

# Trabalho 3

Gustavo Hammerschmidt

# Seções

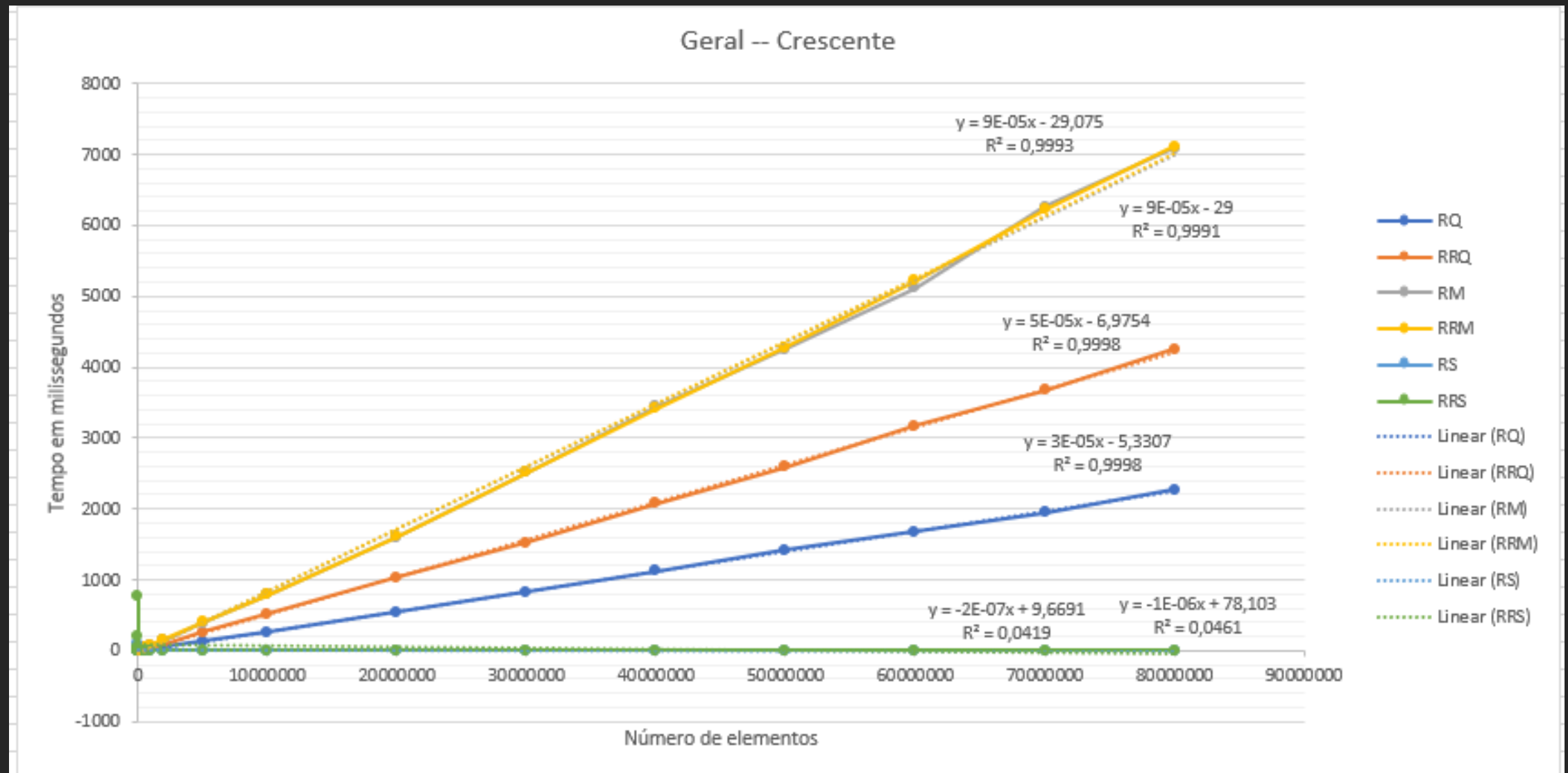
- 1.1 Dados
- 1.2 Geral
- 1.3 Recursivos vs Aleatórios e Recursivos
- 1.4 Comparação de pares
- 1.5 Previsões
- 2.1 Opiniões – Introdução
- 2.2 Opiniões – Respostas
- 3.1 Equações de Recorrência
- 3.2 Recorrência – Recursive Quicksort
- 3.3 Recorrência – Random Recursive Quicksort
- 3.4 Recorrência – Recursive Mergesort
- 3.5 Recorrência – Random Recursive Mergesort
- 3.6 Recorrência – Recursive Selectionsort
- 3.7 Recorrência – Random Recursive Selectionsort
- 4.1 Detalhes

RS and RRS hit StackOverflowError when element number is 20000.

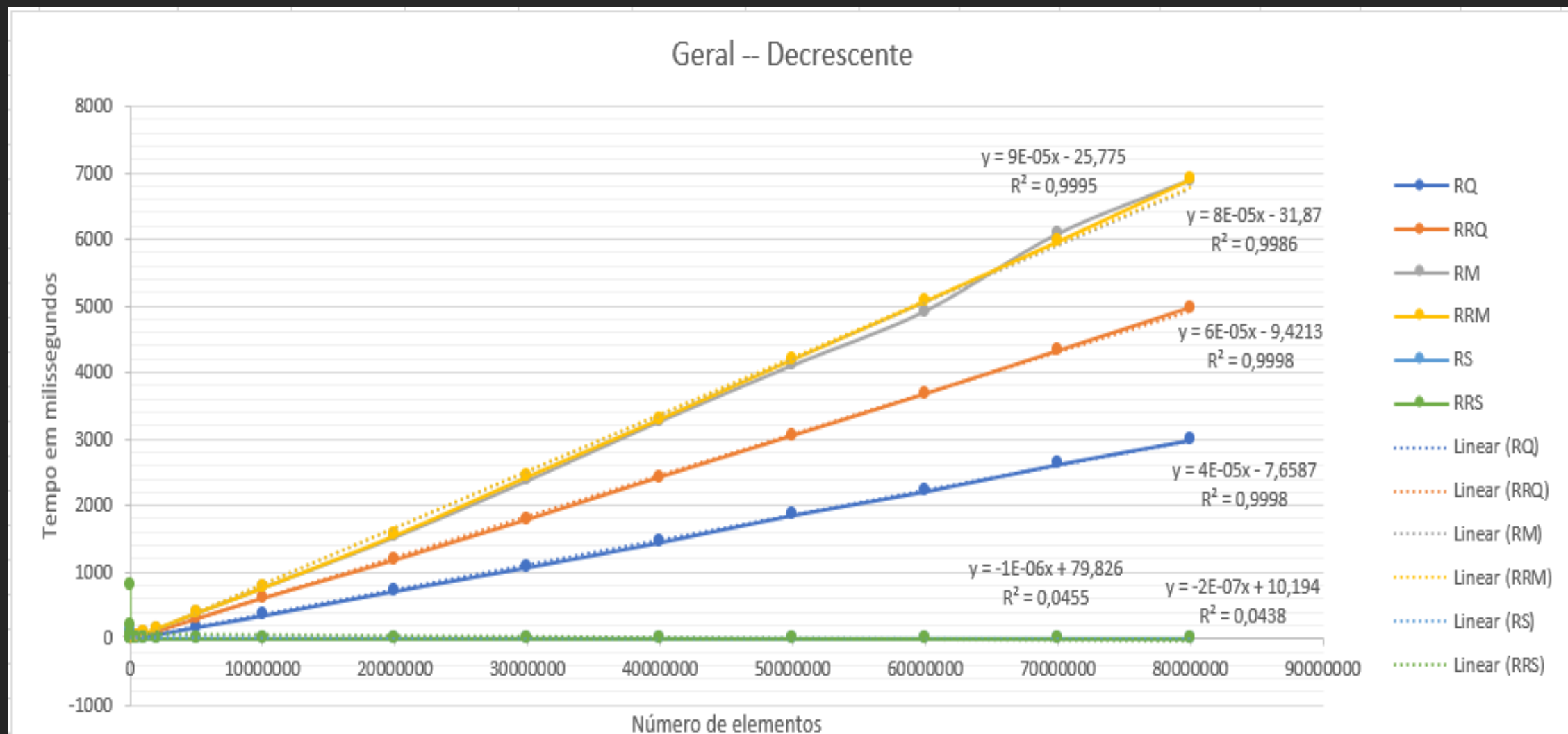


RS and RRS hit StackOverflowError  
when element number is 20000.

