Universidade do Vale do Itajaí

Campus Kobrasol

Análise de Algoritmos

Avaliação M3

Grupo: Maurício Macário de Farias Junior

Professor: Antonio Carlos Sobieranski

São José, 2018, 09 de Março

Resumo

Neste trabalho será apresentado a comparação entre três algoritmos de ordenação (Bubble Sort, Insertion Sort e Merge Sort), levando em consideração o tempo de execução e o uso de memória, nota-se que foram executados os processos de ordenação 100 vezes para que a comparação fosse mais visível e facil de inspecionar, também serão comparados os métodos fibonacci sendo um sem recursividade e um com, esses processos foram executados 1000000 de vezes para que pudesse ser observado a efetividade de cada um.

Desenvolvimento

Aqui serão mostrados os resultados do uso das ferramentas para avaliar os algoritmos de ordenação e também os métodos de implementação fibonacci.

Para a visualização do tempo de execução foi utilizado a ferramenta Gprof, para a visualização do uso de memória e o numero de instruções executadas foi usada a ferramenta massif do Valgrind, o eixo X da imagem irá representar a quantidade de instruções executadas e o eixo Y irá representar a quantidade de memória utilizada no programa, e para a visualização do gráfico mostrando a porcentagem gasta em cada parte foi usado o CallGrind novamente do Valgrind e exebido com o Kcachegrind.

Todos os testes dos algoritmos de ordenação foram feitos e serão exibidos na ordem de tamanho de vetores 100, 1000 e 10000, foi escolhido que cada um desses algoritmos fosse executado 100 vezes para melhor visualização de dados.

Para os métodos do fibonacci foram utilizados os parâmetros 5, 25 e 100, e também, cada um dos métodos foi executado 1000000 de vezes para melhor visualização.

Bubble Sort

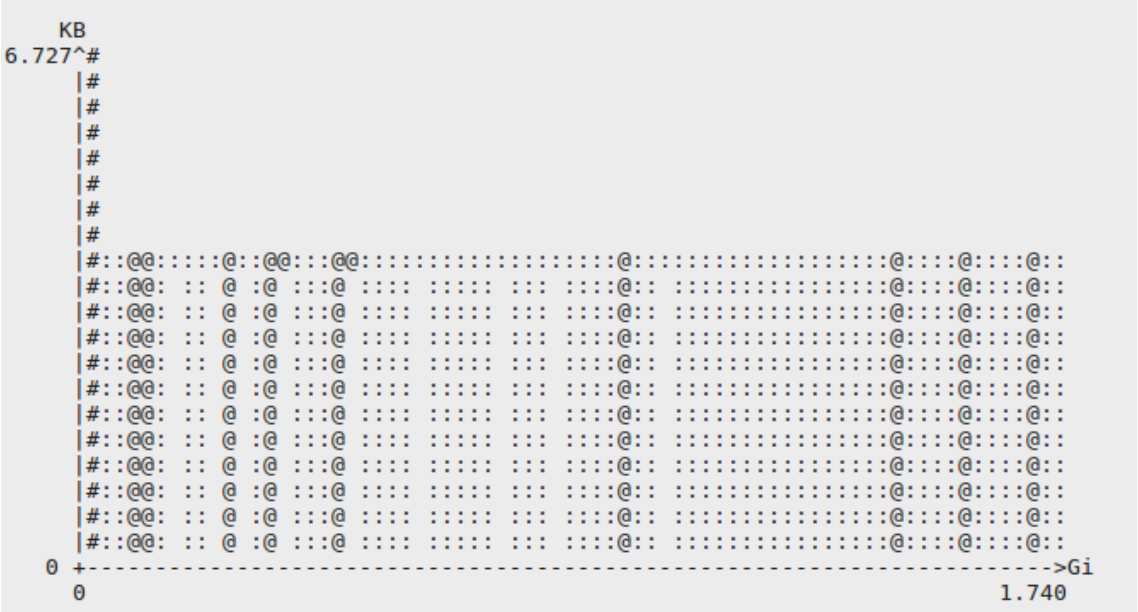
Abaixo estarão apresentados os tempos de para os três tamanhos de vetores ordenados, pode-se perceber que o tempo de execução aumenta consideravelmente.

