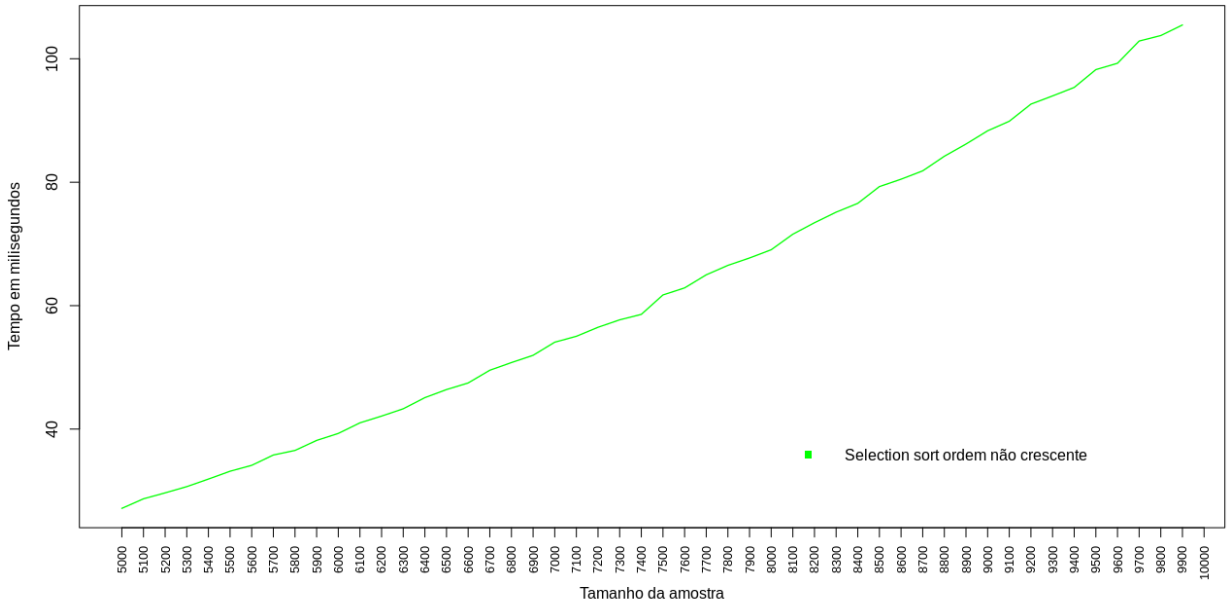
`Cenário`: *Arranjo completamente aleatório*



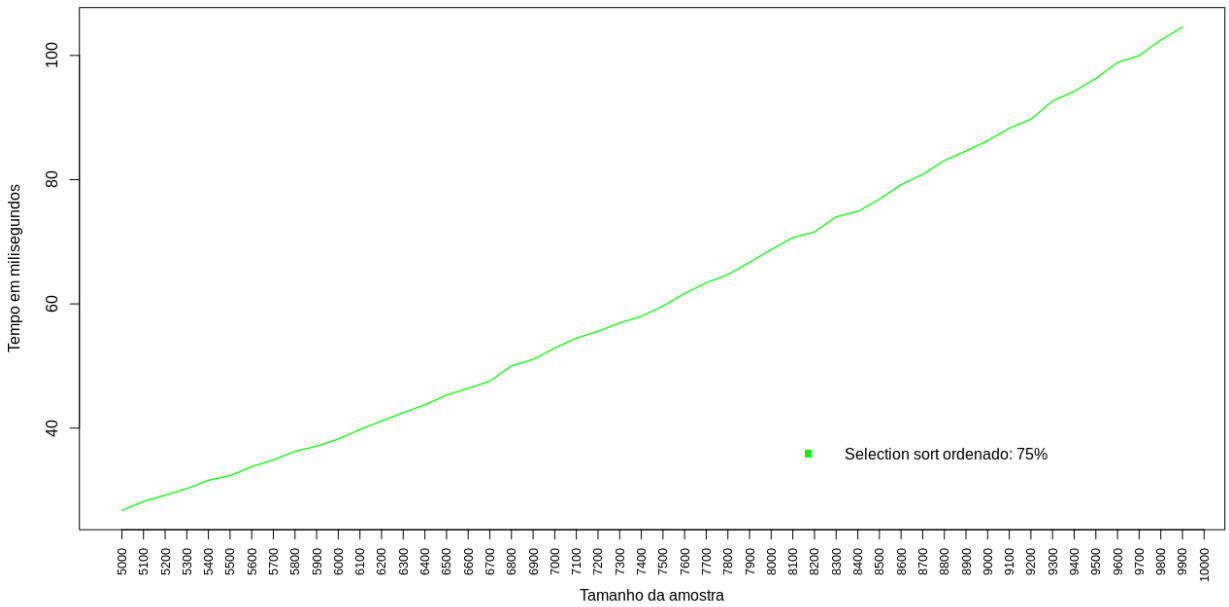
`Cenário`: *Arranjo em ordem não decrescente*



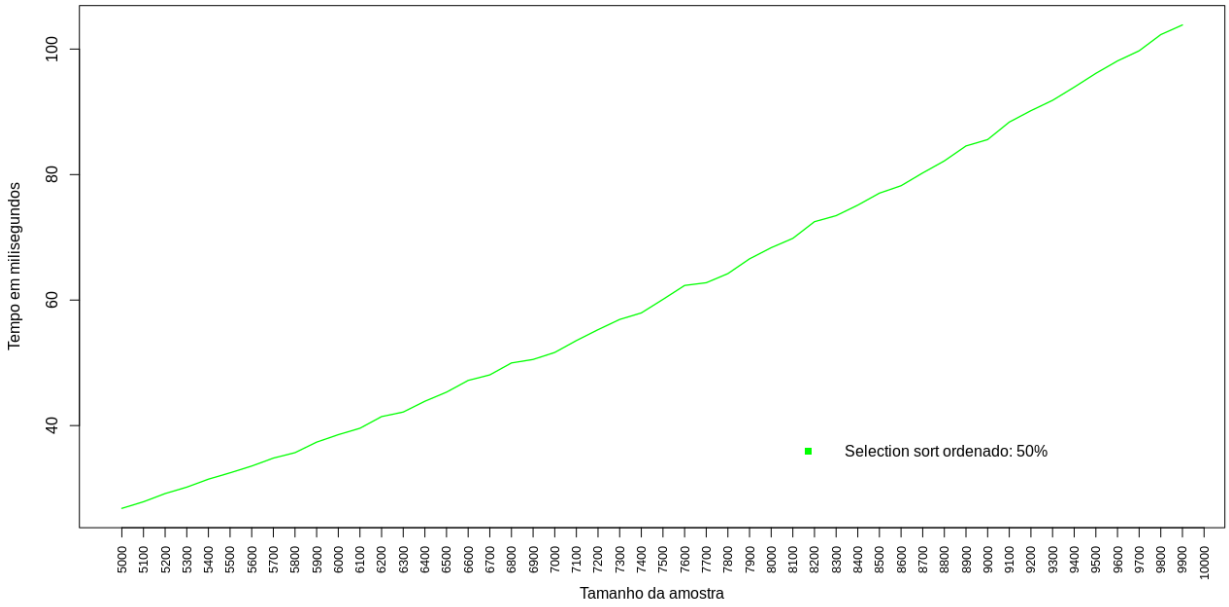
`Cenário`: *Arranjo em ordem não crescente*