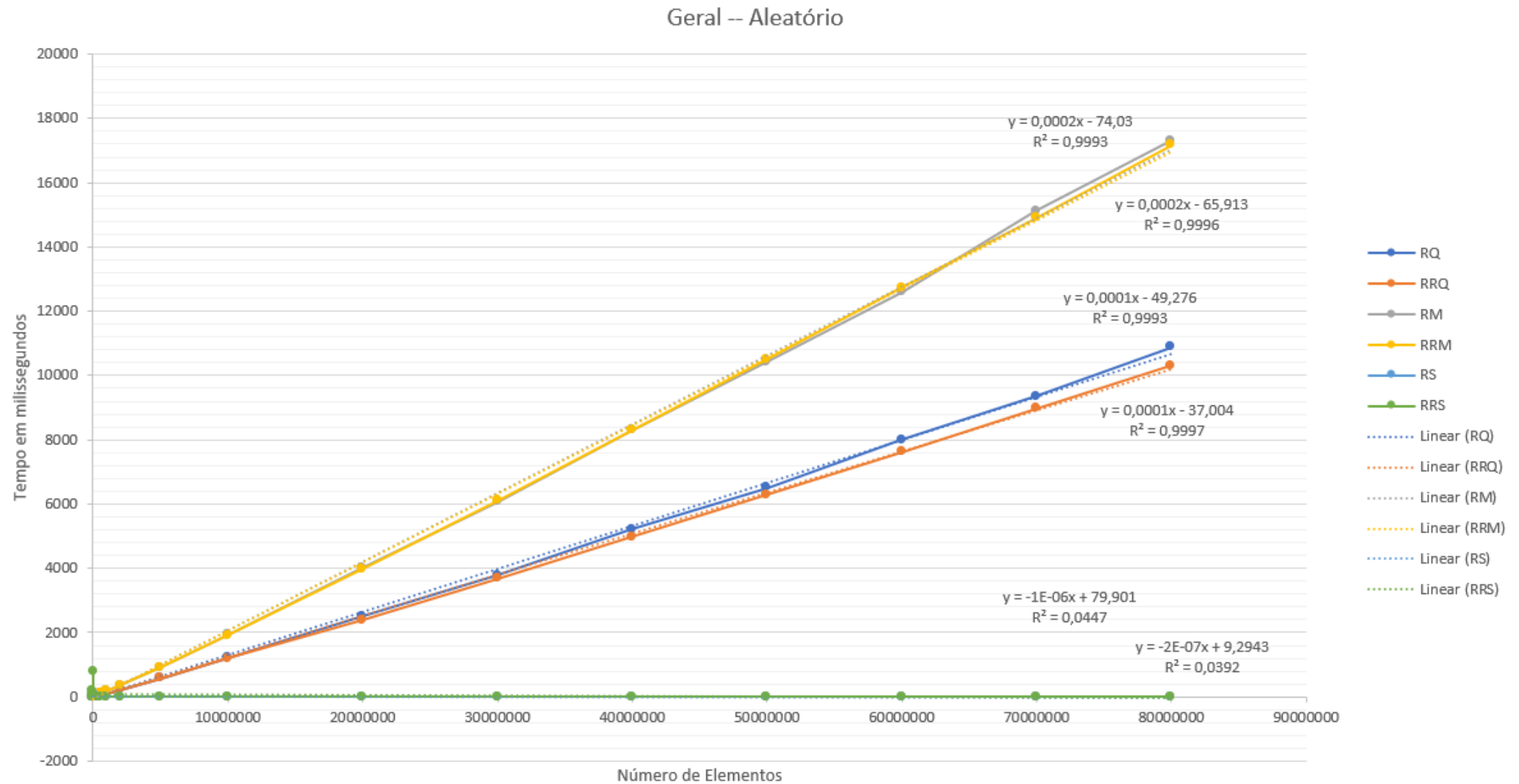
## 1.2.1 Geral



## 1.2.2 Geral



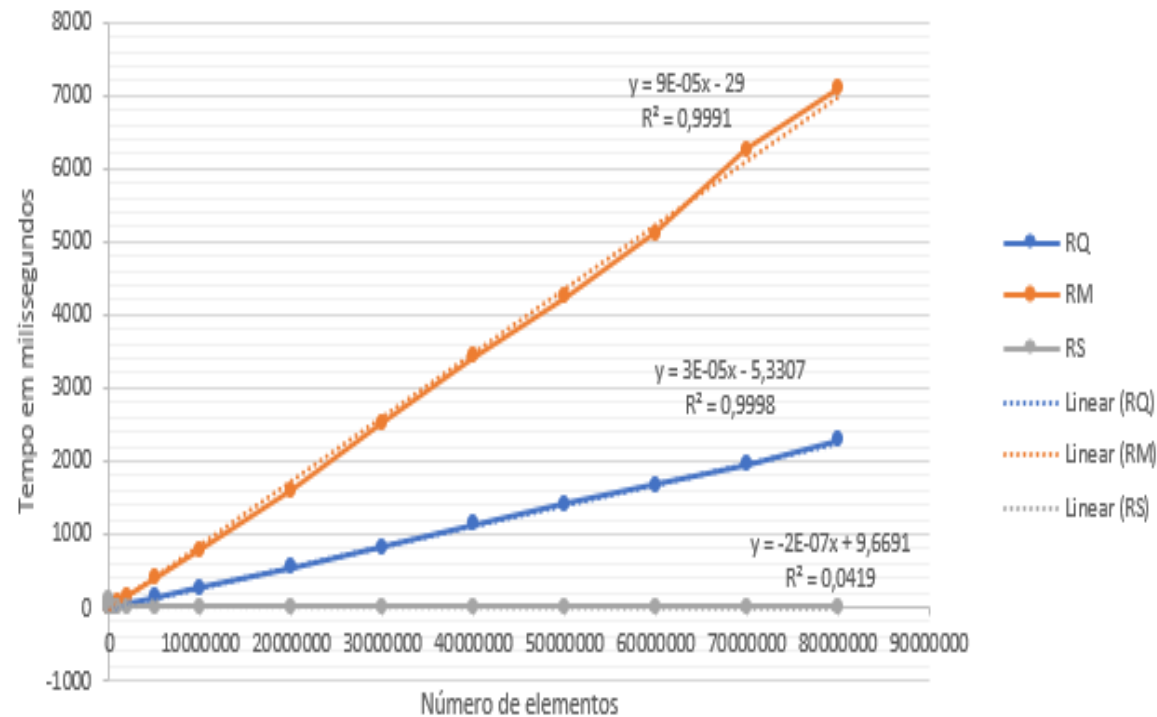
## 1.2.3 Geral



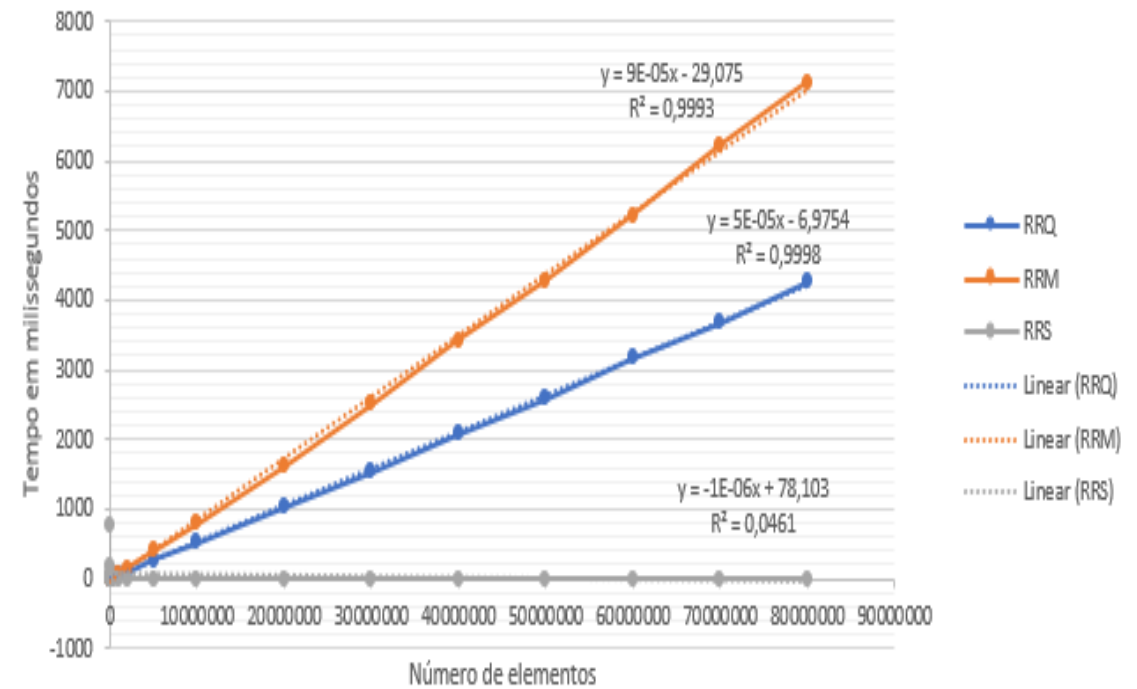


## 1.3.1 Recursivos vs Aleatórios e Recursivos

Comparação entre Recursivos -- Crescente