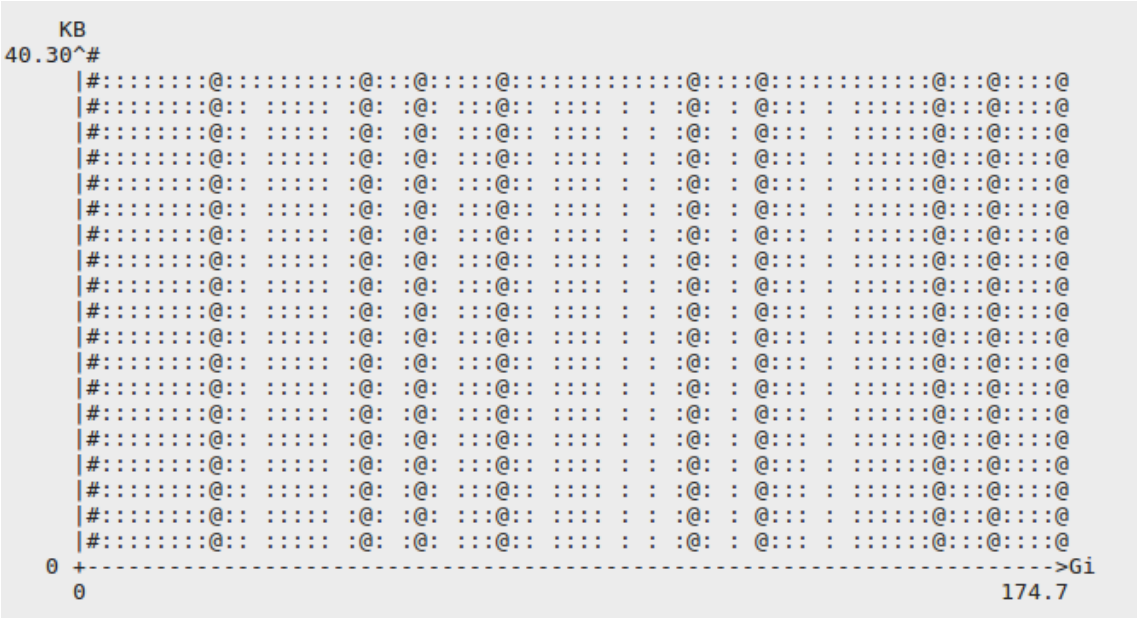
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00   246833     0.00     0.00  swap(int*, int*)
  0.00      0.00     0.00      100     0.00     0.00  bubbleSort(int*, int)
```

