

Análises de algoritmos de ordenação | Big-O

João Eudes Lima

Novembro, 2020

Introdução

Existem muitas formas de se ordenar um vetor, mas é comum aceitar qualquer implementação da própria linguagem ou biblioteca quando não se percebe gargalos ou problemas de performances dado as máquinas mais robustas atuais.

Entretanto, podemos nos deparar com o ambiente onde segundos fazem minutos ou até horas de diferença em um processo (e vale lembrar que tempo é dinheiro), e ao aplicar algum algoritmo que impulse o desempenho pode ser desesperador se dá conta com o que não foi implementado antes, mas para isso cabe ao desenvolvedor ter a visão ampliada sobre o problema e aplicar o que convém para a solução.

Iremos comparar alguns algoritmos por meio de Big-O para extrair o tempo significativo das execuções tendo em foco alguns métodos de ordenação, e assim obter algumas métricas e eficiência na execução de cada cenário.

Ambiente de benchmark, análises e processos

Para os testes optei pela utilização da linguagem de programação Rust por gerar um compilado com vários benefícios de otimizações por conta da LLVM e Clang.

Para os benchmarks foi realizado um laço de repetição que se repetia 250 vezes começando com o valor de 100 números para serem ordenados e que aumenta 100 e para proporcionar um melhor ambiente de comparação foi inserido um outro laço para pegar a média de 4 execuções do algoritmo desabilitado o "Turbo boost"

Para suavizar as anomalias do Gnuplot utilizei comandos como 'using 1:2 smooth bezier' e criei script shell para geração automática a partir dos dados em tempo.

Utilizei do Valgrind para relatório de uso de memória e liberação da mesma.

```
pub fn loop_vec_in_case(type_case: TypeCase, callback: fn(&mut Vec<i32>)) -> ResultSampleBench {
    let mut n: i32 = 100;
    let mut data: HashMap<i32, f32> = HashMap::new();

    let loop_media: f32 = 4.0;

    for _ in 0..250 {
        let mut vec: Vec<i32> = match type_case {
            TypeCase::BETTER => { (0..n).collect() }
            TypeCase::MEDIUM => { gen_shuffled_vec(lim: n) }
            TypeCase::WORSE => { (0..n).rev().collect() }
        };

        for _ in 1..=loop_media as i32 {
            let now: Instant = Instant::now();

            callback(&mut vec);

            let duration: f32 = now.elapsed().as_secs_f32();

            data.entry(key: n).or_insert_with(|| {
                Entry::new(n, {
                    .and_modify(|old: &mut f32| { *old += duration / loop_media });
                    .or_insert(default: duration / loop_media);
                })
            });

            println!("{}", n, " ", *data.get(&n).unwrap());

            n += 100;
        }

        let mut times: Vec<f32> = vec![];
        let mut items: Vec<i32> = vec![];

        for (k: i32, v: f32) in data.drain() {
            items.push(k);
            times.push(v);
        }

        return ResultSampleBench::new(total_items: items, total_time: times);
    }
}
```

Loop com callback resultando o vetor para determinado caso.

```
joaoeudes7@archlinux
OS: Arch Linux
Kernel: x86_64 Linux 5.9.6-arch1-1
Uptime: 1m
Packages: 1110
Shell: zsh 5.8
Resolution: 3840x1080
DE: KDE 5.75.0 / Plasma 5.20.2
WM: KWin
GTK Theme: Breeze [GTK2/3]
Icon Theme: Papirus-Dark
Disk: 76G / 125G (64%)
CPU: Intel Core i7-7500U @ 4x 2.7GHz [55.0°C]
GPU: Intel Corporation HD Graphics 620 (rev 02)
RAM: 1264MiB / 15908MiB
```

Especificações da máquina utilizada.

Concepções

1 O que é melhor caso?

Denominado quando os dados para ordenação se encontra em um estado pré-ordenado

2 O que é o pior caso?

Denominado quando os dados para ordenação se encontra em um estado ordenado inversamente

3 O que é o caso médio?

Também conhecido por tratar outros casos, como quando os dados para ordenação se encontra em um estado com posições aleatórias

Algoritmos

4 Merge sort

4.1 Descrição

O Mergesort é um algoritmo que funciona de forma recursivo. A cada chamada divide o vetor recebido e então é tratado de forma separada, onde os resultados são combinados no final.