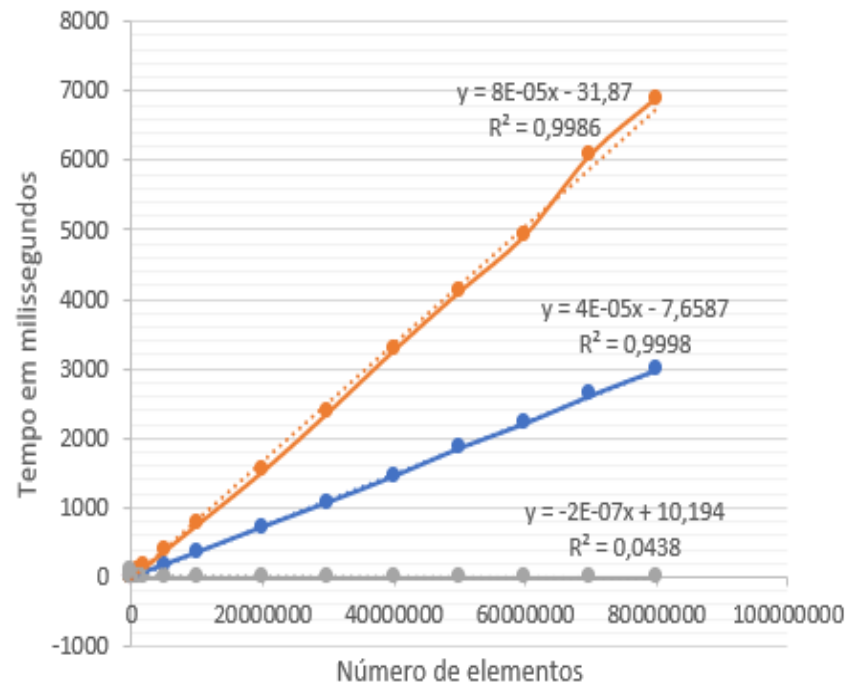


Comparação entre Aleatórios e Recursivos -- Crescente

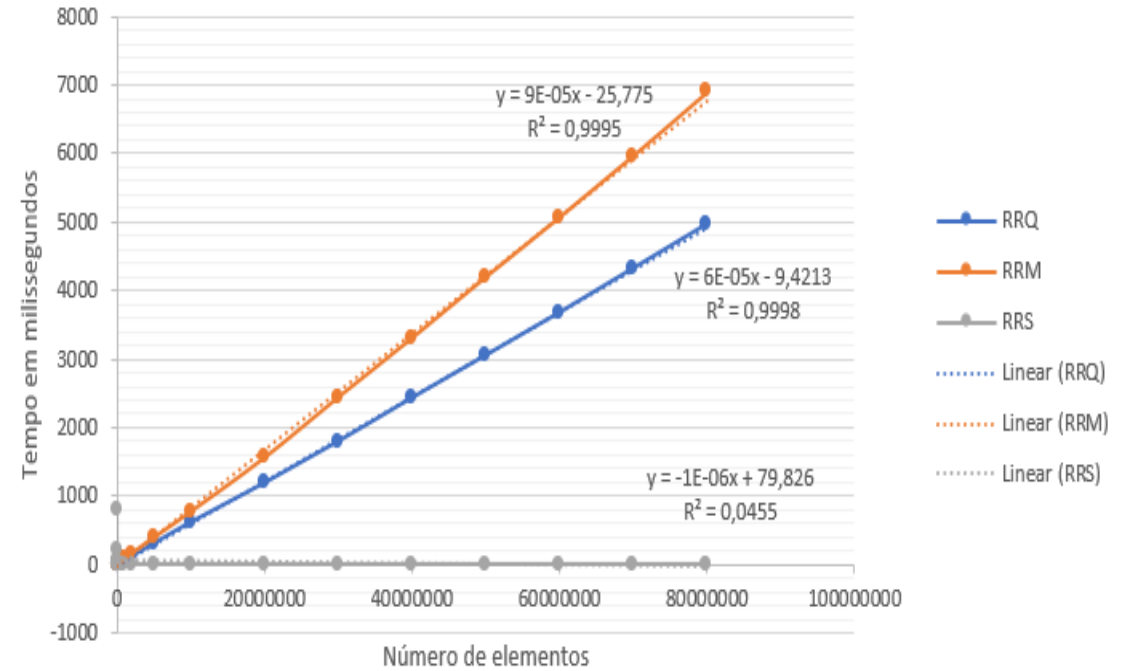


## 1.3.2 Recursivos vs Aleatórios e Recursivos

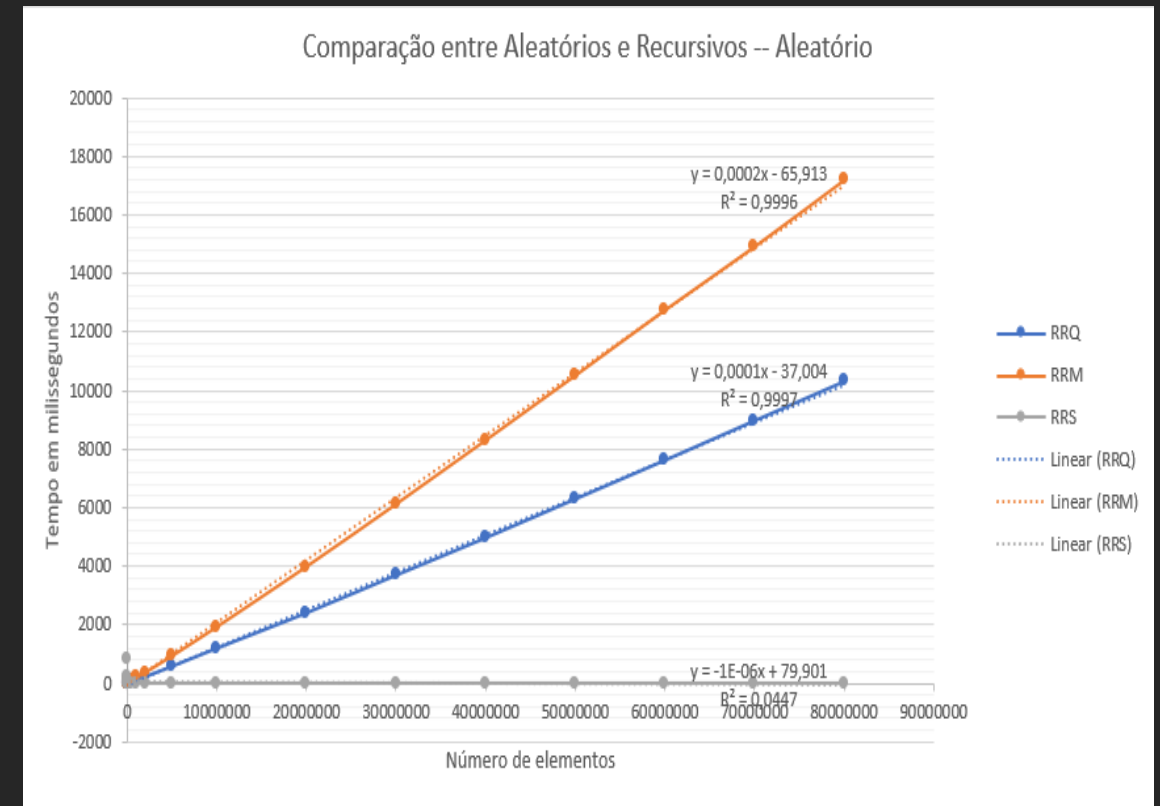