```
 time   seconds   seconds    calls  ms/call  ms/call  name
 79.24      0.17     0.17      100     1.66     2.02  bubbleSort(int*, int)
 16.81      0.20     0.04 24943364     0.00     0.00  swap(int*, int*)
  2.40      0.21     0.01                             frame_dummy
```

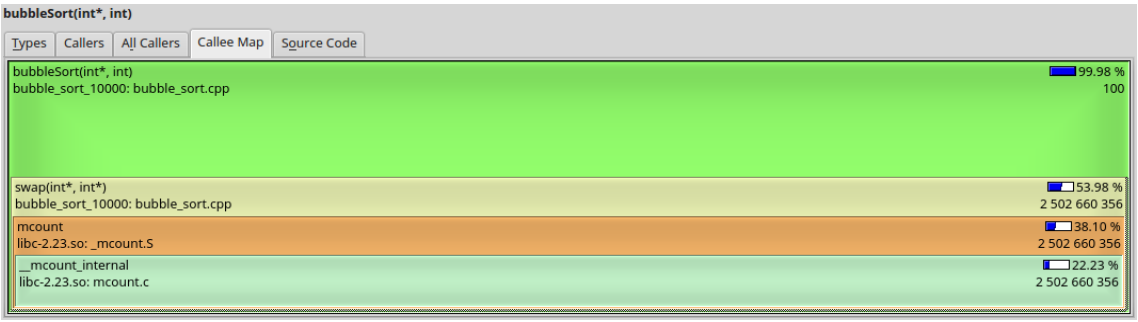
```
time   seconds   seconds    calls  ms/call  ms/call  name
79.55     24.48    24.48      100   244.77   302.26  bubbleSort(int*, int)
18.68     30.23     5.75 2502660356     0.00     0.00  swap(int*, int*)
 1.44     30.67     0.44                             frame_dummy
```

Abaixo estão apresentados o uso de memória para cada um dos três tamanhos de vetores, pode-se perceber que a quantidade de memória consumida é baixa mas as instruções são consideravelmente altas.





Abaixo está apresentado o tempo gasto em cada parte do algoritmo nota-se que a função swap é chamada muitas vezes e consome grande parte do tempo de execução.



Insertion Sort

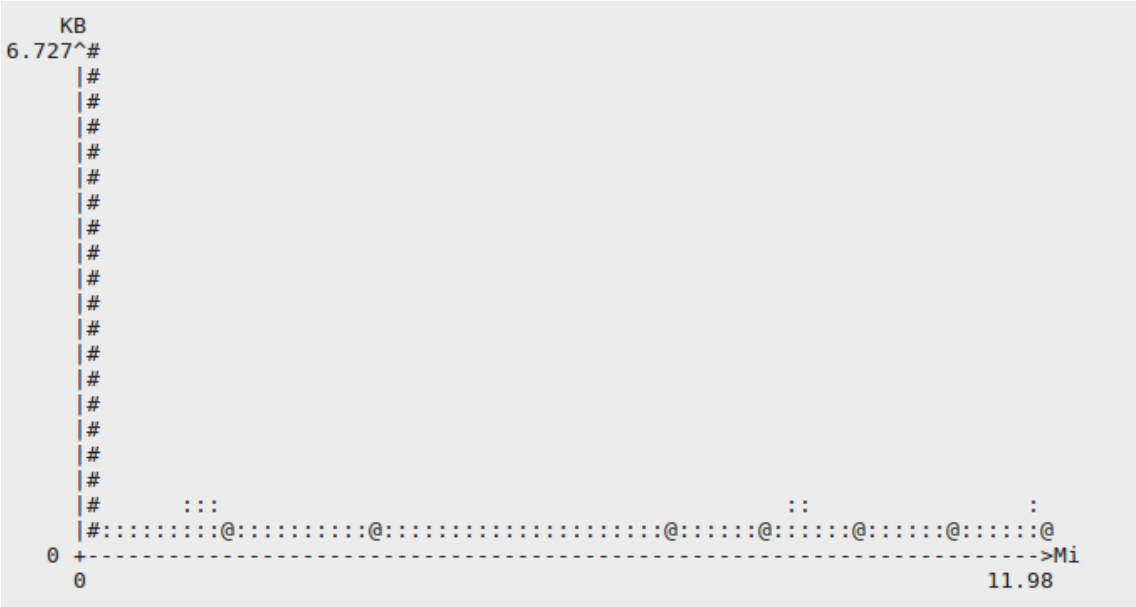
Abaixo estarão apresentados os tempos de para os três tamanhos de vetores ordenados, nota-se que o aumento no tempo de execução se mostra menor que o aumento no método Bubble Sort.

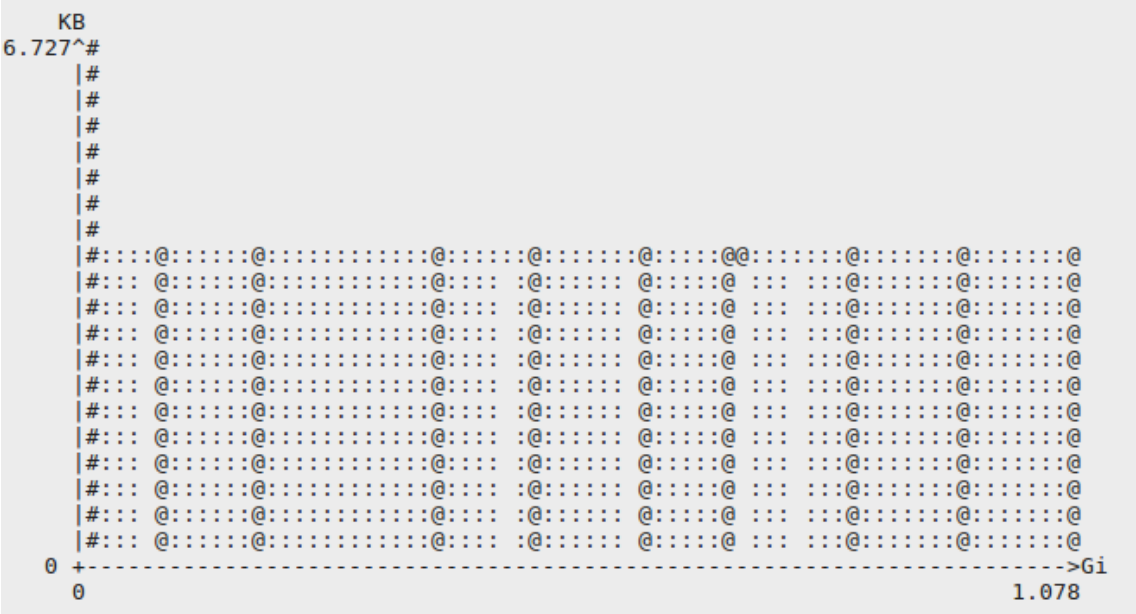
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      100     0.00     0.00  insertion_sort(int*, int)
```