4.2 Lógica de execução

1. Divide o vetor ao meio
2. Pega o item do meio
3. Divide novamente o subconjunto de forma recursiva e ordenadas
4. Quando a classificação das duas coleções estiver concluída, os resultados serão mesclados
5. Agora o Merge Sort escolhe o item que é menor e insere esse item na nova coleção.
6. Em seguida, seleciona os próximos elementos e classifica o elemento menor de ambas as coleções

4.3 Algoritmo

```
rust - merge_ed.rs

1 pub fn merge_sort(arr: &mut Vec<i32>, start: usize, end: usize) {
2     if start < end {
3         let mid = (start + end) / 2;
4
5         merge_sort(arr, start, mid);
6         merge_sort(arr, mid + 1, end);
7         merge(arr, start, mid, end);
8     }
9 }
10
11 fn merge(arr: &mut Vec<i32>, start: usize, mid: usize, end: usize) {
12     let mut i = start;
13     let mut j = mid + 1;
14     let diff = end - start + 1;
15
16     let mut b: Vec<i32> = vec![0; diff];
17
18     for k in 0..diff {
19         if j > end || arr[i] < arr[j] && i <= mid {
20             b[k] = arr[i];
21             i += 1;
22         } else {
23             b[k] = arr[j];
24             j += 1;
25         }
26     }
27
28     for k in 0..diff {
29         arr[start + k] = b[k];
30     }
31 }
32
```

4.4 Complexidade

O Algoritmo merge tem tempo parcialmente equivalente em todos os casos, considerando a porcentagem de tempo e a proximidade entre eles e complexidade analítica.

$$T^m(n) = C_1 + C_2 + C_3(n+1) + C_4n + (C_5 + C_6 + C_7 + C_8)\left(\frac{n}{2}\right) + C_9 + C_{10} + C_{11}(n+1) + (C_{12} + C_{13})n + C_{14}$$

$$T^m(n) = (C_3 + C_4 + C_9 + C_{11} + C_{12} + C_{13})n + (C_5 + C_6 + C_7 + C_8)\left(\frac{n}{2}\right) + C_1 + C_2 + C_3 + C_{11} + C_{14}$$

Considerando:

$$a = C_1 + 2C_2 + C_3 + C_4 + C_5$$

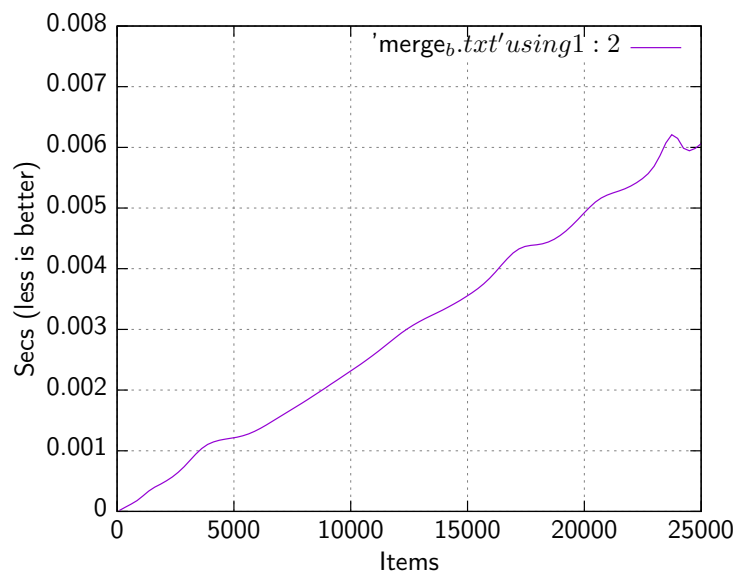
Temos para:

$$T(1) = C_1$$

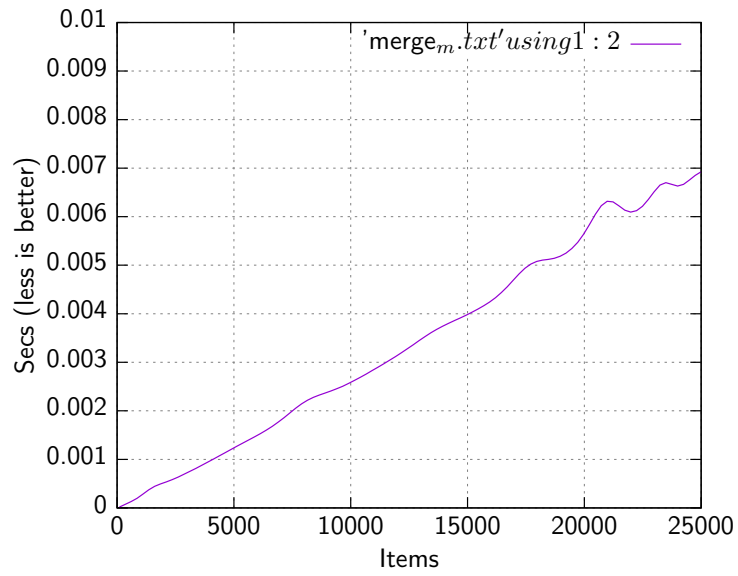
$$T(n) = a + 2T\left(\frac{n}{2}\right) + T^m(n)$$

Com isso notamos que a complexidade é $\theta(n \log_2 n)$

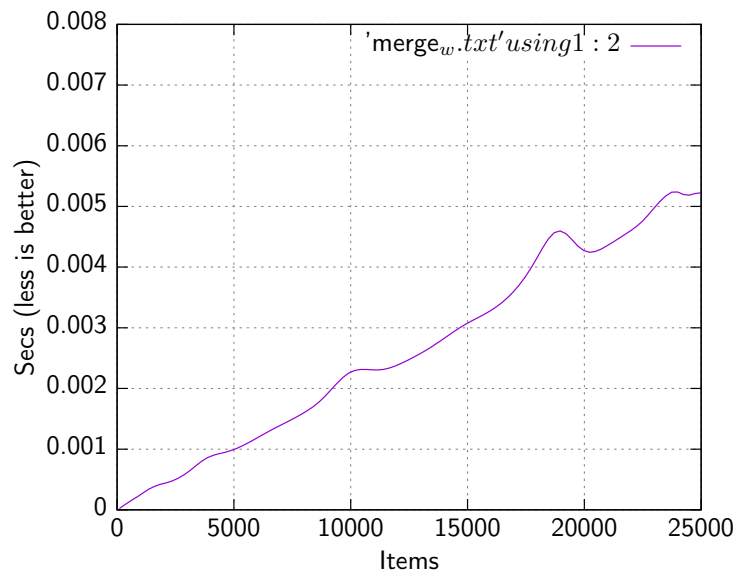
4.4.1 Melhor caso



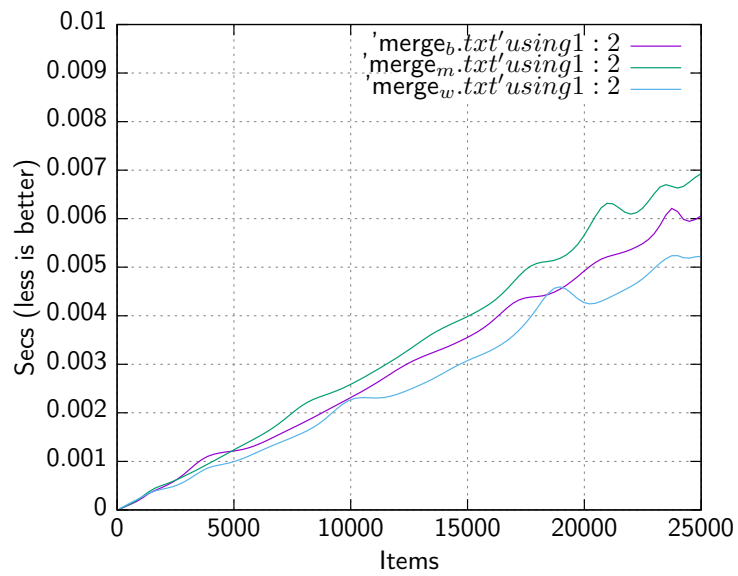
4.4.2 Caso Médio



4.4.3 Pior caso



4.4.4 Geral



De acordo com o relatório do Valgrid durante o teste geral:
Total heap usage: 12,549,292 allocs, 12,549,292 frees,
713,067,461 bytes allocated

Podemos notar que apesar de seu bom tempo, ele tem um uso elevado de RAM já que necessita de um vetor auxiliar para fazer o swap entre as posições.