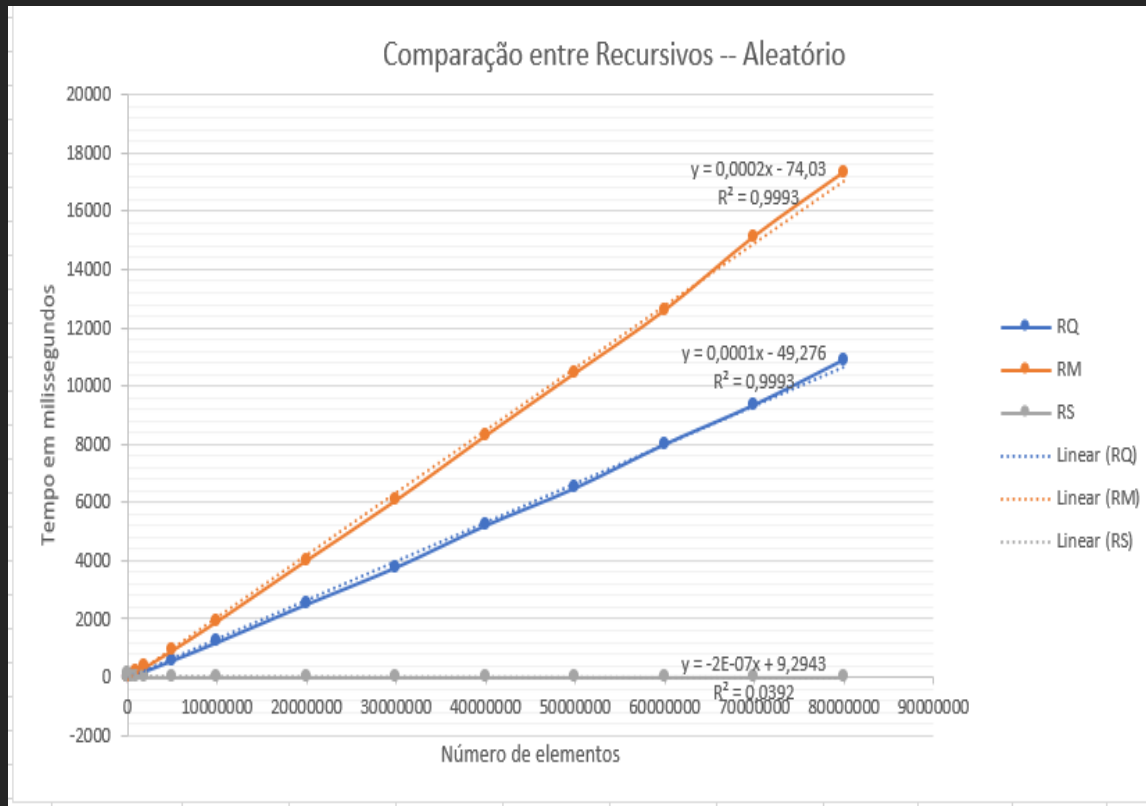
Comparação entre Recursivos -- Decrescente



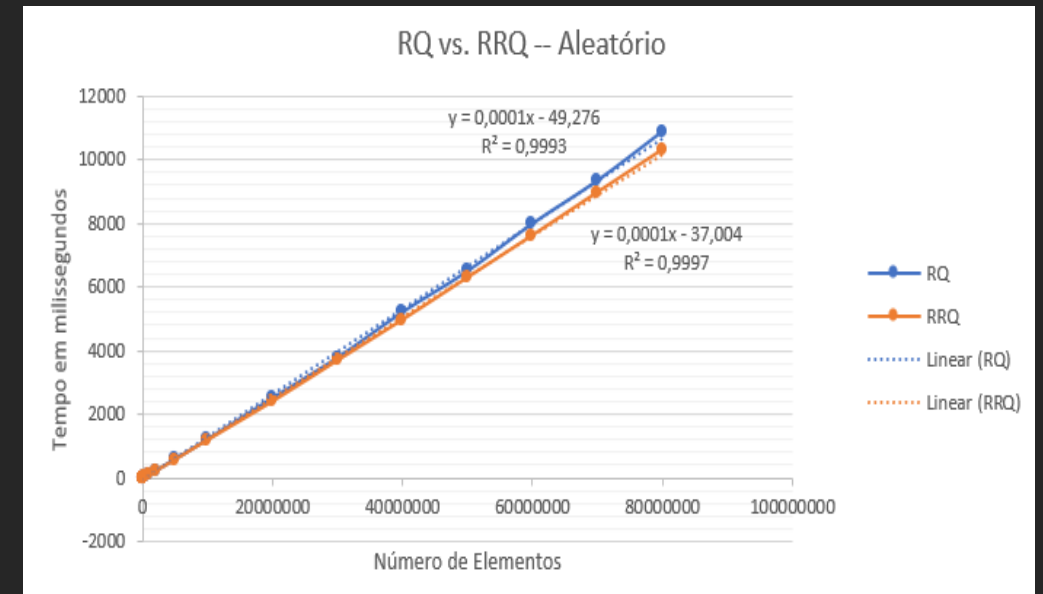
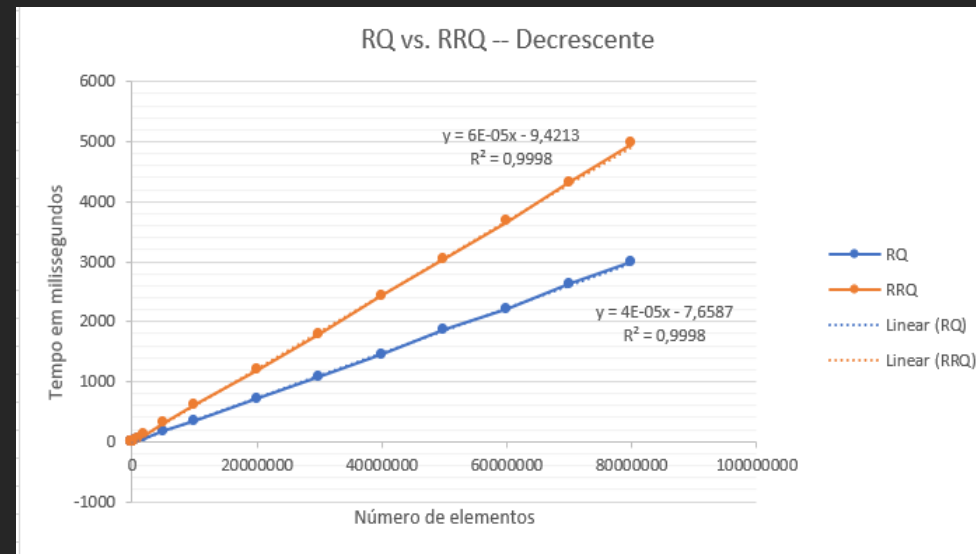
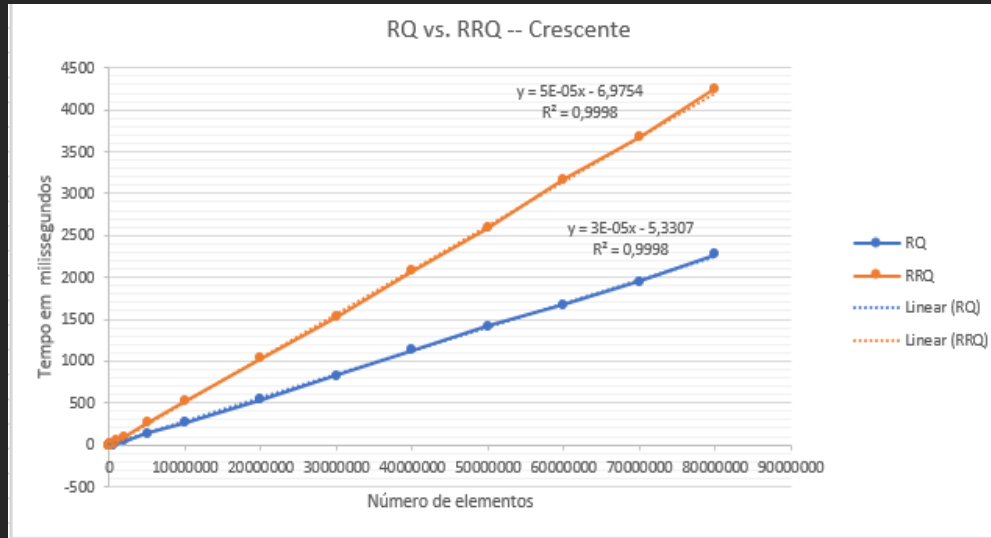
Comparação entre Aleatórios e Recursivos -- Decrescente