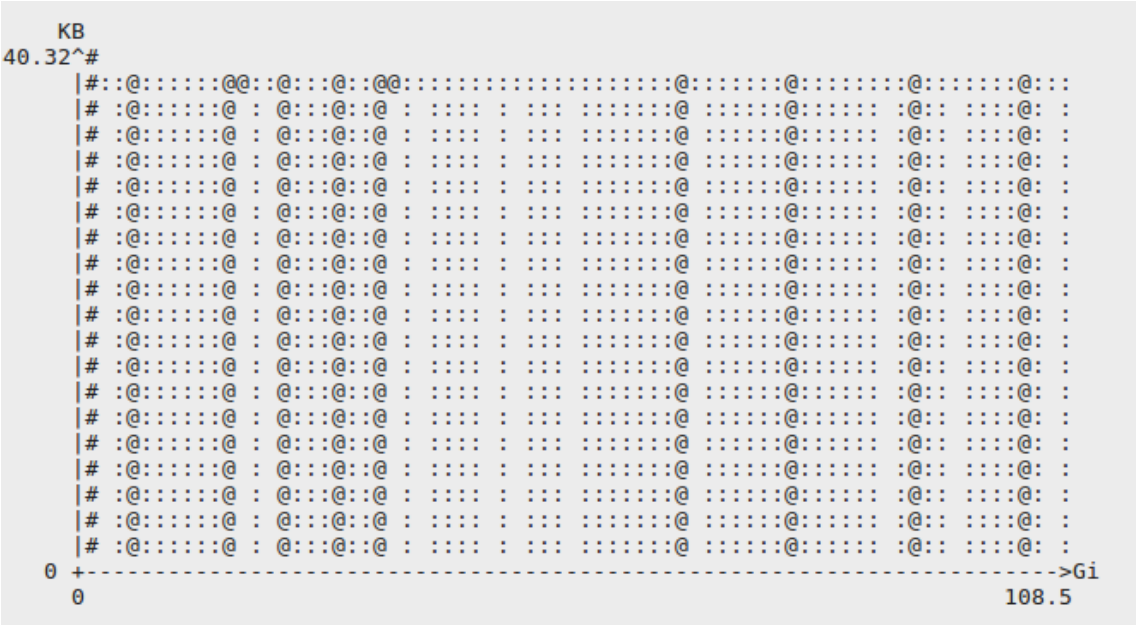
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.84      0.16     0.16      100     1.61     1.61  insertion_sort(int*, int)
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.80     11.73    11.73      100   117.33   117.33  insertion_sort(int*, int)
```

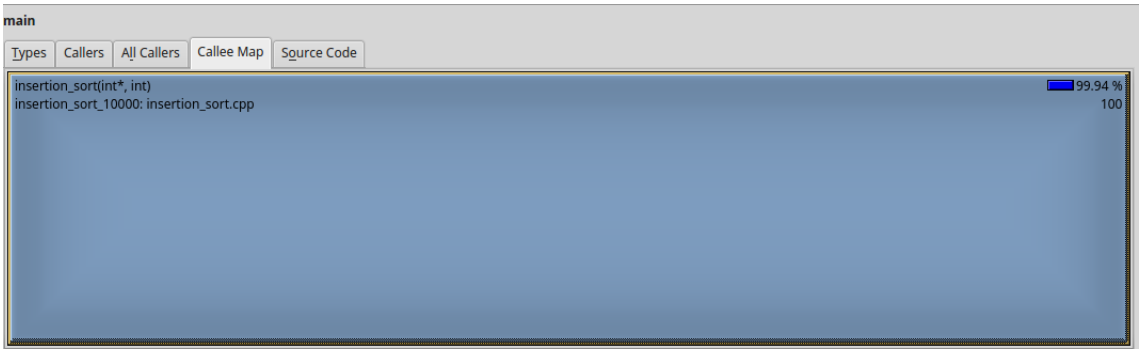
Abaixo estão apresentados o uso de memória para cada um dos três tamanhos de vetores, nota-se que assim como o tempo de execução, o número de instruções também diminuiu consideravelmente.