5 Insertion Sort

5.1 Descrição

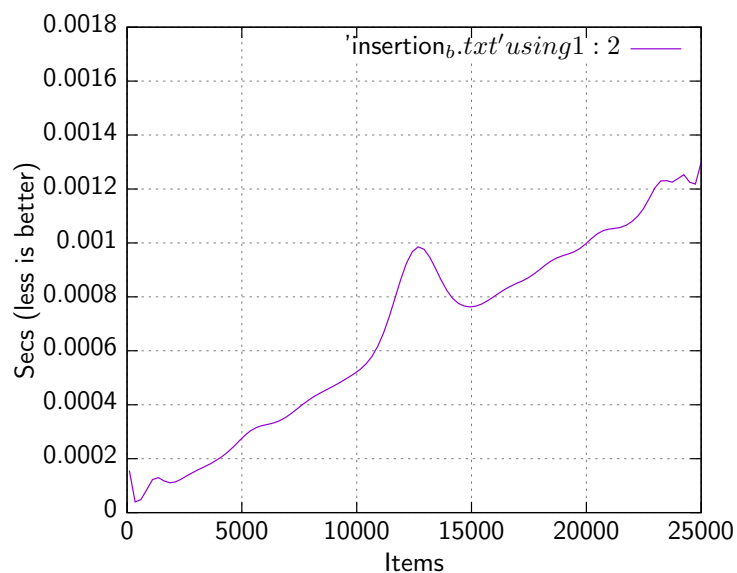
5.2 Algoritmo

```
1 pub fn insertion_sort(arr: &mut [i32]) {
2     for j in 1..(arr.len() as i32) {
3         let k = arr[j as usize];
4         let mut i = j - 1;
5
6         while i > -1 && arr[i as usize] > k {
7             arr[(i+1) as usize] = arr[i as usize];
8             i -= 1;
9         }
10
11         arr[(i+1) as usize] = k;
12
13     }
14 }
15
```

5.3 Complexidade

O Algoritmo Insertion tem tempo parcialmente equivalentes no pior e no caso médio, considerando a complexidade analítica.

5.3.1 Melhor caso



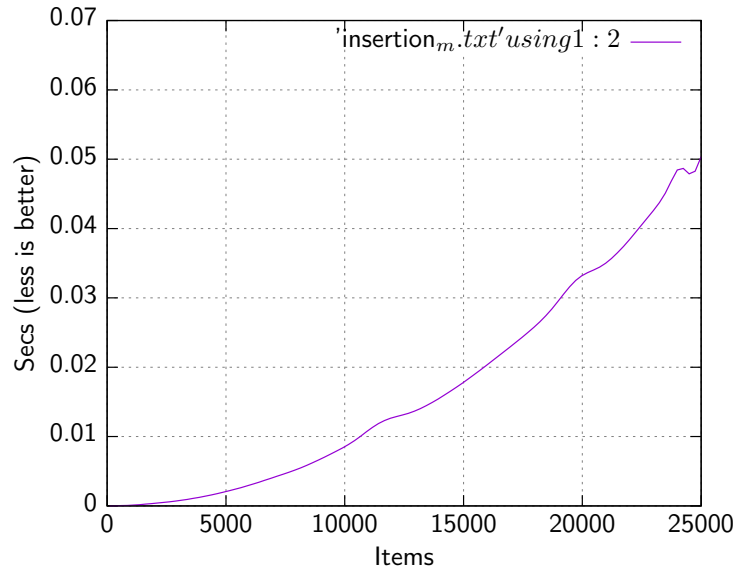
Apesar de algumas anomalias, podemos observar que a complexidade é linear