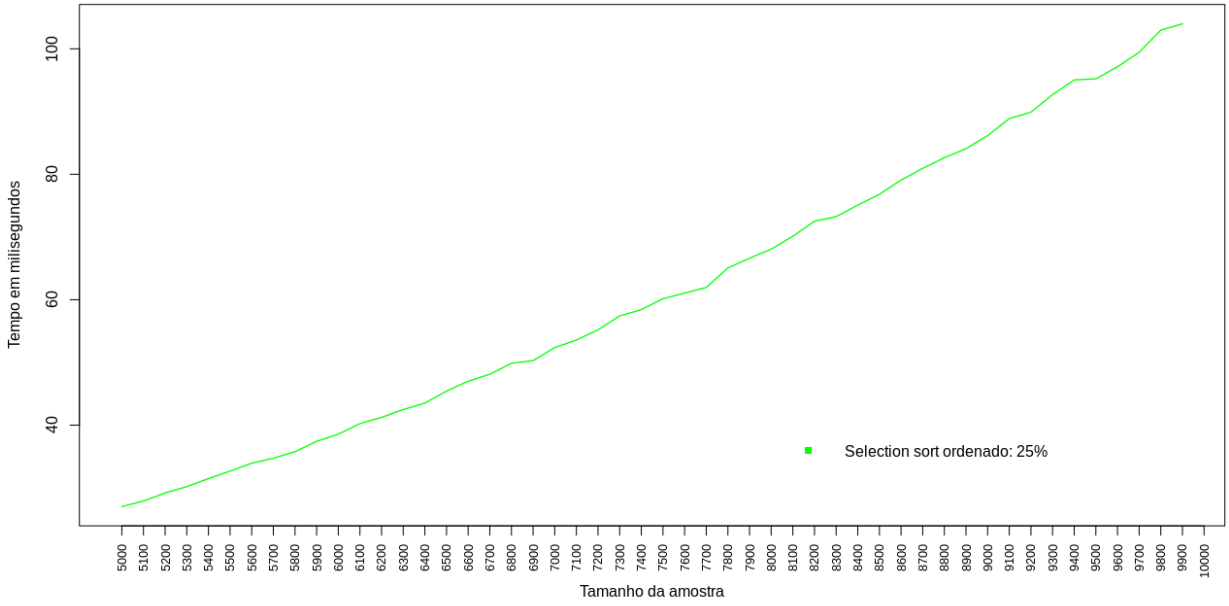
`Cenário`: *Arranjo 75% ordenado*



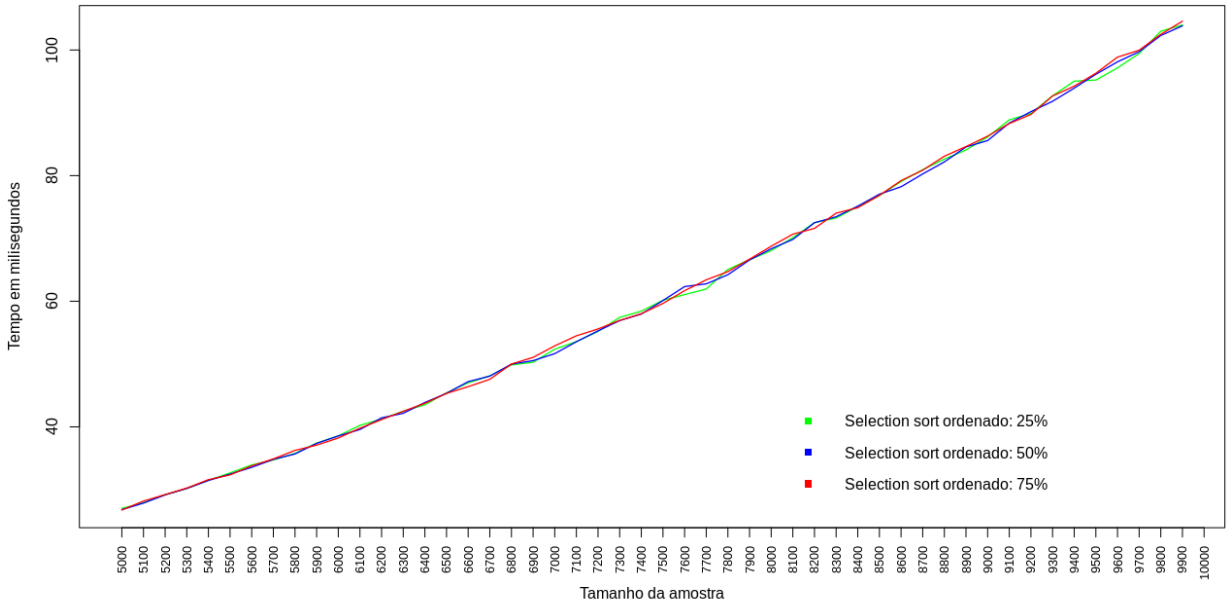
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 50% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*



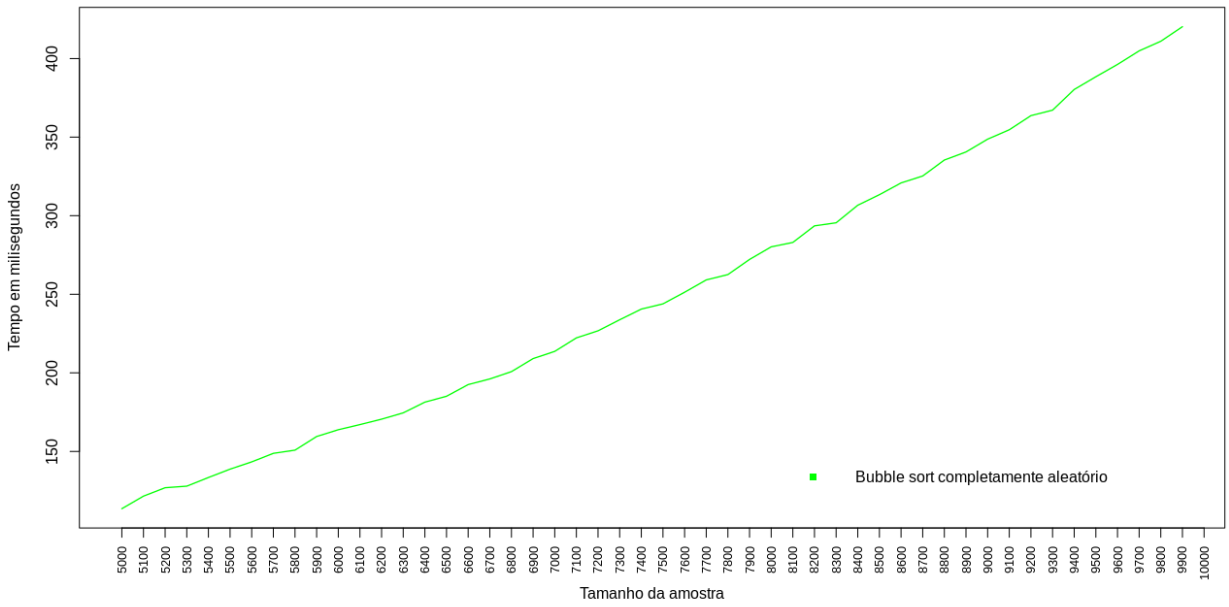
O selection sort tem como única vantagem a sua implementação simples. Todos os gráficos possuem praticamente o mesmo tempo de execução nos 6 cenários apresentados, isso se deve ao fato de que antes de fazer qualquer troca, o algoritmo realiza uma busca sequencial($O(n-i)$) para cada posição do vetor, tornando-o ineficiente.

Bubble Sort

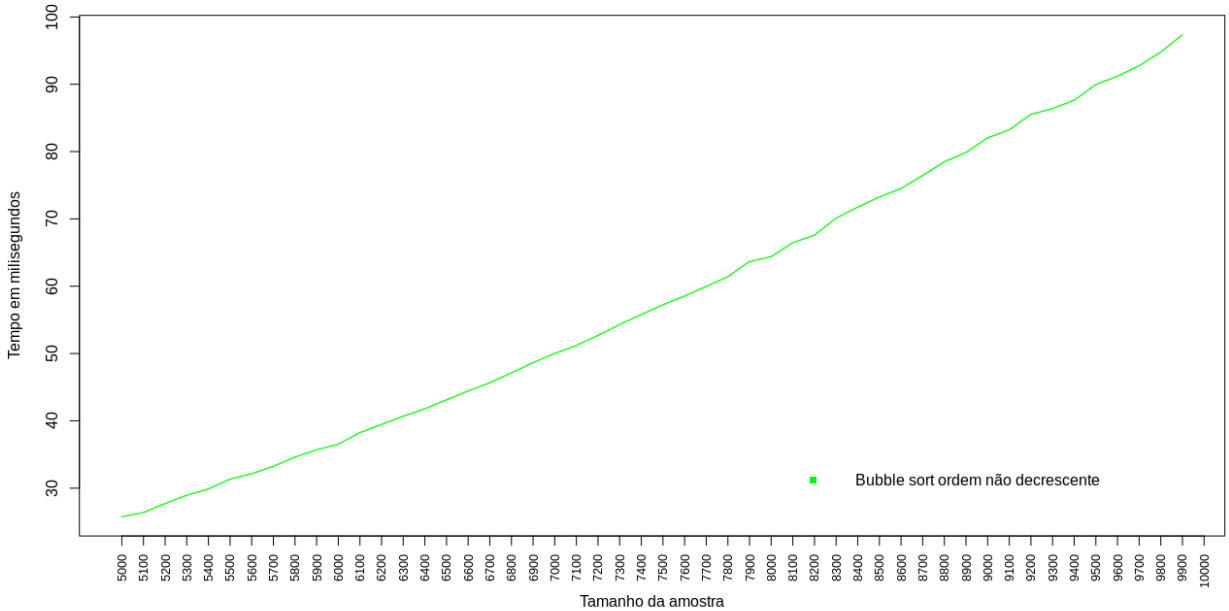
- Ordenação por flutuação: a ideia é percorrer o arranjo diversas vezes, e a cada passagem fazer "flutuar" para o topo o maior elemento da sequência.
- Complexidade no melhor caso: $O(n)$
- Complexidade no pior caso: $O(n^2)$
- Funcionamento:

```
procedure bubbleSort( A : lista de itens ordenaveis ) defined as:
  do
    trocado := false
    for each i in 0 to length( A ) - 2 do:
      // verificar se os elementos estão na ordem certa
      if A[ i ] > A[ i + 1 ] then
        // trocar elementos de lugar
        trocar( A[ i ], A[ i + 1 ] )
        trocado := true
      end if
    end for
  // enquanto houver elementos sendo reordenados.
  while trocado
end procedure
```