Abaixo está apresentado o tempo gasto em cada parte do algoritmo.



Merge Sort

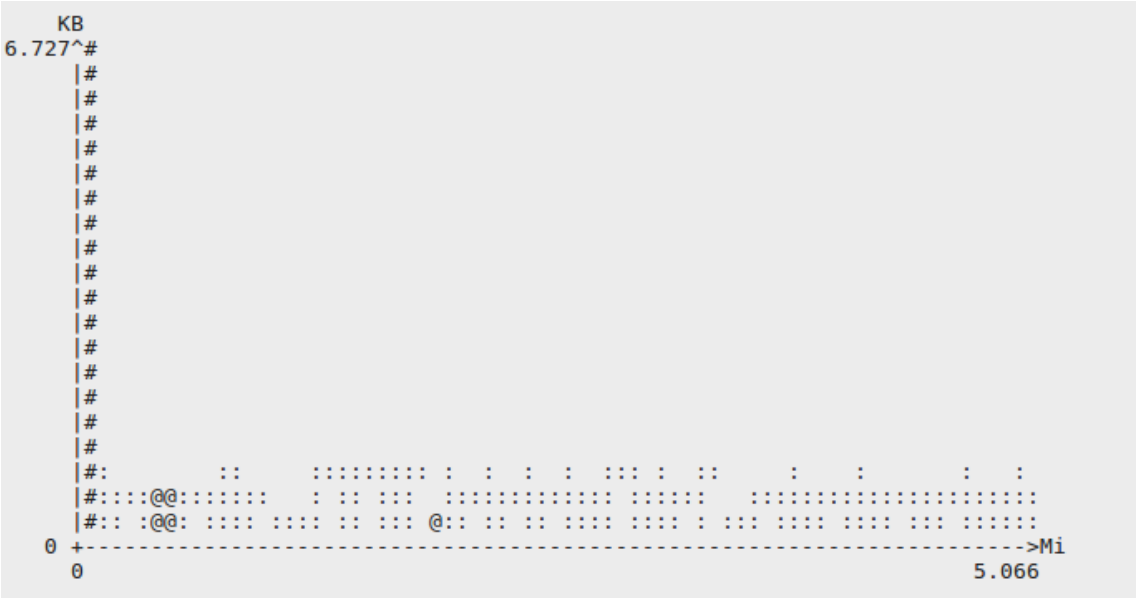
Abaixo estarão apresentados os tempos de para os três tamanhos de vetores ordenados, nota-se que o tempo de execução se mostra muito mais curto que os outros métodos.

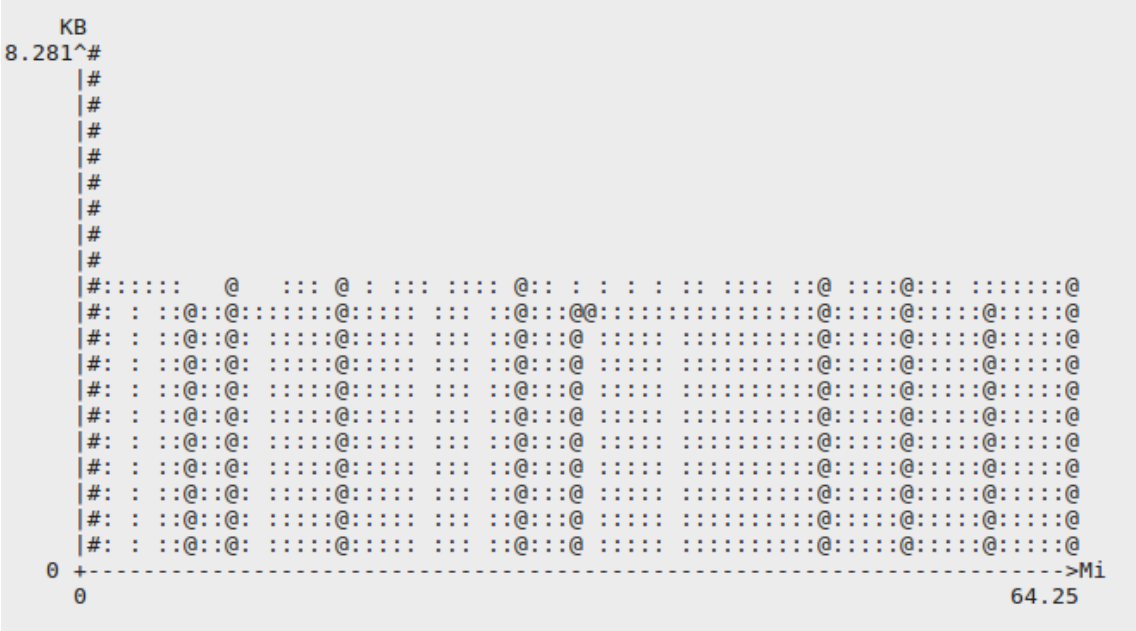
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     9900     0.00     0.00  merge(int*, int, int, int)
  0.00      0.00     0.00      100     0.00     0.00  mergeSort(int*, int, int)
```