$$T_b(n) = C_2(n-1) + C_3(n-2) + C_4(n-2) + C_6(n-2) + C_{11}$$

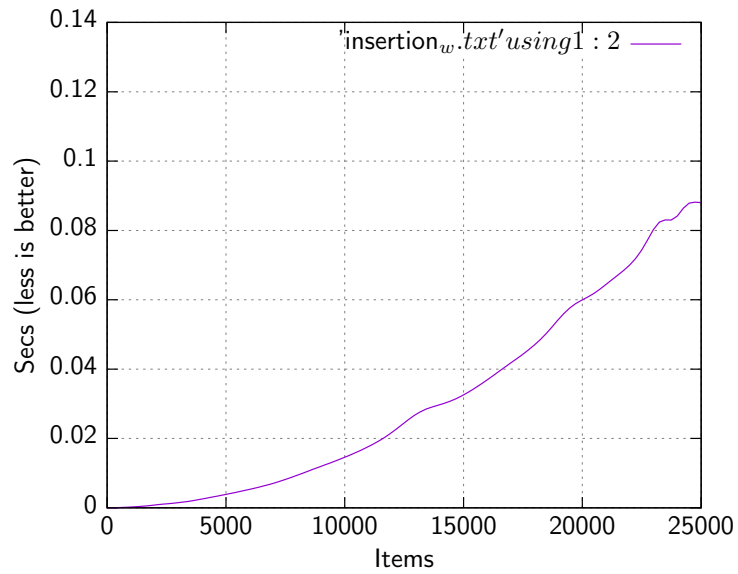
$$T_b(n) = C_2(n-1) + (n-2)(C_2 + C_3 + C_4 + C_6) + C_{11}$$

Com isso notamos que a complexidade é $O(n)$ (linear)

5.3.2 Caso Médio



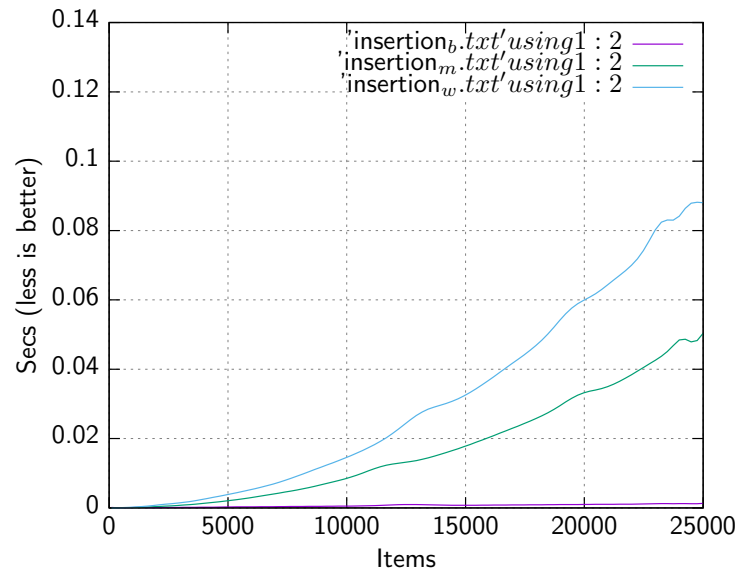
5.3.3 Pior caso



$$T_w(n) = C_1n + (C_2 + C_3 + C_7)(n - 1) + C_4\left(\frac{n}{2}(n + 1) - 1\right) + (C_5 + C_6)\left(\frac{n}{2}(n - 1)\right)$$

Com isso notamos que a complexidade é $O(n^2)$ (quadrática)

5.3.4 Geral



De acordo com o relatório do Valgrid durante o teste geral:
Total heap usage: 292 allocs, 292 frees
12,566,917 bytes allocated

6 Quick sort

6.1 Descrição

Estruturado como uma Árvore Binária, sua estratégia de ordenação é baseada no padrão de projeto (Design Pattern) divisão e conquista, utilizando como parte da estratégia a recursividade.