`Cenário`: *Arranjo completamente aleatório*

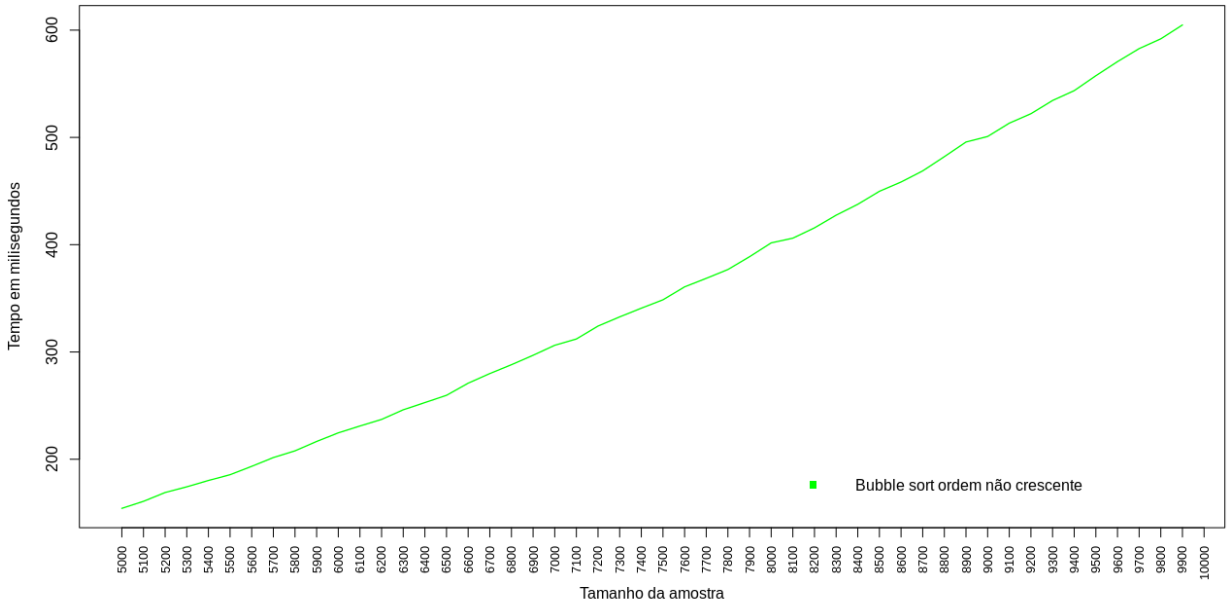


`Cenário`: *Arranjo em ordem não decrescente*

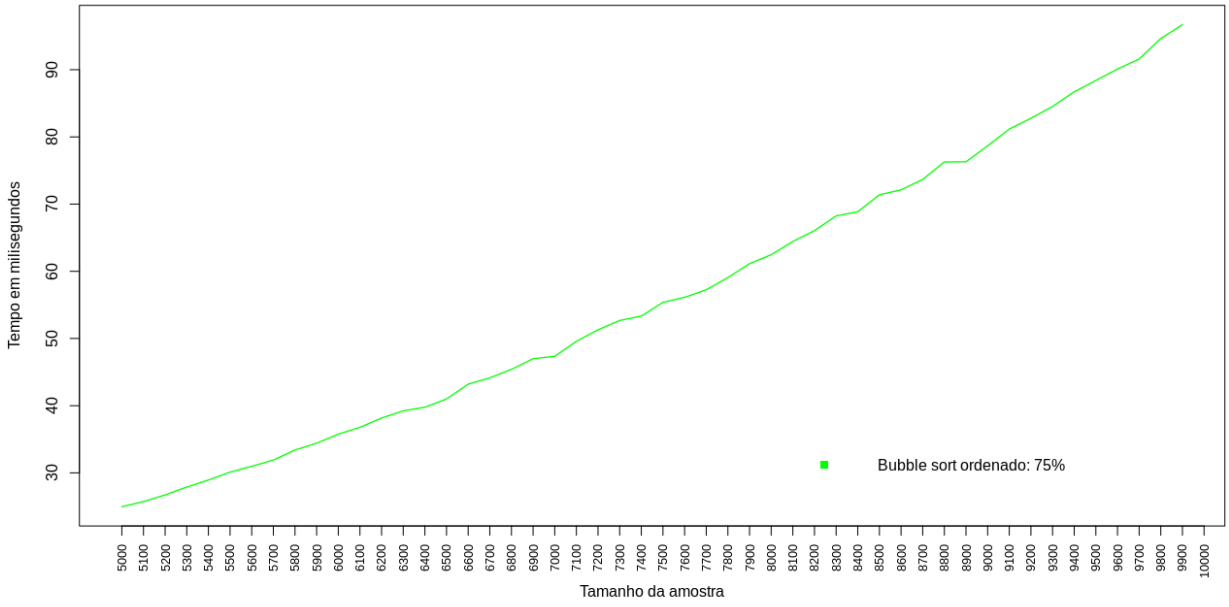


semelhante ao insertion, o bubble opera melhor em arranjos que tendem a ordem total.

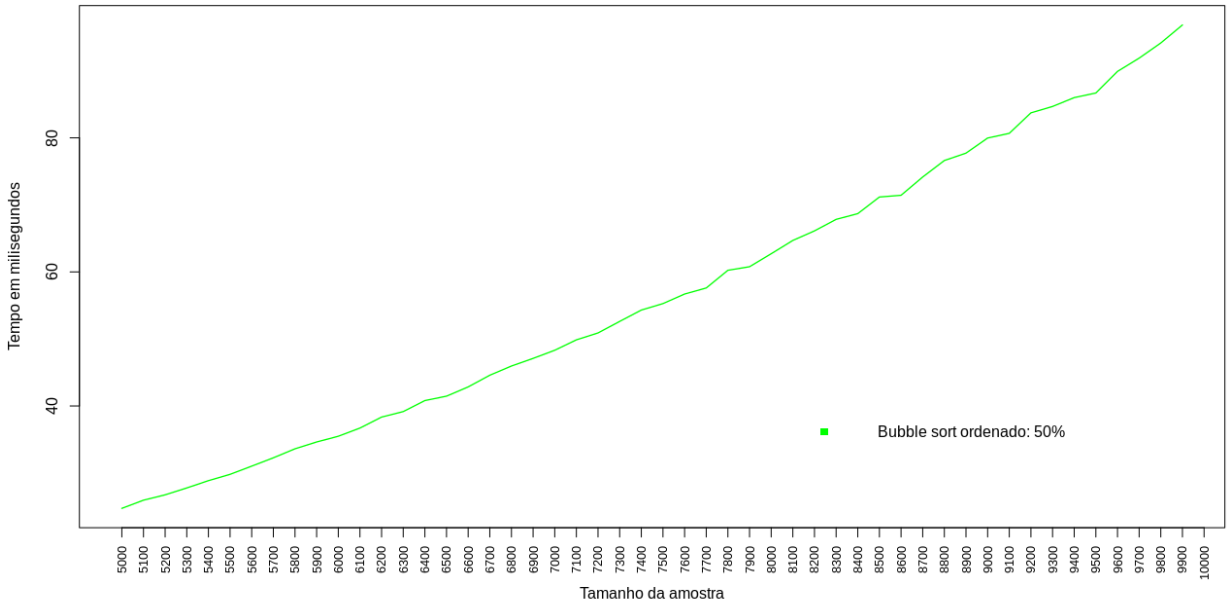
`Cenário`: *Arranjo em ordem não crescente*



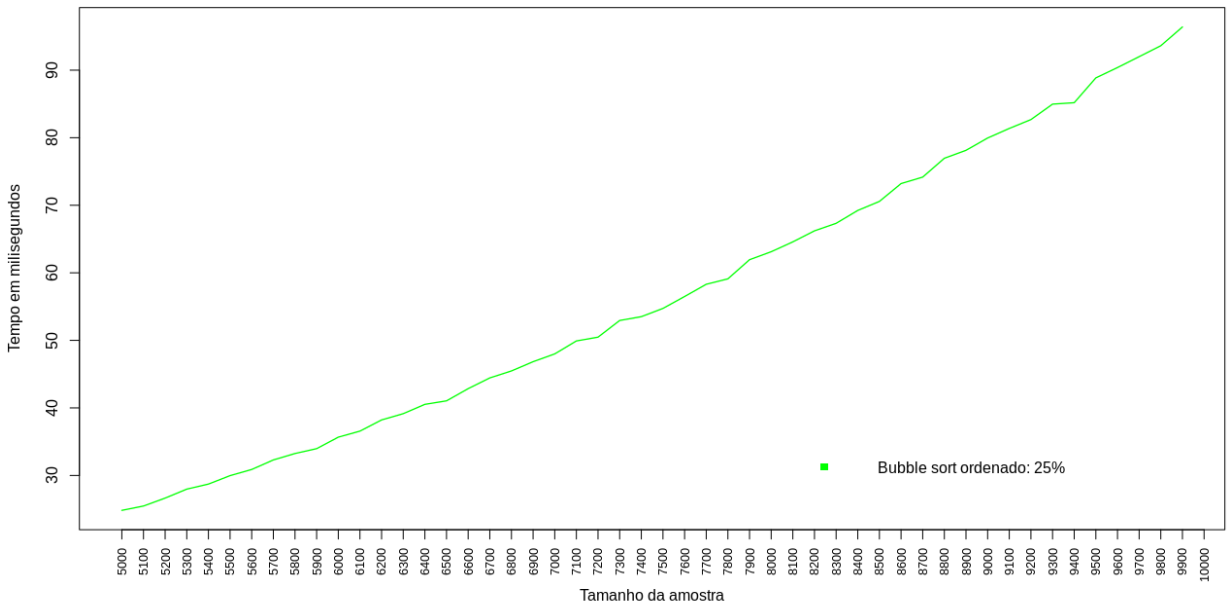
`Cenário`: *Arranjo 75% ordenado*



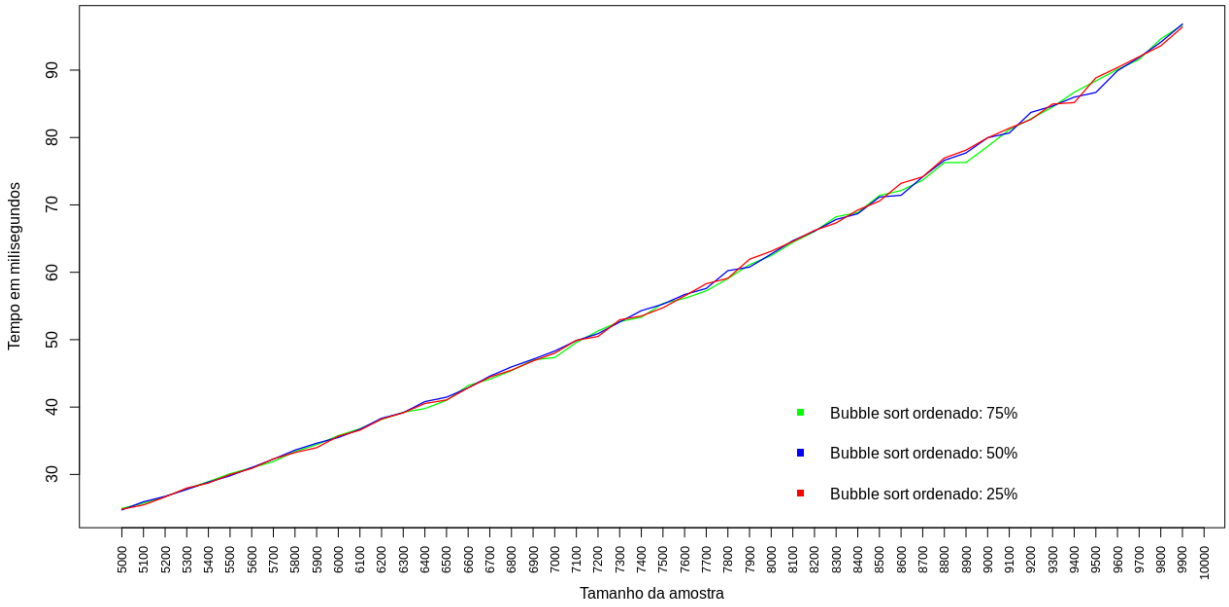
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*