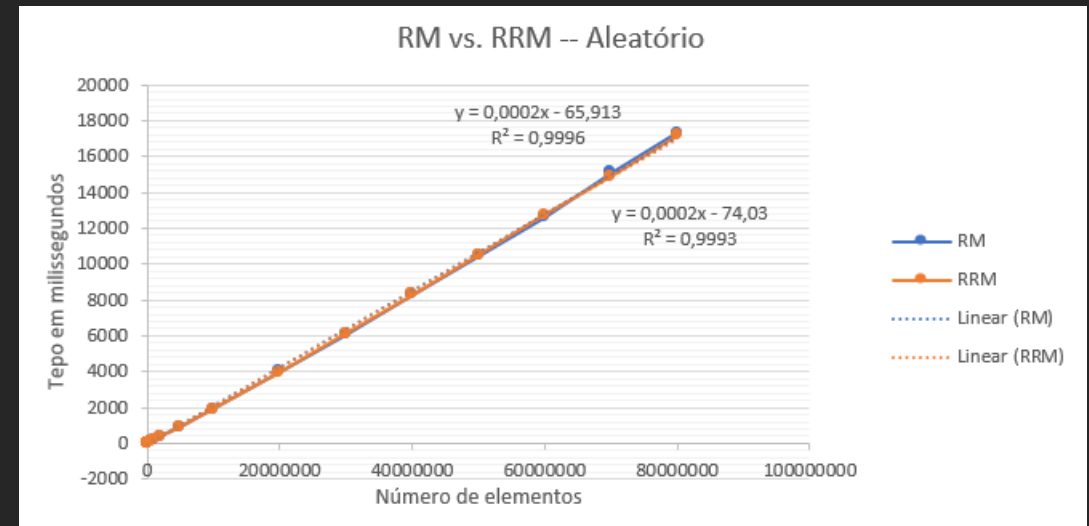
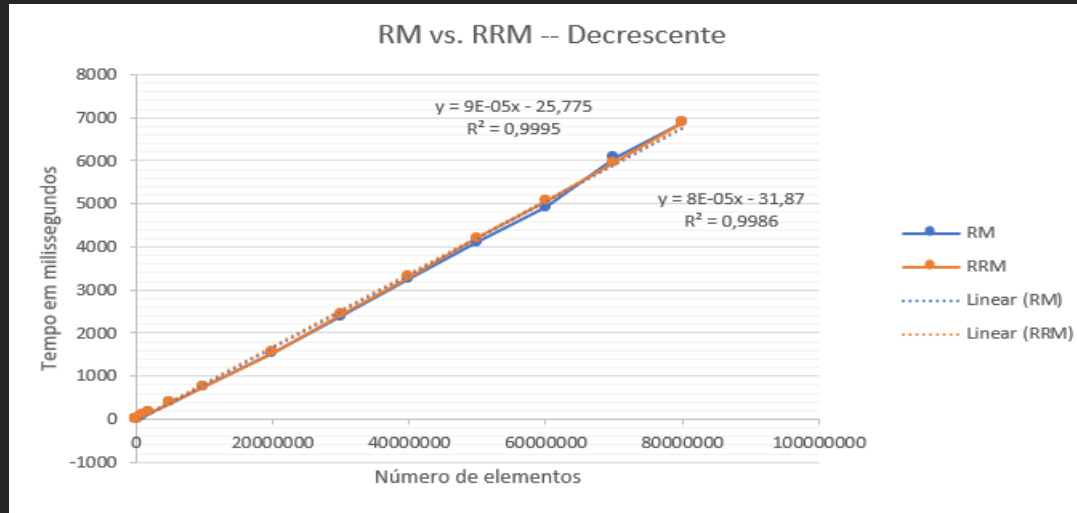
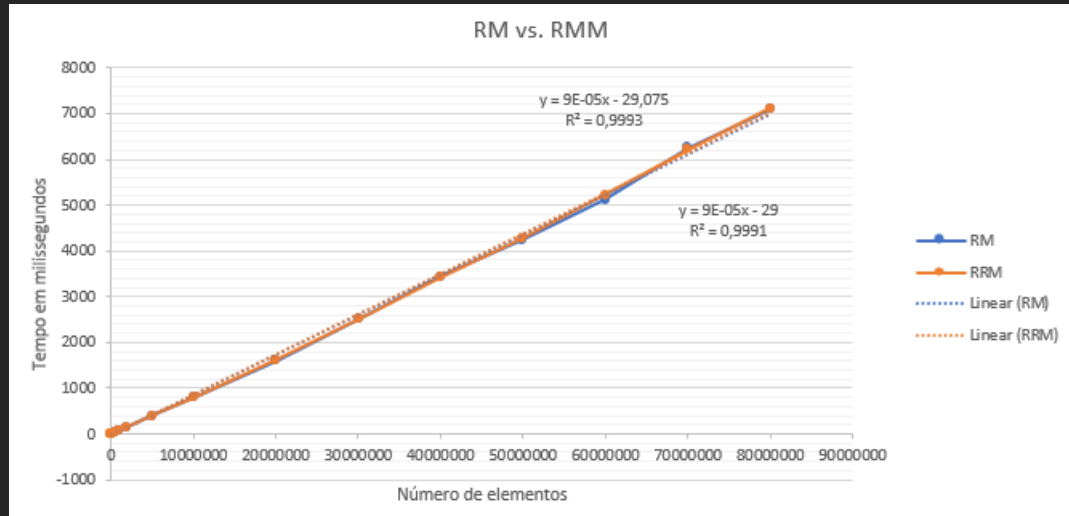
## 1.3.3 Recursivos vs Aleatórios e Recursivos