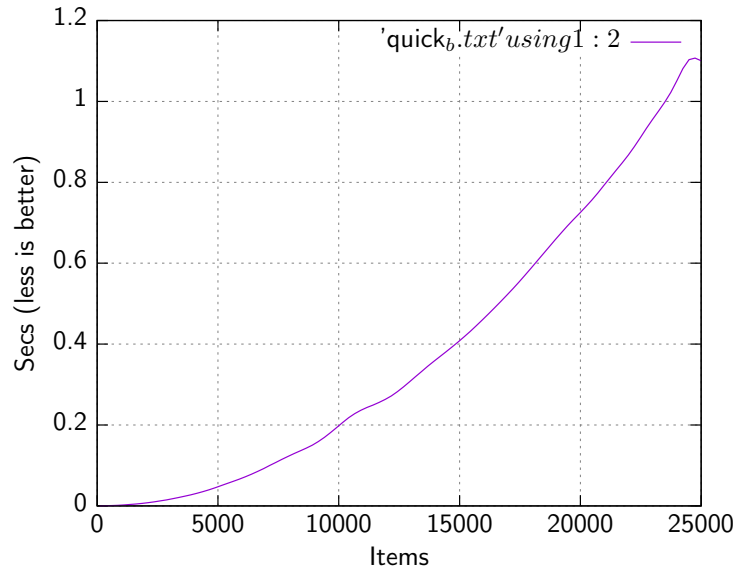
6.2 Algoritmo

```
1 fn quick_sort(arr: &mut [i32], start: i32, end: i32) {
2     if start < end {
3         let p = partition(arr, start as usize, end as usize);
4
5         quick_sort(arr, start, p - 1);
6         quick_sort(arr, p + 1, end);
7     }
8 }
9
10 fn partition(arr: &mut [i32], start: usize, end: usize) -> i32 {
11     let mut i = start;
12
13     for j in start..end {
14         if arr[j] < arr[end] {
15             arr.swap(i, j);
16
17             i += 1;
18         }
19     }
20
21     arr.swap(i, end);
22
23     return i as i32;
24 }
25
```

6.3 Complexidade

O Algoritmo Quick tem tempo parcialmente equivalentes no melhor e no caso médio, considerando a complexidade analíticaIIIIII.