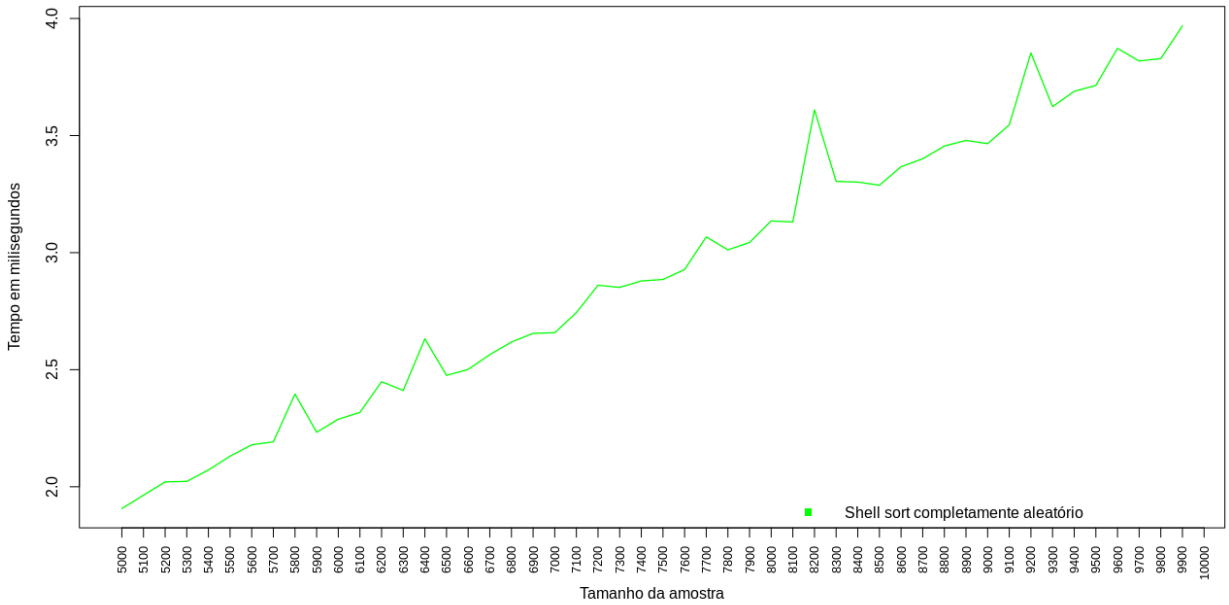
Shell Sort

- O método Shell: é uma extensão do algoritmo de ordenação por inserção. Ele permite a troca de posições distantes ums das outras, diferente do insertion sort que possui a troca de itens adjacentes para determinar o ponto de inserção.
- A complexidade do algoritmo é **desconhecida**, ninguém ainda foi capaz de encontrar uma fórmula fechada.
- O método não é estável.
- Os itens separados de `h` posições (itens distantes) são ordenados: o elemento na posição `x` é comparado e trocado (caso satisfaça a condição de ordenação) com o elemento na posição `x-h`. Este processo repete até `h=1`, quando esta condição é satisfeita o algoritmo é equivalente ao método de inserção.
- Funcionamento:

```
// Algoritmo de ordeção ShellSort
    public static void ordenaçãoShell(int[] v) {
        final int N = v.length;
        int incremento = N;
        do {
            incremento = incremento / 2;
            for (int k = 0; k < incremento; k++) {
                for (int i = incremento + k; i < N; i += incremento) {
                    int j = i;
                    while (j - incremento >= 0 && v[j] < v[j -
incremento]) {
                        int tmp = v[j];
```

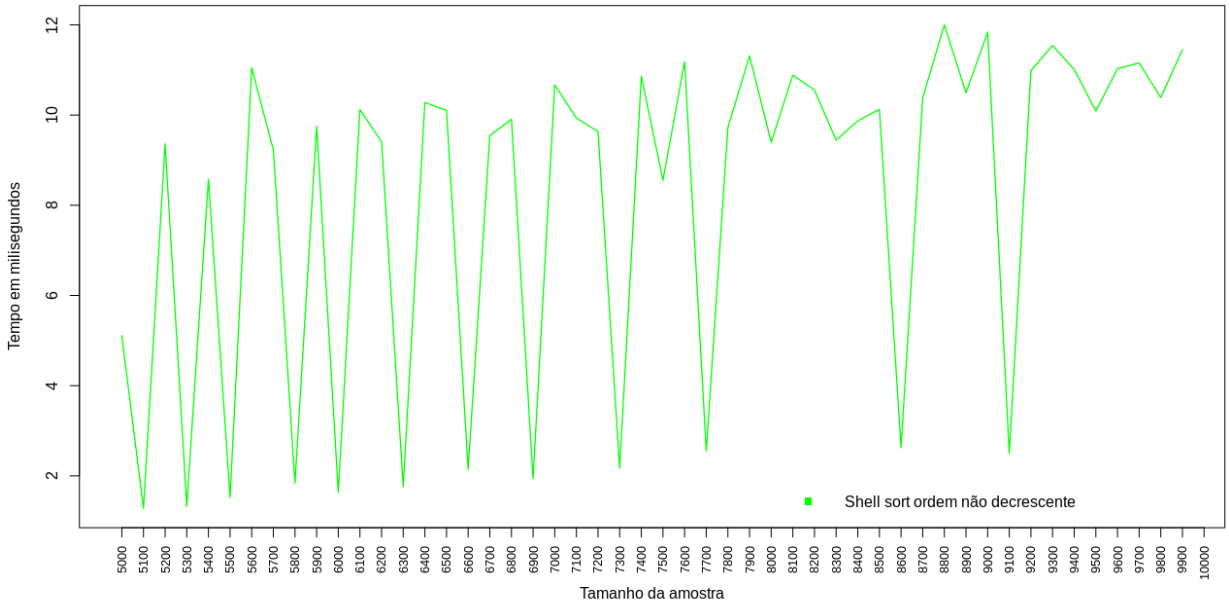
```
                        v[j] = v[j - incremento];
                        v[j - incremento] = tmp;
                        j -= incremento;
                    }
                }
            }
        } while (incremento > 1);
    }
```

`Cenário`: *Arranjo completamente aleatório*

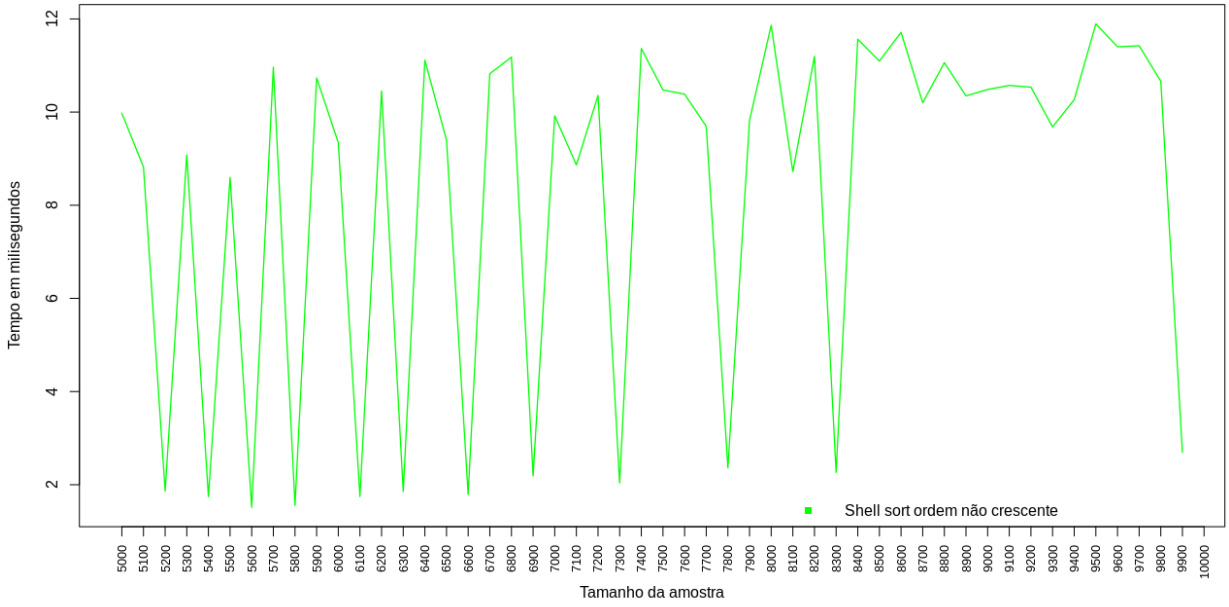


O melhor resultado nas análises foi em um arranjo sem ordem total, quando operado em alguma ordem parcial os resultados fogem do "padrão" dos algoritmos de ordenação.

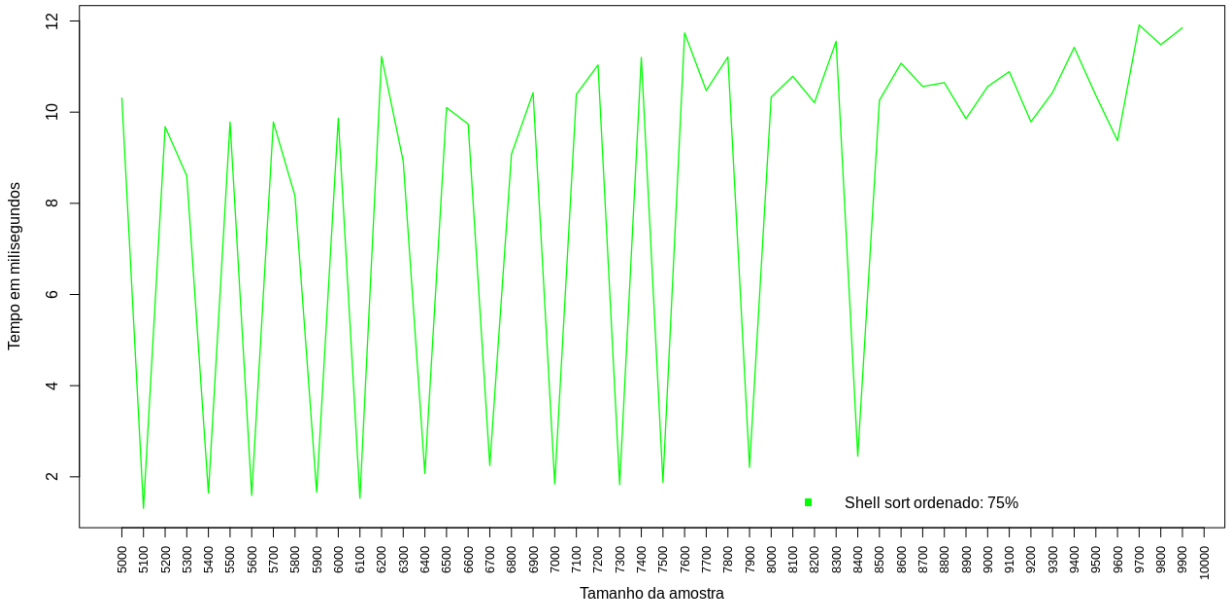
`Cenário`: *Arranjo em ordem não decrescente*