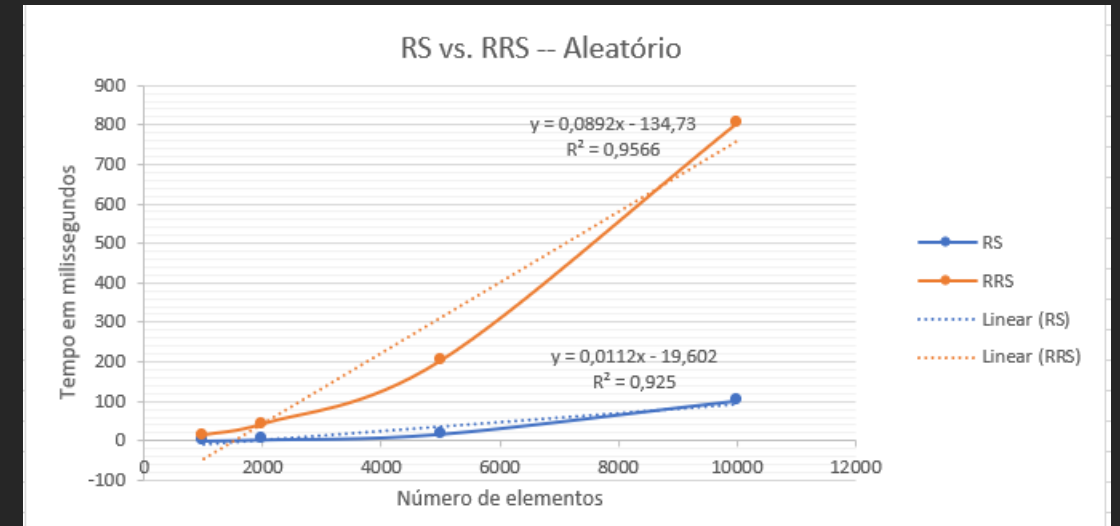
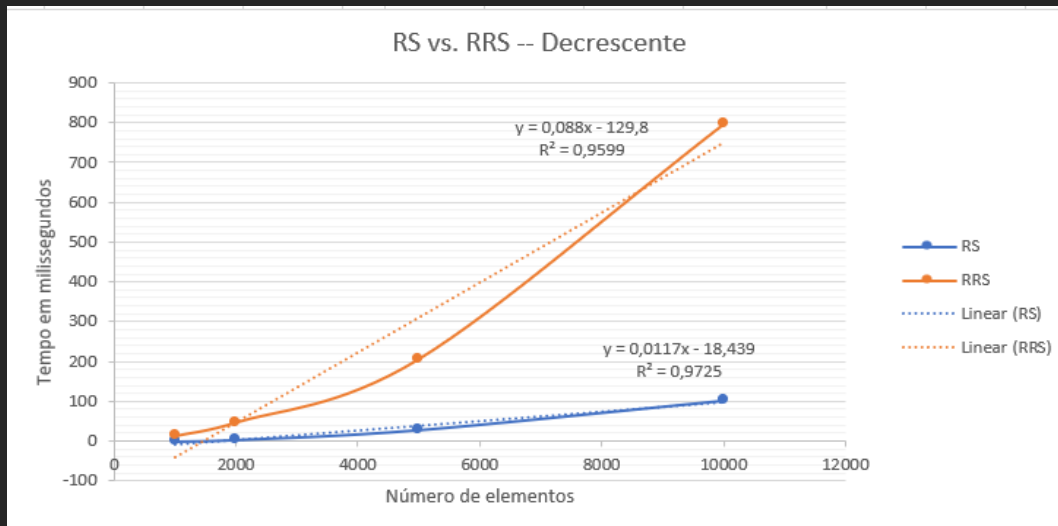
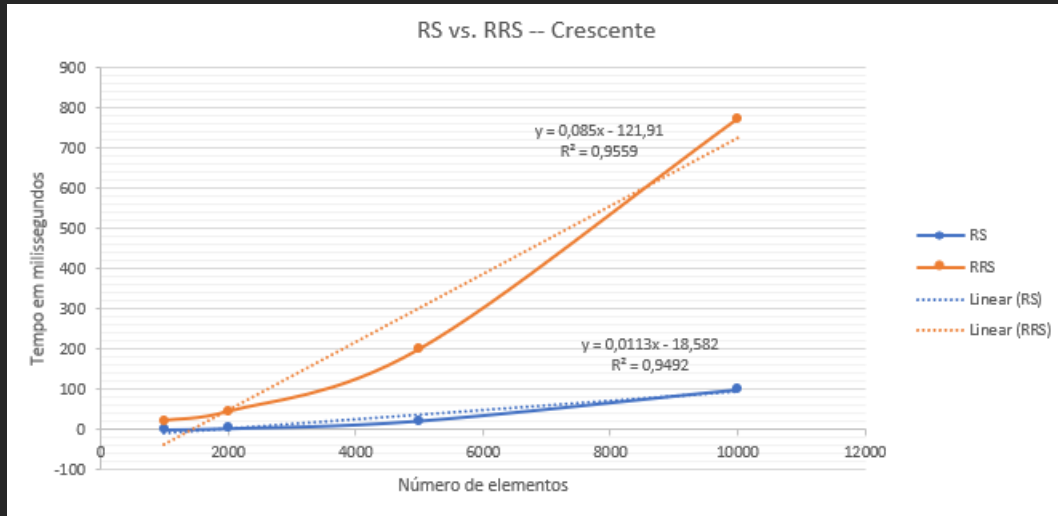
## 1.4.1 RQ vs RRQ



## 1.4.2 RM vs RRM



## 1.4.3 RS vs RRS



# 1.5 Previsões

Crescente

Previsões														
n_elems	RQ		RRQ		RM		RRM			n_elems	RS		RSS	
	ms	s	ms	s	ms	s	ms	s			ms	s	ms	s
80000000	2395	2	3993	4	7171	7	7171	7		10000	94	0	728,09	1
160000000	4795	5	7993	8	14371	14	14371	14		20000	207	0	1578,09	2
320000000	9595	10	15993	16	28771	29	28771	29		40000	433	0	3278,09	3
640000000	19195	19	31993	32	57571	58	57571	58		80000	885	1	6678,09	7
1280000000	38395	38	63993	64	115171	115	115171	115		160000	1789	2	13478,1	13

Decrescente

Previsões														
n_elems	RQ		RRQ		RM		RRM			n_elems	RS		RSS	
	ms	s	ms	s	ms	s	ms	s			ms	s	ms	s
80000000	3192	3	4791	5	6368	6	7174	7		10000	99	0	750,2	1
160000000	6392	6	9591	10	12768	13	14374	14		20000	216	0	1630,2	2
320000000	12792	13	19191	19	25568	26	28774	29		40000	450	0	3390,2	3
640000000	25592	26	38391	38	51168	51	57574	58		80000	918	1	6910,2	7
1280000000	51192	51	76791	77	102368	102	115174	115		160000	1854	2	13950,2	14

Aleatório

Previsões														
n_elems	RQ		RRQ		RM		RRM			n_elems	RS		RSS	
	ms	s	ms	s	ms	s	ms	s			ms	s	ms	s
80000000	7951	8	7963	8	15926	16	15934	16		10000	92	0	757	1
160000000	15951	16	15963	16	31926	32	31934	32		20000	204	0	1649	2
320000000	31951	32	31963	32	63926	64	63934	64		40000	428	0	3433	3
640000000	63951	64	63963	64	127926	128	127934	128		80000	876	1	7001	7
1280000000	127951	128	127963	128	255926	256	255934	256		160000	1772	2	14137	14

## 2.1 Opiniões - Introdução

### 1.3 Parte 03

Que conclusão a equipe obteve dos algoritmos randomizados ou aleatórios. Eles representam uma opção viável? Não faz nenhum sentido pensar nesta abordagem? Há alguma relação no tocante ao custo médio dos algoritmos randomizados experimentados? Etc.

Para apoiar as discussões e conclusões, a equipe poderá consultar a **seção “Ordenação: uma versão aleatória”**; esse texto foi retirado do Livro Texto (ref. no rodapé da página).



## 2.2.1 Opiniões - Respostas

Que conclusão a equipe obteve dos algoritmos randomizados ou aleatórios. Eles representam uma opção viável?

- No geral, fazendo a comparação entre os três tipos de arrays ordenados: crescente, decrescente e aleatório, a performance dos algoritmos aleatórios e recursivos em relação à dos somente recursivos é inferior.
- A performance dos algoritmos quicksort foi apenas melhor no array aleatório por uma pequena diferença, lembrando que a partição do algoritmo apenas recursivo ocorre na metade do array e do algoritmo recursivo e aleatório ocorre de forma aleatória. No caso do quicksort, a aleatoriedade faz sentido para arrays com muitos elementos devido a alta probabilidade de ser um vetor não ordenado, seja de forma crescente, seja de forma decrescente: os dois piores cenários para o algoritmo em comparação aos três avaliados.

## 2.2.2 Opiniões - Respostas

Que conclusão a equipe obteve dos algoritmos randomizados ou aleatórios. Eles representam uma opção viável?