6.3.1 Melhor caso



$$T_b(n) = C_1 + C_2 + C_3 + C_4 + 2T_b\left(\frac{n-1}{2}\right) + T^p(n)$$

Considerando

$$a = C_1 + C_2 + C_3 + C_4$$

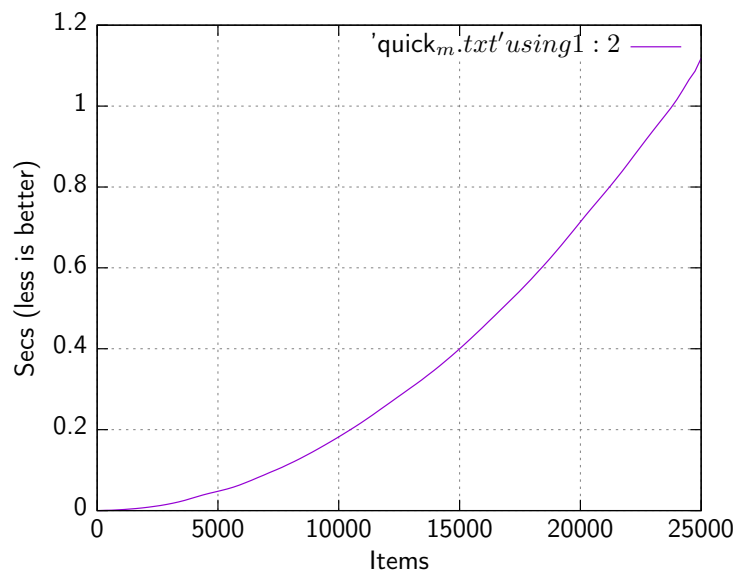
$$T_b(0) = C_1$$

$$T_b(1) = C_1$$

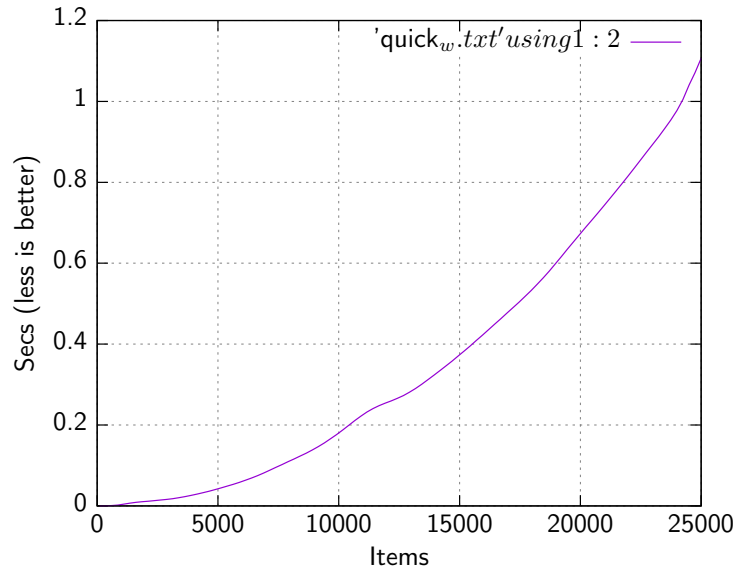
$$T_b(n) = a + 2T_b\left(\frac{n-1}{2}\right) + T^p(n)$$