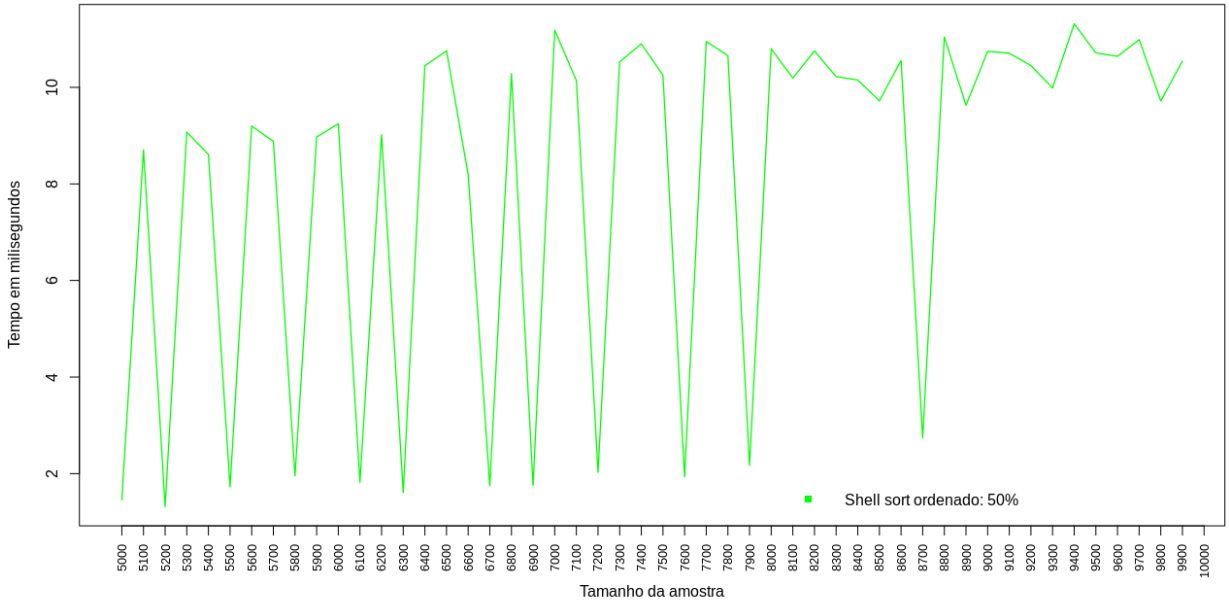
`Cenário`: *Arranjo em ordem não crescente*



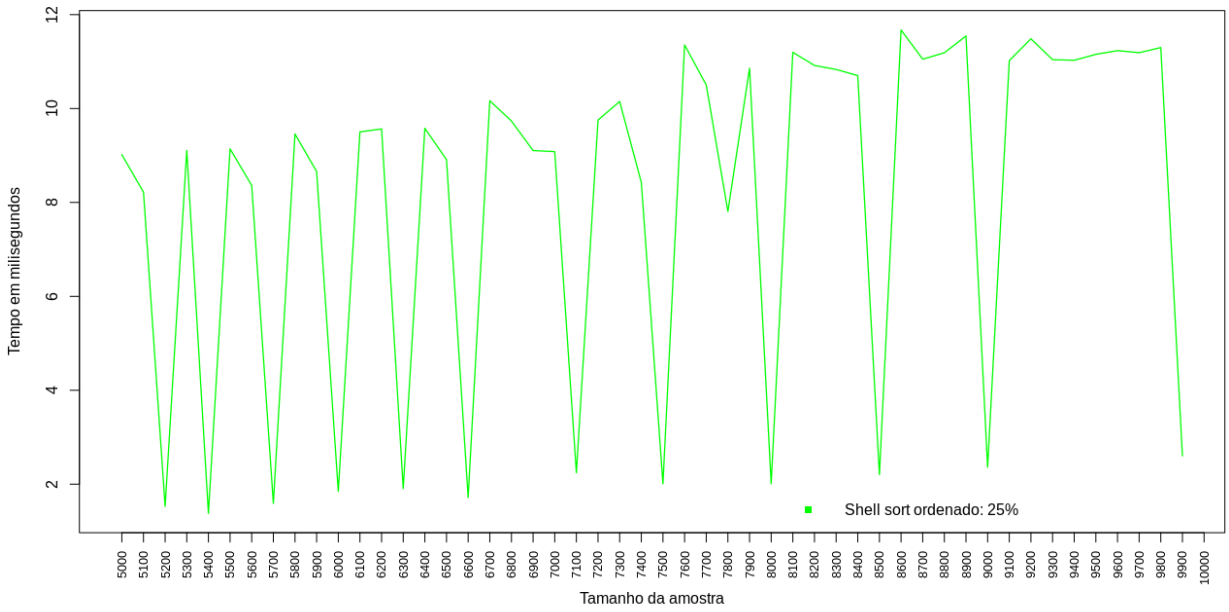
`Cenário`: *Arranjo 75% ordenado*



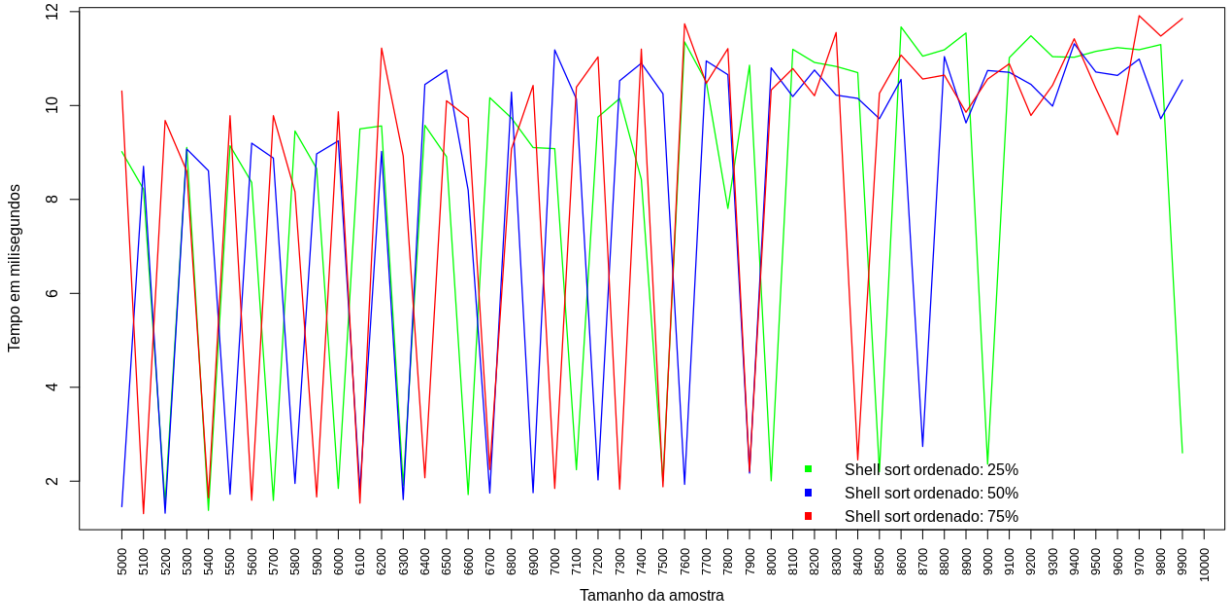
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*



Radix Sort

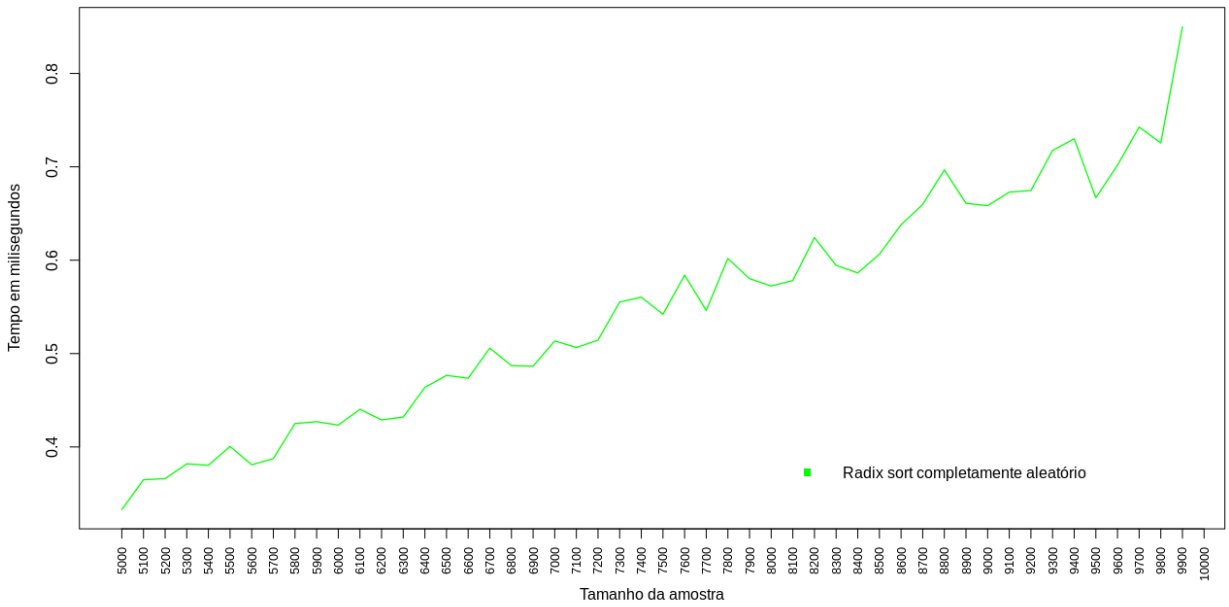
- Rápido e estável
- Pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou números.
- Ordena as chaves em qualquer ordem relacionada com a lexicografia

- Radix sort é um algoritmo que ordena inteiros processando dígitos individuais. **Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.**
- A versão implementada neste projeto é a LSD - (Least significant digit).
- O radix sort LSD começa do dígito menos significativo até o mais significativo, ordenando tipicamente da seguinte forma: chaves curtas vem antes de chaves longas, e chaves de mesmo tamanho são ordenadas lexicograficamente. **Isso coincide com a ordem normal de representação dos inteiros**, como a sequência "1, 2, 3, 4, 5, 6, 7, 8, 9, 10". Os valores processados pelo algoritmo de ordenação são frequentemente chamados de “chaves”, que podem existir por si próprias ou associadas a outros dados. As chaves podem ser strings de caracteres ou números.
- Funcionamento:

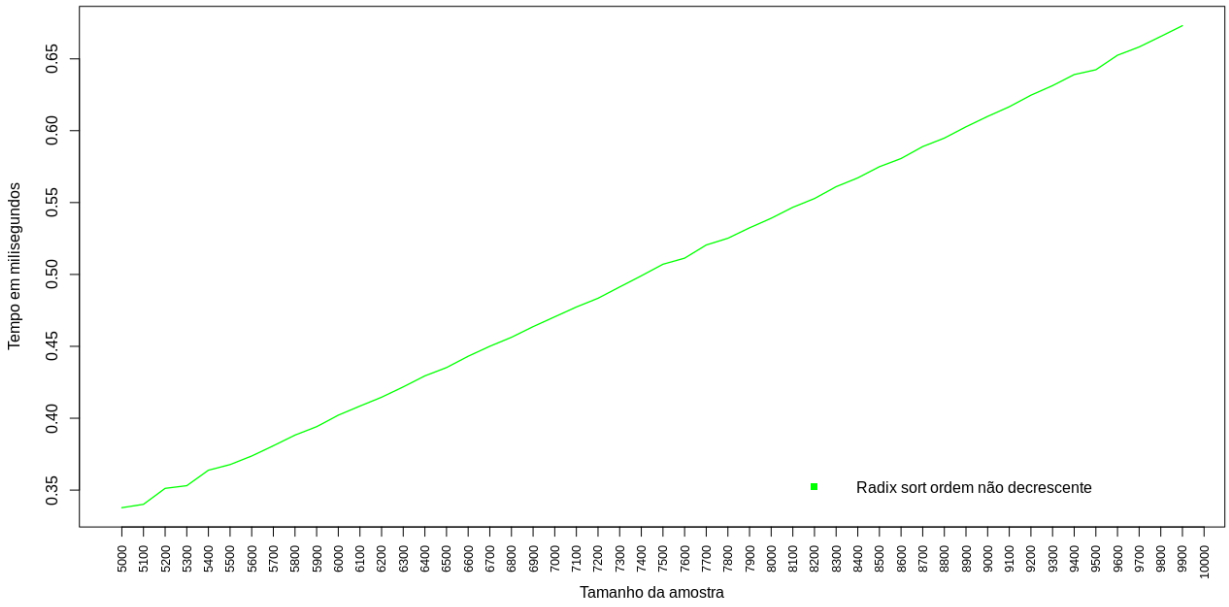
```
long int getMax( long int * first, long int * last, long int &count )
{
    long int maximum = *first;

    while( first < last )
    {
        if( maximum < *first )
        {
            maximum = *first;
            count++; // if
        }
        first++;
        count += 2; // while and increment.
    }

    return maximum;
}

void countingSort( long int * first, long int * last, long int digit, long
int &count )
{
    long int size = std::distance( first, last );

    long int * output = new long int [size];

    long int cnt[10] = { 0 };

    long int * left = first;

    cnt[0] = 0;

    for( int i = 0; i < size; i++ )
    {
```

```
        cnt[ (*left/digit)%10 ]++;
        left++;
        count += 3; // increments and loop.

    }

    for( int i = 1; i < 10; i++ )
    {
        cnt[i] += cnt[i -1];
        count++;
    }

    left = last;

    for( int i = size - 1; i >= 0; i-- )
    {
        left--;
        output[ cnt[ (*left/digit)%10 ] - 1 ] = *left;
        cnt[ (*left/digit)%10 ]--;
        count += 3;

    }

    left = first;

    for( int i = 0; i < size; i++ )
    {
        *left = output[i];
        left++;
        count += 2;
    }

    delete[] output;

}

void radix( long int * first, long int * last, long int &count )
{
    long int maximum = getMax( first, last, count );

    for( long int digit = 1; maximum/digit > 0; digit *= 10 )
    {
        countingSort( first, last, digit, count );
        count++;
    }

}
```

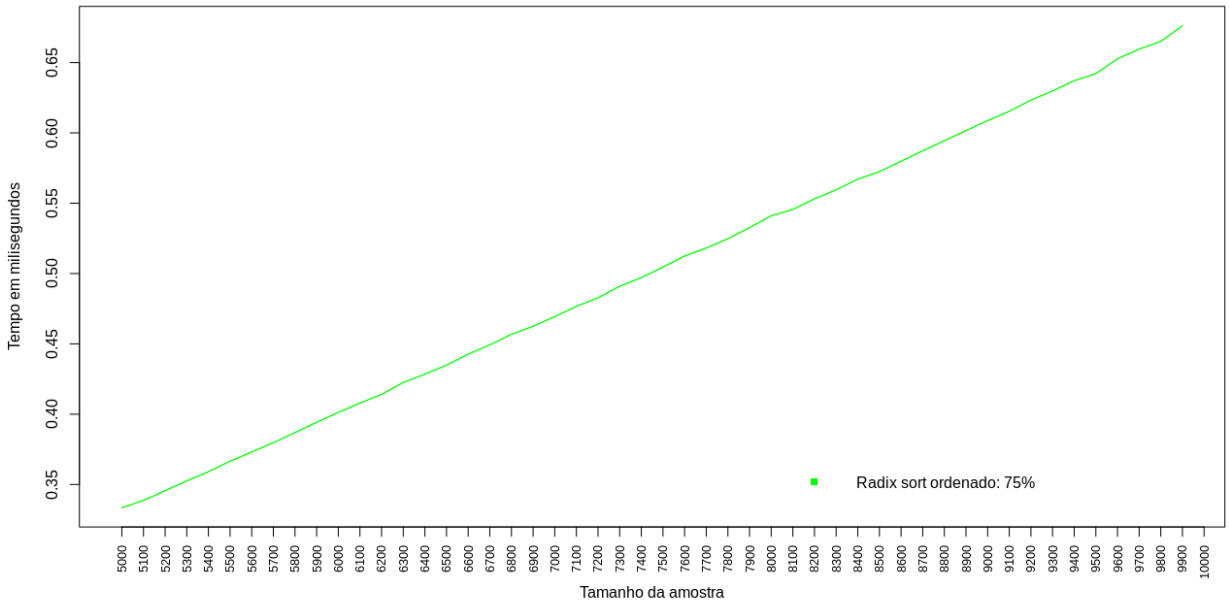
`Cenário`: *Arranjo completamente aleatório*



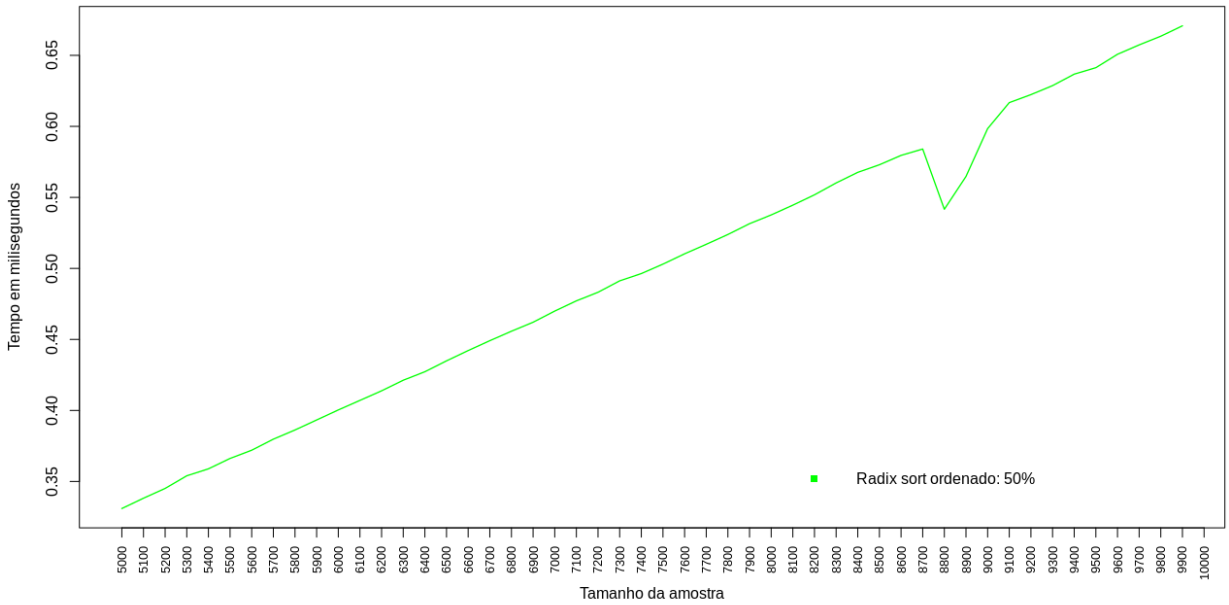
`Cenário`: *Arranjo em ordem não decrescente*