- Entre os algoritmos Selectionsort (Select sort), Mergesort e quicksort, apenas para o algoritmo quicksort e em um cenário, a aleatoriedade fez uma diferença positiva. Isso se deve ao fato de o quicksort ser um algoritmo com o método dividir e conquistar, onde -- no caso do quicksort -- toda a parte de ordenação ocorre na divisão durante as chamadas recursivas e partições. No caso do mergesort, sabe-se que o algoritmo dividirá os arrays em conjuntos de dois elementos e fará o merge das menores partes até o todo com a fase do merge. Por isso, a aleatoriedade não implica uma diferença positiva, pois ela apenas modifica o ponto de divisão dos conjuntos de forma aleatória. Seria mais bem aplicada ao algoritmo se nele fossem incluídas as modificações de paralelismo e inclusão do algoritmo insertionsort para os conjuntos até 1000 elementos -- isso para um conjunto de elementos grande. Para o selectionsort, a aleatoriedade não serve, pois o algoritmo depende de uma ordem de ordenação; quando se usa o RRS, a chance de encontrar os menores valores já no começo da execução e ordená-los de forma crescente é muito baixa, ou seja, a aleatoriedade atrapalha a forma como algoritmo opera, fazendo com que os seus tempos tenham sido muito ruins comparados ao RS, mesmo para conjuntos pequenos.

## 2.2.3 Opiniões - Respostas

Não faz nenhum sentido pensar nesta abordagem?

- Só faz sentido pensar nessa abordagem quando os fatores seguintes estiverem presentes: (1) o conjunto de elementos é muito grande; (2) o algoritmo utiliza a aleatoriedade antes de ordenar os elementos: na fase de divisão ou organização de ponteiros ou na construção de um heap; (3) se o algoritmo pode ser paralelizado; e (4) se há a possibilidade de inserir outros algoritmos em certos estados da ordenação do array.

## 2.2.4 Opiniões - Respostas

Há alguma relação no tocante ao custo médio dos algoritmos randomizados experimentados? Etc.

- Sim, o tempo médio dos algoritmos aleatórios foi, proporcionalmente entre os três algoritmos, o de melhor performance comparado aos cenários crescente e decrescente.

## 3.1 Equações de Recorrência

### 1.4 Parte 04

Para cada algoritmo recursivo, que são: *quicksort*, *selectsort* e *mergesort*, escrever sua equação de recorrência. Aqui, nós pedimos que sejam incluídos todos os custos, ou seja, incluir na contabilização dos comandos executados.

## 3.2.1 Recorrência – Recursive Quicksort

```
// swap(...) -> C1 + ... + C3 = 3 = O(1)
//
private static void swap(int[] arr, int i, int j){

    int aux = arr[i];           // C1 <- 1
    arr[i] = arr[j];           // C2 <- 1
    arr[j] = aux;              // C3 <- 1
}
```

```
// partition(...) -> C1 + ... + C10 = 6 + 2n = O(n)
//
private static int partition(int[] arr, int p, int r){

    int x = (r+p)/2;             // C1 <- 1
    swap_r(arr, x, r);          // C2 <- 1

    x = arr[r];                 // C3 <- 1

    int i = p - 1;              // C4 <- 1

    for(int j = p; j < r; j++){  // C5 <- n/2

        if(arr[j] <= x){         // C6 <- n/2

            i++;                 // C7 <- n/2
            swap(arr, i, j);     // C8 <- n/2

        }

    }

    swap(arr, i+1, r);           // C9 <- 1

    return i + 1;               // C10 <- 1

}
```

## 3.2.2 Recorrência – Recursive Quicksort

```
// quicksort(...) -> C1 + ... + C4 = 1 + n + 2 * n * log n = O( n * log n )
//
private static void quicksort(int[] arr, int p, int r){

    if(p < r){                                // C1 <- 1

        int q = partition(arr, p, r);        // C2 <- n
        quicksort(arr, p, q - 1);            // C3 <- n * log n
        quicksort(arr, q + 1, r);            // C4 <- n * log n

    }

    // quicksort_wrapper(...) -> C1 + C2 = 1 + n * log n = O( n * log n )
    //
    public static void quicksort_wrapper(int[] arr){

        int N = arr.length - 1;              // C1 <- 1

        quicksort(arr, 0, N);                // C2 <- n * log n

    }

}
```

RM & RRM & RQ & RRQ

$$T(0) = 1$$

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





## 3.3.2 Recorrência – Random Recursive Quicksort

```
// quicksort_r(...) -> C1 + ... + C4 = 1 + n + 2 * n * log n = O( n * log n )
//
private static void quicksort_r(int[] arr, int p, int r){

    if(p < r){                                // C1 <- 1

        int q = partition_r(arr, p, r);      // C2 <- n
        quicksort_r(arr, p, q - 1);          // C3 <- n * log n
        quicksort_r(arr, q + 1, r);          // C4 <- n * log n

    }

    // quicksort_random_wrapper(...) -> C1 + C2 = 1 + n * log n = O( n * log n )
    //
    public static void quicksort_random_wrapper(int[] arr){

        int N = arr.length - 1;              // C1 <- 1

        quicksort_r(arr, 0, N);              // C2 <- n * log n

    }
}
```

RM & RRM & RQ & RRQ

$$T(0) = 1$$

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



## 3.4.2 Recorrência – Recursive Mergesort

```
// merge(...) -> C1 + ... + C5 = O( (13r + 3q - 19*p + 32) * log N )
//
private static void mergesort(int array[], int left, int right) {

    if (left < right) {                                // C1 <- 1

        int mid = (left + right) / 2;                  // C2 <- 1

        mergesort(array, left, mid);                   // C3 <- log n
        mergesort(array, left: mid + 1, right);        // C4 <- log n

        merge(array, left, mid, right);                // C5 <- 13r + 3q - 19*p + 28
    }
}
```

### 3.4.3 Recorrência – Recursive Mergesort

```
// mergesort_wrapper( n ) -> C1 ->  
//  
//      -> (13r + 3q - 19*p + 32) * log N  :: 13r - 13p + 13 == 13n  
//  
//      -> (3q - 6p + 19 + 13n) * log n  
//  
//      -> ( x + 13n) * log n :: x == (3q - 6p + 19) => q-p ~ log n & p ~ 0 + 19  
//  
//      -> ( x + 13n) * log n :: x == (log n + 19)  
//  
//      -> ( x + 13n) * log n :: n > log n > c ~ (13n + log n + 19)  
//  
//      -> ( x + 13n) * log n :: (x + 13n) = (13 n) :: x ~ 0  
//  
//      -> 13n * log n  
//  
//      -> O( 13n * log n)  
//
```

```
// mergesort_wrapper( n ) -> O( n * log n )  
public static void mergesort_wrapper(int[] arr){
```

```
    mergesort(arr, left: 0, right: arr.length-1);
```

```
}
```

```
// C1 <- (13r + 3q - 19*p + 32) * log N
```

RM & RRM & RQ & RRQ

$T(0) = 1$

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

## 3.5.1 Recorrência – Random Recursive Mergesort

```
// merge(...) -> C1 + ... + C5 = O( (13r + 3q - 19*p + 32) * log N )
//
private static void mergesort_random(int array[], int left, int right){

    if(left < right){                                     // C1 <- 1

        int mid = (r_generator.nextInt( bound: right - left) + left); // C2 <- 1

        mergesort_random(array, left, mid);               // C3 <- log n
        mergesort_random(array, left: mid + 1, right);    // C4 <- log n

        merge(array, left, mid, right);                   // C5 <- 13r + 3q - 19*p + 28

    }

}
```

## 3.5.2 Recorrência – Random Recursive Mergesort

```
// ... mergesort_random_wrapper(n) -> O( n * log n )
//
public static void mergesort_random_wrapper(int[] arr){

    mergesort_random(arr, left: 0, right: arr.length-1);    // C1 <- (13r + 3q - 19*p + 32) * log N

}
```

```
2
3  RM & RRM & RQ & RRQ
4  .....
5      T(0) = 1
6
7      T(n) = 2T(n/2) + n
8
9
```

## 3.6.1 Recorrência – Recursive Selectionsort

```
// selectionsort(...) -> C1 + ... + C8 = (2 + 3n) * n = 2n + 3n2 = O( n2 )  
//  
private static void selectionsort(int[] array, int index) {  
  
    if(index >= array.length - 1) { return; }           // C1 <- n  
  
    int k = index;                                     // C2 <- 1  
    for(int i = index + 1; i < array.length; i++ ) {   // C3 <- n-1  
        if (array[i] < array[k] ){ k = i; }             // C4 <- n-1  
    }  
  
    int temp = array[index];                           // C5 <- 1  
    array[index] = array[k];                           // C6 <- 1  
    array[k] = temp;                                   // C7 <- 1  
  
    selectionsort(array, index + 1);                   // C8 <- n * n  
}
```