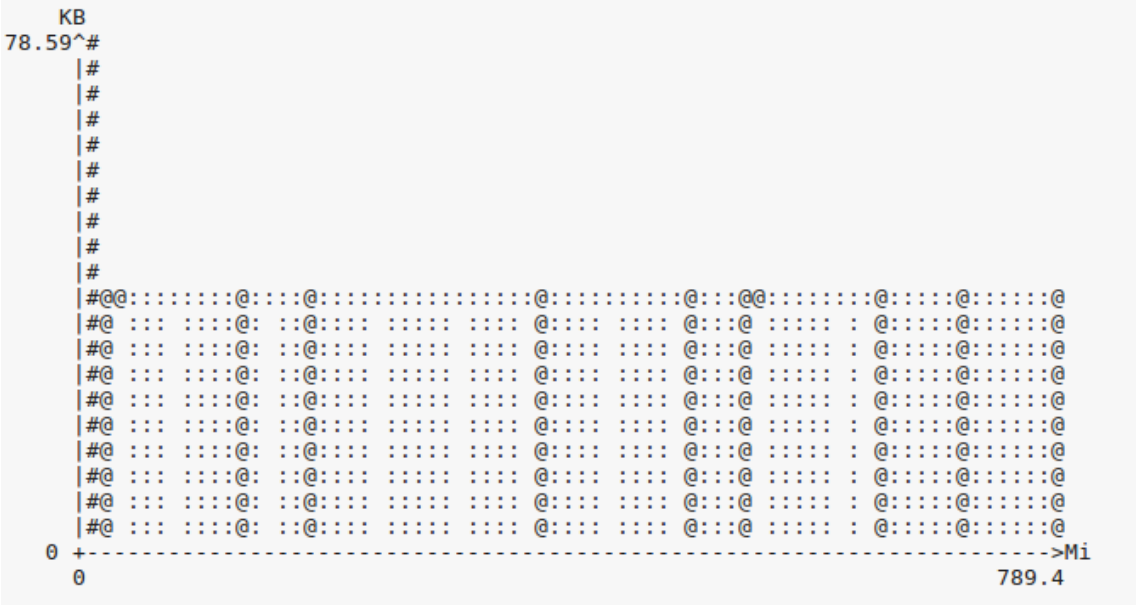
```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.52      0.02     0.02    99900     0.20     0.20  merge(int*, int, int, int)
  0.00      0.02     0.00      100     0.00   201.04  mergeSort(int*, int, int)
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 79.96      0.18     0.18   999900     0.00     0.00  merge(int*, int, int, int)
  9.14      0.20     0.02      100     0.20     1.96  mergeSort(int*, int, int)
  9.14      0.22     0.02                             main
```