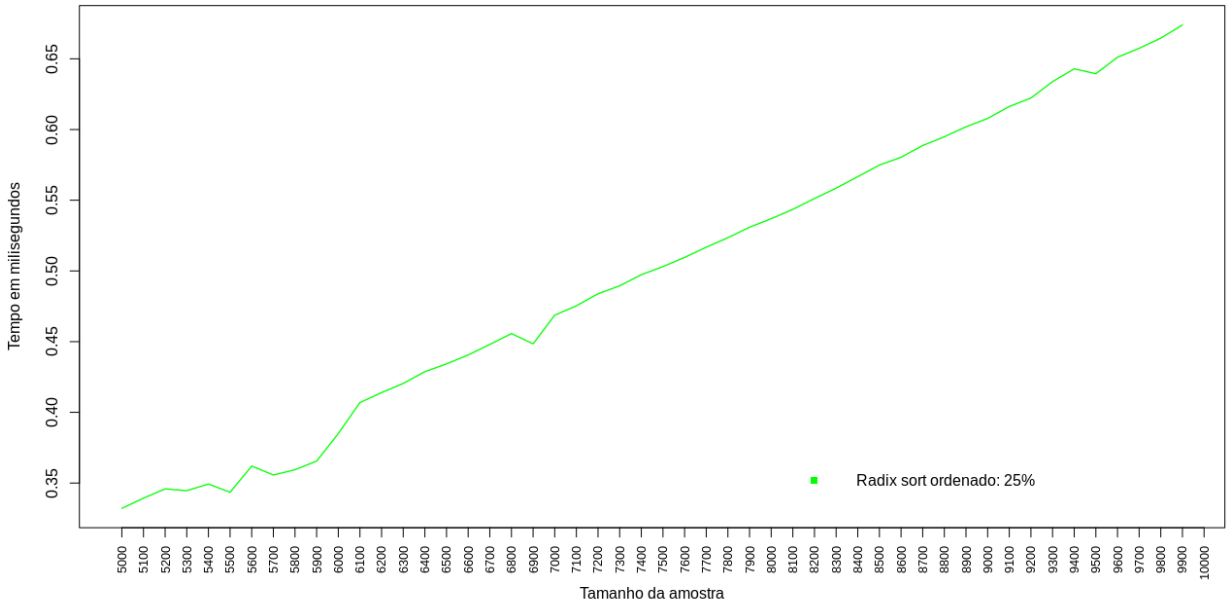
`Cenário`: *Arranjo 75% ordenado*



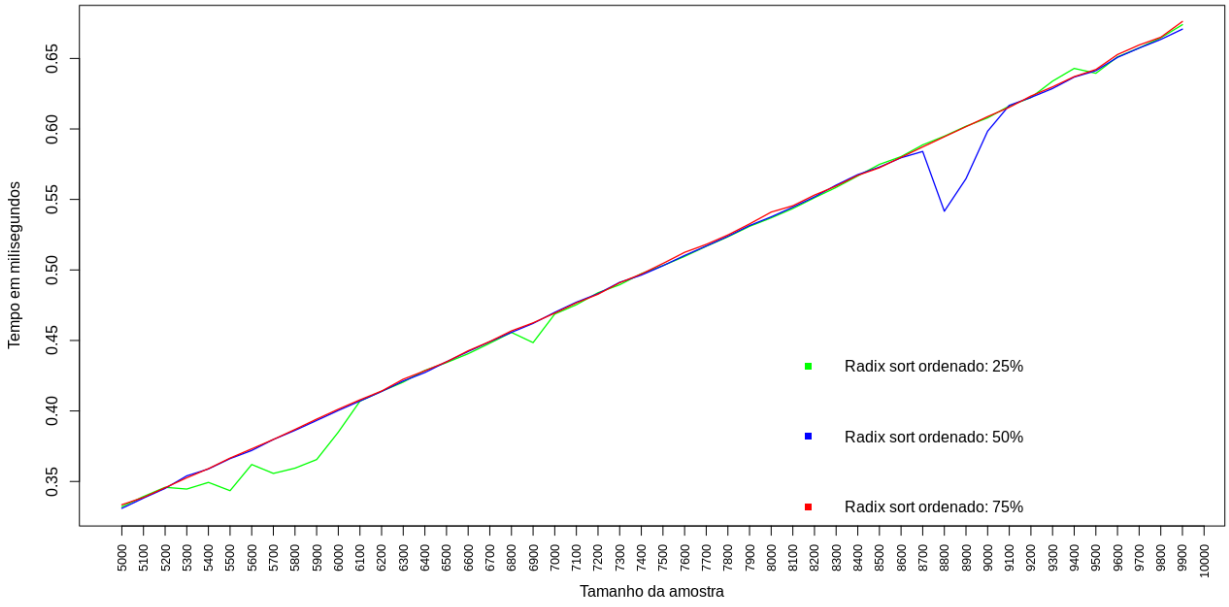
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*



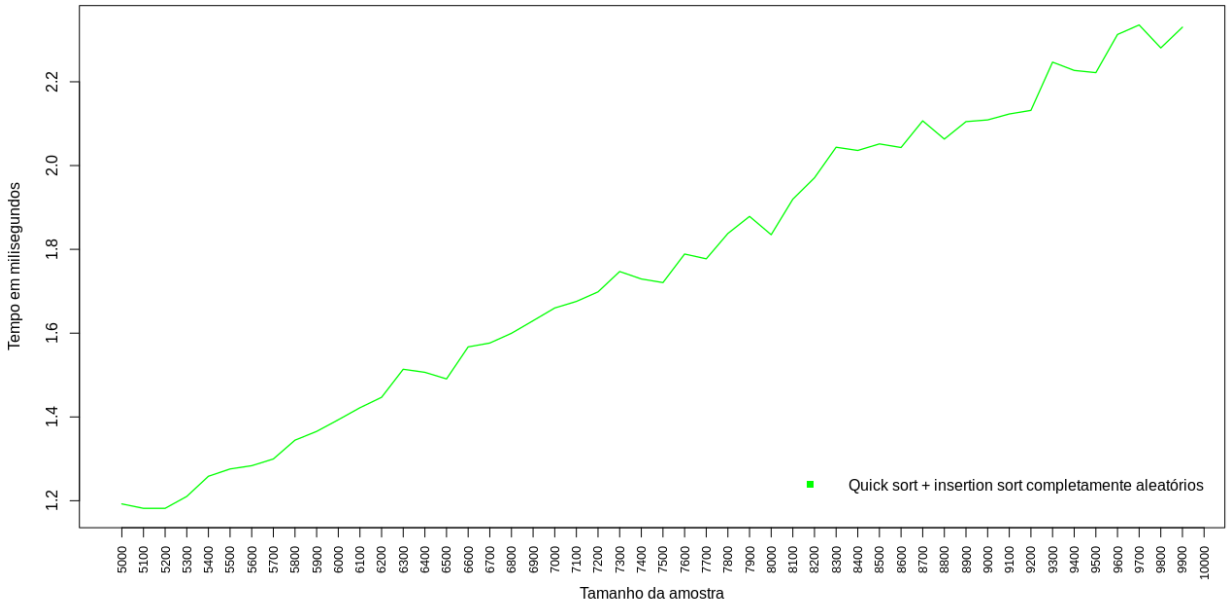
A eficiência do radix independe do cenário e do tamanho da amostra, visto que nenhuma medida do tempo de execução alcançou 1 ms.

Quick + Insertion

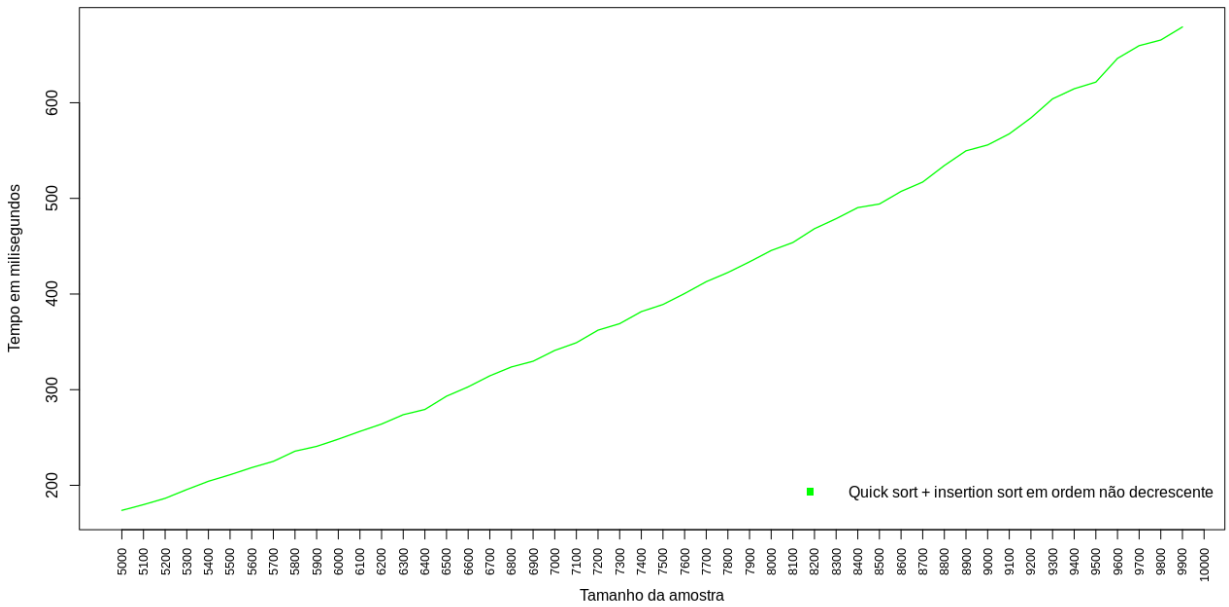
- O insertion é utilizado nas camadas mais baixas da recursão do quick quando o intervalo entre `first` e `last` é pequeno.
- Funcionamento:

```
void variable( long int * first, long int * last, long int &count )
{

    if( first == last )
    {
        count++; // if.
        return;
    }

    // inserindo o insertion em segmentos parcialmente ja ordenados pelo
partition.
    if( std::distance( first, last ) < 9 )
    {
        count++; // if
        insertion( first, last, count );
    }

    long int * pivot = partition( first, last, last - 1, count );
    variable( first, pivot, count);
    variable( pivot + 1, last, count );

}
```

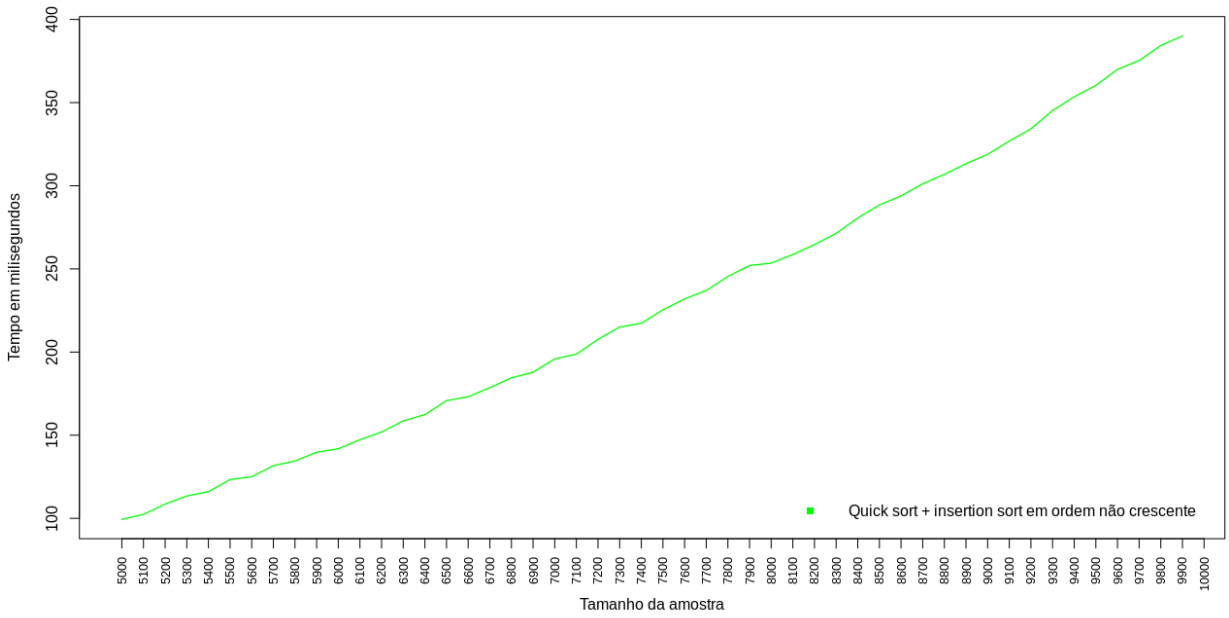
`Cenário`: *Arranjo completamente aleatório*