## 3.6.2 Recorrência – Recursive Selectionsort

```
// selectionsort_wrapper(...) -> C1 = O( n2 )  
//  
public static void selectionsort_wrapper(int[] arr){  
    selectionsort(arr, index: 0);           // C1 <- n2  
}
```

10 RS & RSS

11

12  $T(0) = 1$

13

14  $T(n) = T(n-1) + n$

15



## 3.7.1 Recorrência – Random Recursive Selectionsort

```
// partition(...) -> C1 + ... + C15 = 3 + 6n = O( n )
//
private static int partition(int[] arr, int pivot_index){

    int i = 0;                                // C1 <- 1

    if(pivot_index != 0){                      // C2 <- 1

        int aux = arr[0];                     // C3 <- 1
        arr[0] = arr[pivot_index];           // C4 <- 1
        arr[pivot_index] = aux;               // C5 <- 1

    }

    for(int j = 0; j < (arr.length - 1); j++){ // C6 <- n-1

        if(arr[j+1] < arr[0]){                 // C7 <- n-1

            int aux = arr[j + 1];              // C8 <- n-1
            arr[j + 1] = arr[i + 1];           // C9 <- n-1
            arr[i + 1] = aux;                  // C10 <- n-1
            i++;                               // C11 <- n-1

        }

    }

}
```

```
    int aux = arr[0];                        // C12 <- 1
    arr[0] = arr[i];                          // C13 <- 1
    arr[i] = aux;                             // C14 <- 1

    return i;                                // C15 <- 1

}
```

## 3.7.2 Recorrência – Random Recursive Selectionsort

```
// selectionsort_random(...) -> C1 + ... + C13 = 9 + 4n = O( n )
//
private static int selectionsort_random(int[] arr, int k){

    if(arr.length == 1){ return arr[0]; }           // C1 <- 1
    else{

        int j = partition(arr, r_generator.nextInt(arr.length)); // C2 <- n

        int L[] = new int[j], R[] = new int[arr.length-j]; // C3 <- 2

        for (int i = 0; i < L.length; i++){ L[i] = arr[i]; } // C4 ~ n/2
        for (int m = 0; m < R.length; m++){ R[m] = arr[j+m]; } // C5 ~ n/2

                                                                    // ^ = n

        if(j == k){ // C6 <- 1
            return arr[j]; // C7 <- 1
        }
        else if(j > k){ // C8 <- 1

            R = null; // C9 <- 1
            return selectionsort_random(L, k); // C10 <- n

        }

    }

}
```

```
else{

    L = null; // C11 <- 1
    k = k - j - 1; // C12 <- 1
    return selectionsort_random(R, k); // C13 <- n

}

}
```





### 3.7.5 Recorrência – Random Recursive Selectionsort

```
// selectionsort_random_wrapper(...) -> C1 = n2 = O( n2 )  
//  
public static void selectionsort_random_wrapper(int[] arr){  
    selectionsort_control(arr, contador: 0, new int[]{});    // C1 <- n2  
}
```

10 RS & RSS

11

12

13

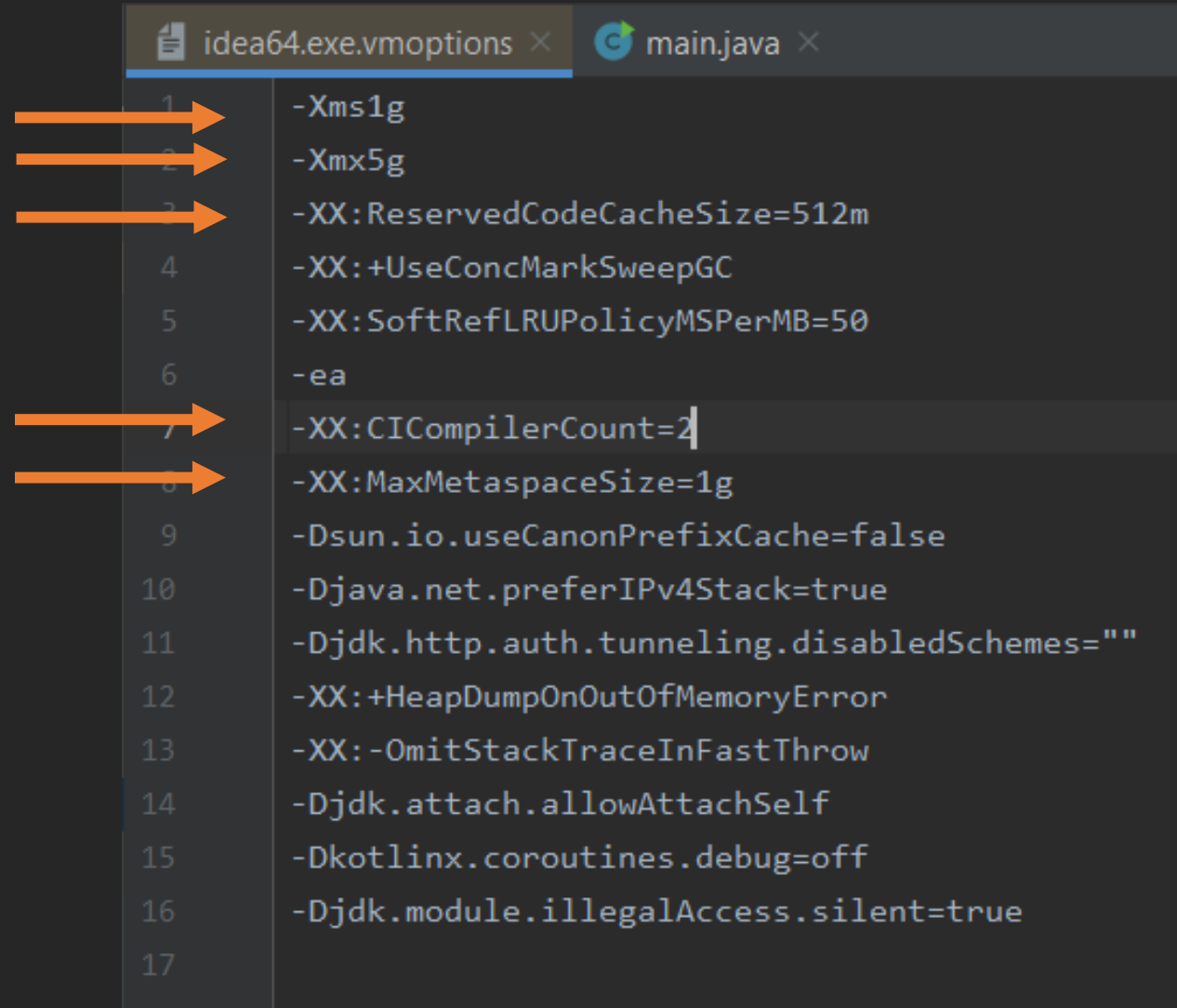
14

15

$$T(0) = 1$$

$$T(n) = T(n-1) + n$$

## 4. Detalhes



```
idea64.exe.vmoptions x main.java x
1 -Xms1g
2 -Xmx5g
3 -XX:ReservedCodeCacheSize=512m
4 -XX:+UseConcMarkSweepGC
5 -XX:SoftRefLRUPolicyMSPerMB=50
6 -ea
7 -XX:CICompilerCount=2
8 -XX:MaxMetaspaceSize=1g
9 -Dsun.io.useCanonPrefixCache=false
10 -Djava.net.preferIPv4Stack=true
11 -Djdk.http.auth.tunneling.disabledSchemes=""
12 -XX:+HeapDumpOnOutOfMemoryError
13 -XX:-OmitStackTraceInFastThrow
14 -Djdk.attach.allowAttachSelf
15 -Dkotlinx.coroutines.debug=off
16 -Djdk.module.illegalAccess.silent=true
17
```



Fim