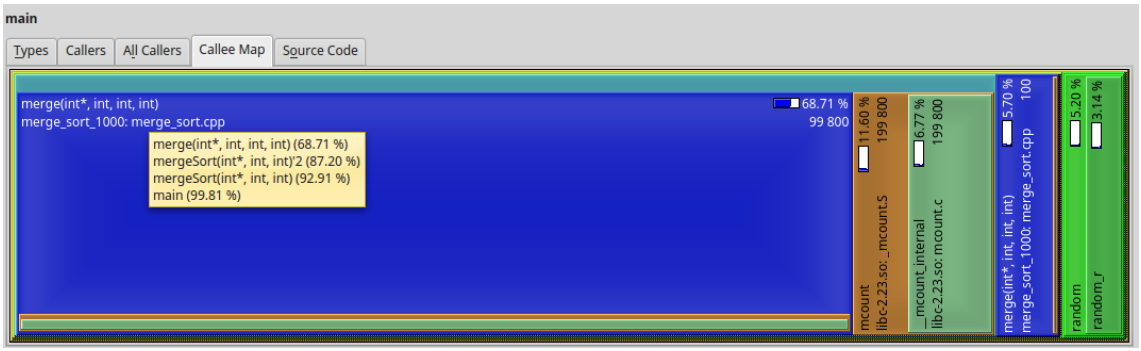
Abaixo estão apresentados o uso de memória para cada um dos três tamanhos de vetores, pode-se notar que houve uma diminuição consideravel de instruções em relação aos outros métodos, mas um aumenta no uso da memória.