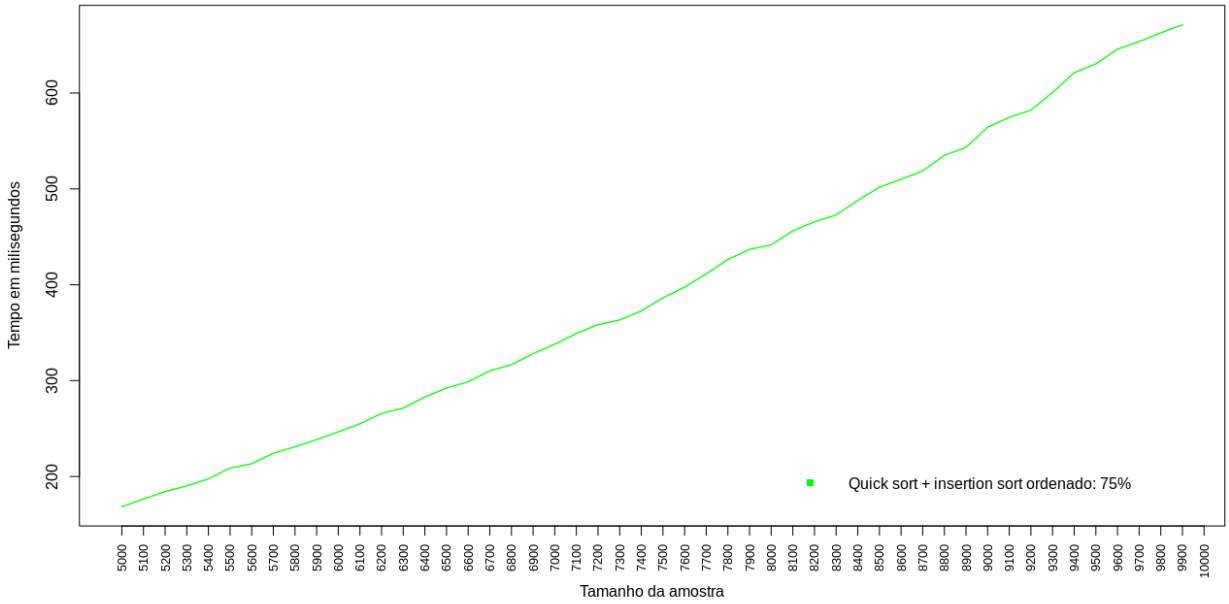
`Cenário`: *Arranjo em ordem não decrescente*



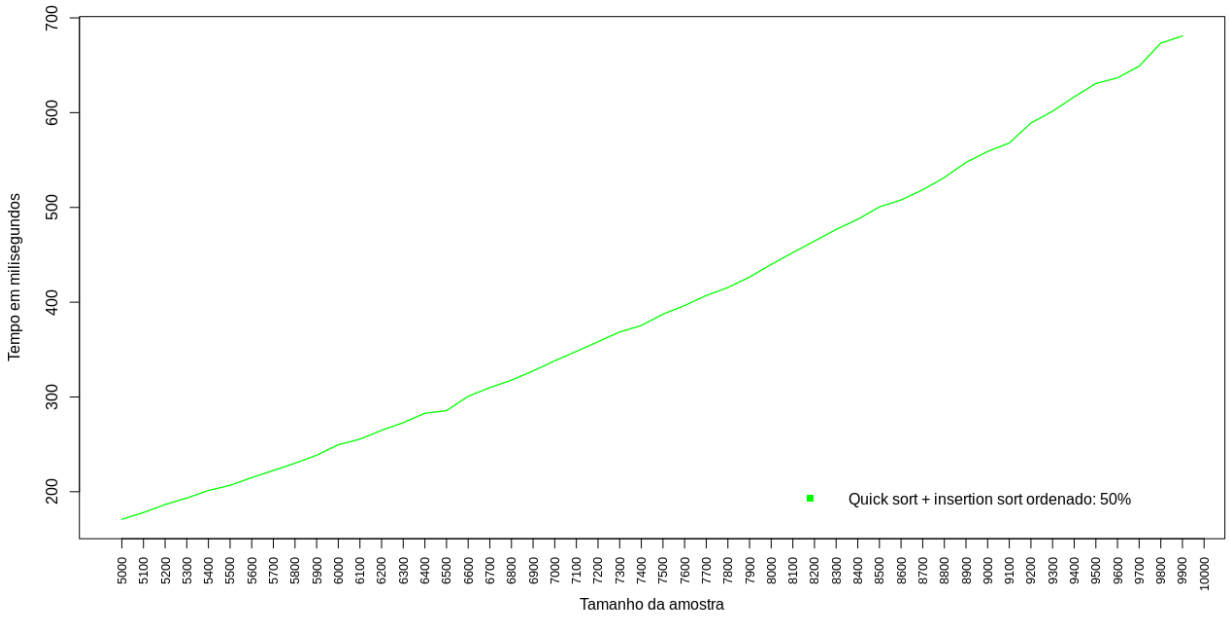
`Cenário`: *Arranjo em ordem não crescente*



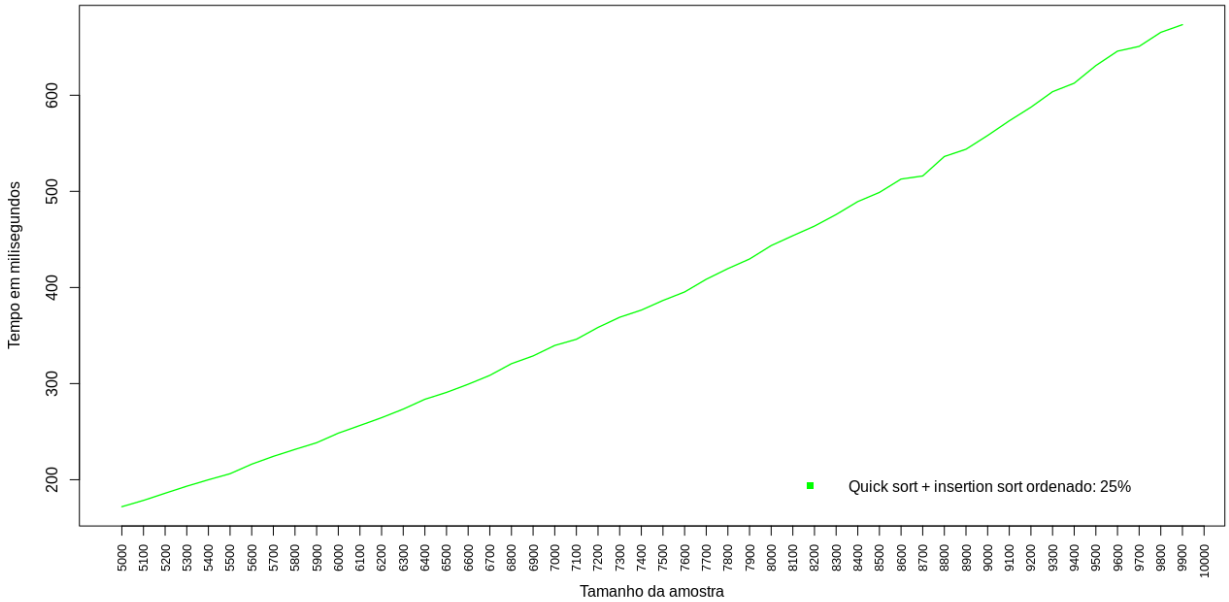
`Cenário`: *Arranjo 75% ordenado*



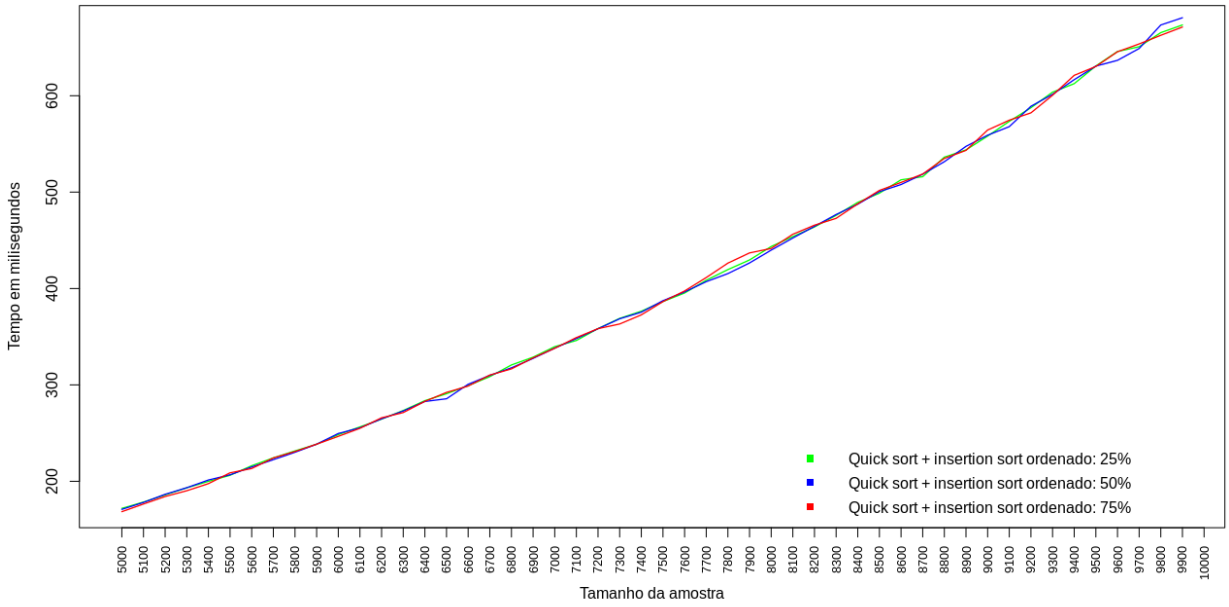
`Cenário`: *Arranjo 50% ordenado*



`Cenário`: *Arranjo 25% ordenado*



`Comparação`: *Arranjo parcialmente ordenado*


Tempo em milisegundos
200
300
400
500
600
Tamanho da amostra
5000 5100 5200 5300 5400 5500 5600 5700 5800 5900 6000 6100 6200 6300 6400 6500 6600 6700 6800 6900 7000 7100 7200 7300 7400 7500 7600 7700 7800 7900 8000 8100 8200 8300 8400 8500 8600 8700 8800 8900 9000 9100 9200 9300 9400 9500 9600 9700 9800 9900 10000
Quick sort + insertion sort ordenado: 25%
Quick sort + insertion sort ordenado: 50%
Quick sort + insertion sort ordenado: 75%

Resultados