E então concluímos que a complexidade desse caso é $O(n \log_2 n)$

6.3.2 Caso Médio



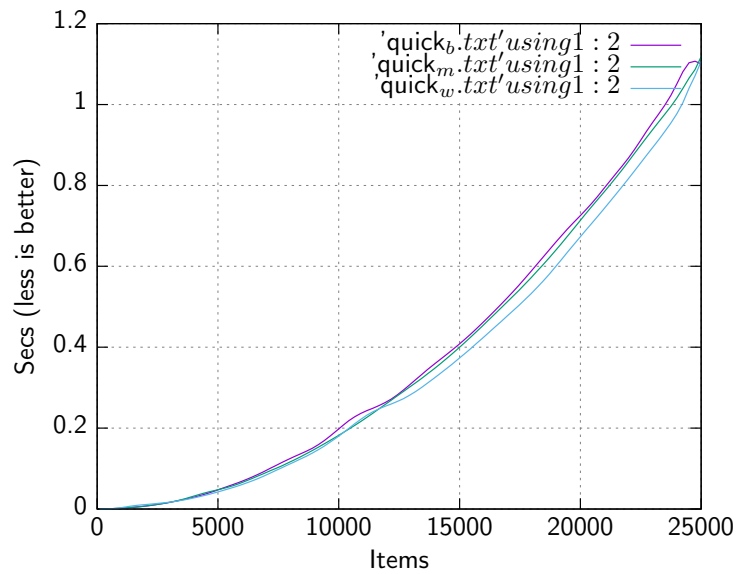
6.3.3 Pior caso



$$T_w(n) = C_1 + 2C_2 + C_3 + C_4 + T^p(n)$$

E então concluímos que a complexidade desse caso é $O(n^2)$

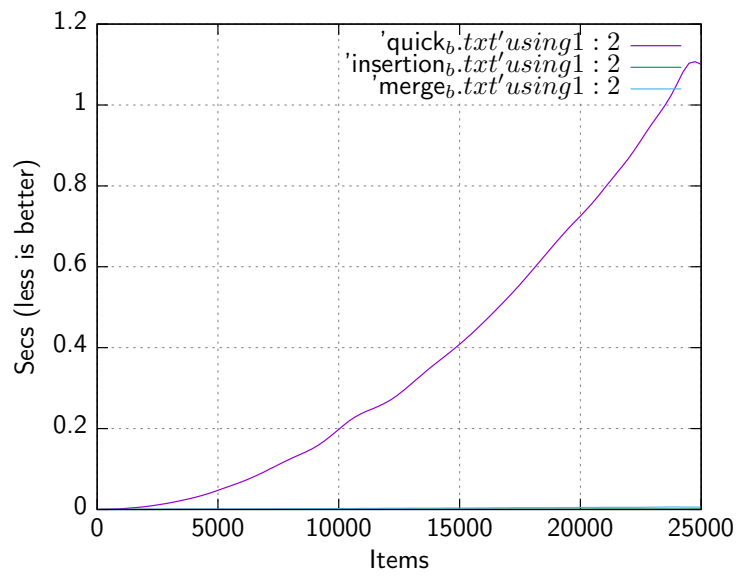
6.3.4 Geral



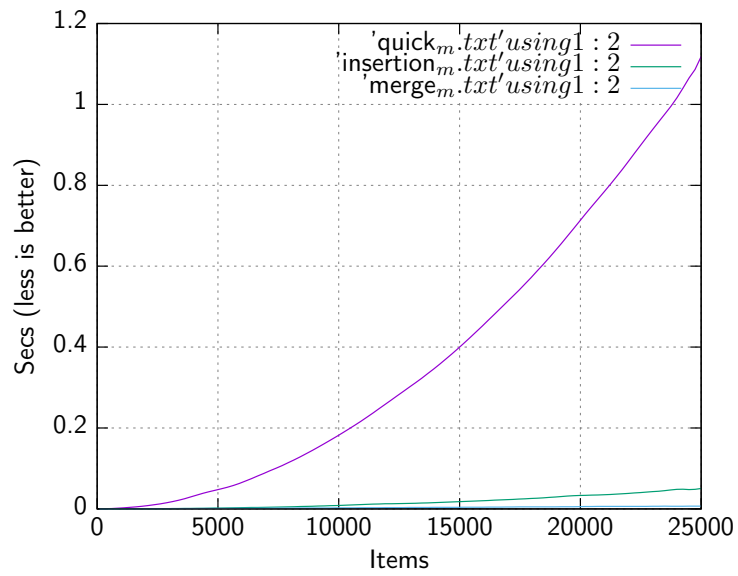
De acordo com o relatório do Valgrid durante o teste geral:
 Total heap usage: 292 allocs, 292 frees
 12,566,917 bytes allocated

7 Comparações

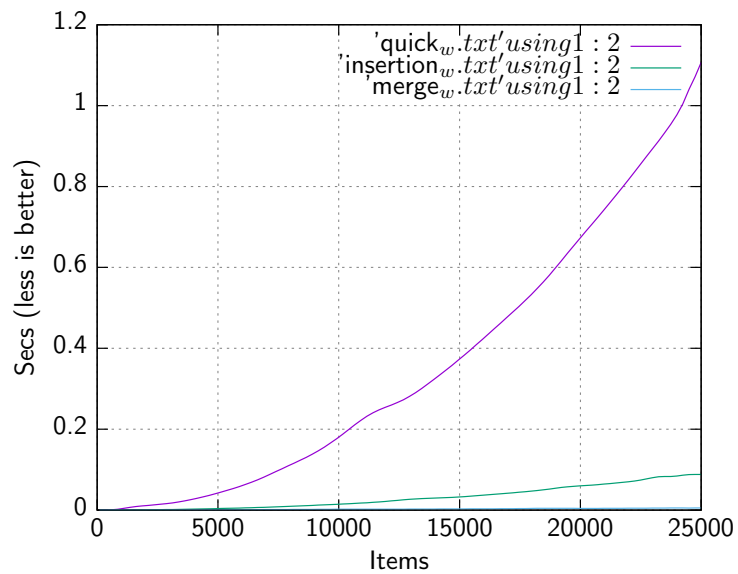
7.1 Qual o melhor no melhor caso?



7.2 Qual o melhor no caso médio?



7.3 Qual o melhor no pior caso?



Foi notado que merge sort se mostrou o mais eficaz em todos os casos, o que não é comum, já que me tese sua complexidade tende a ser maior quando comparado com alguns dos algoritmos, entretanto, isso pode se ter devido otimizações de compilação.

Os algoritmos, assim como outros dados dese relatório se encontra disponível para verificação e validação se necessário, onde pode ser encontrado clicando aqui: [Repositório do relatório](#)

Informações para rodar e mais pode ser encontrado no Readme do projeto.