Abaixo está apresentado o tempo gasto em cada parte do algoritmo, nota-se que parte do tempo está nas funções de geração randômica do vetor, essas funções só se tornaram relevantes o suficiente para aparecerem por conta do baixissimo tempo de execução do algoritmo.



Métodos Fibonacci Sem Recursividade

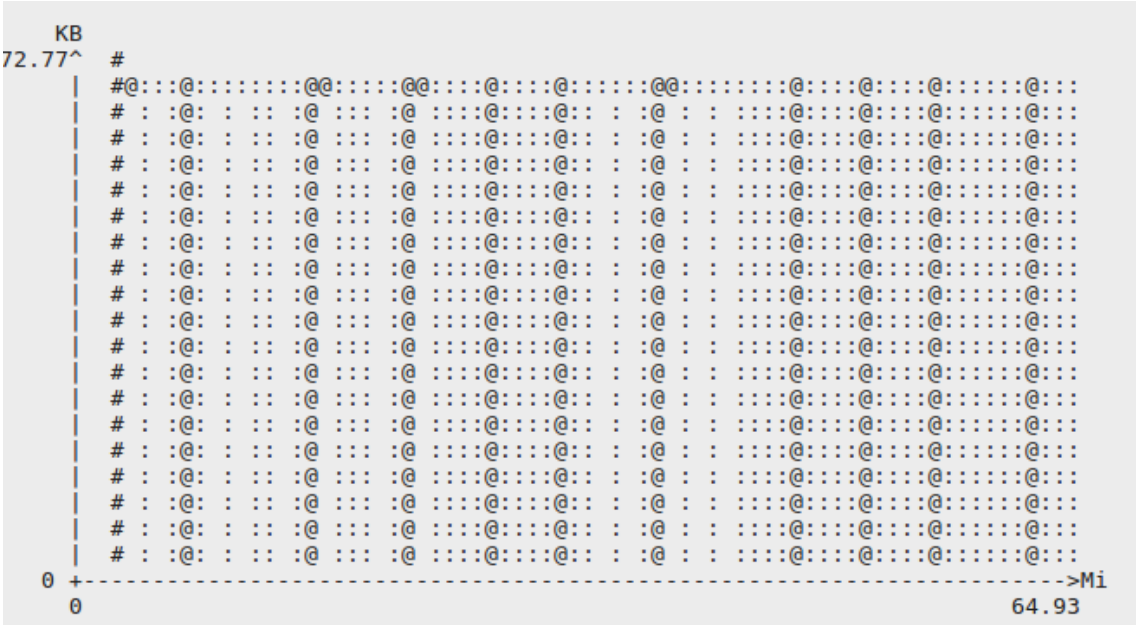
Abaixo estarão apresentados os tempos de execução para os três parâmetros diferentes do método Fibonacci, nota-se que mesmo pelo número gigantesco de chamadas, o tempo de execução é bem curto.

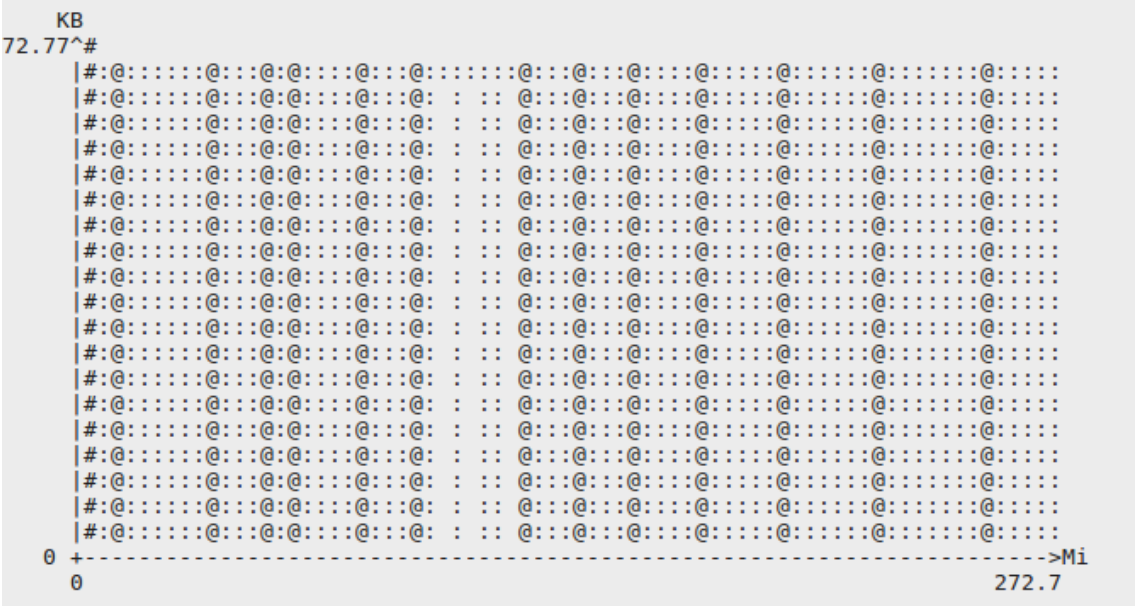
time	seconds	seconds	calls	ns/call	ns/call	name
101.05	0.01	0.01	1000000	10.11	10.11	fibonacci(int)
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonaccii
0.00	0.01	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

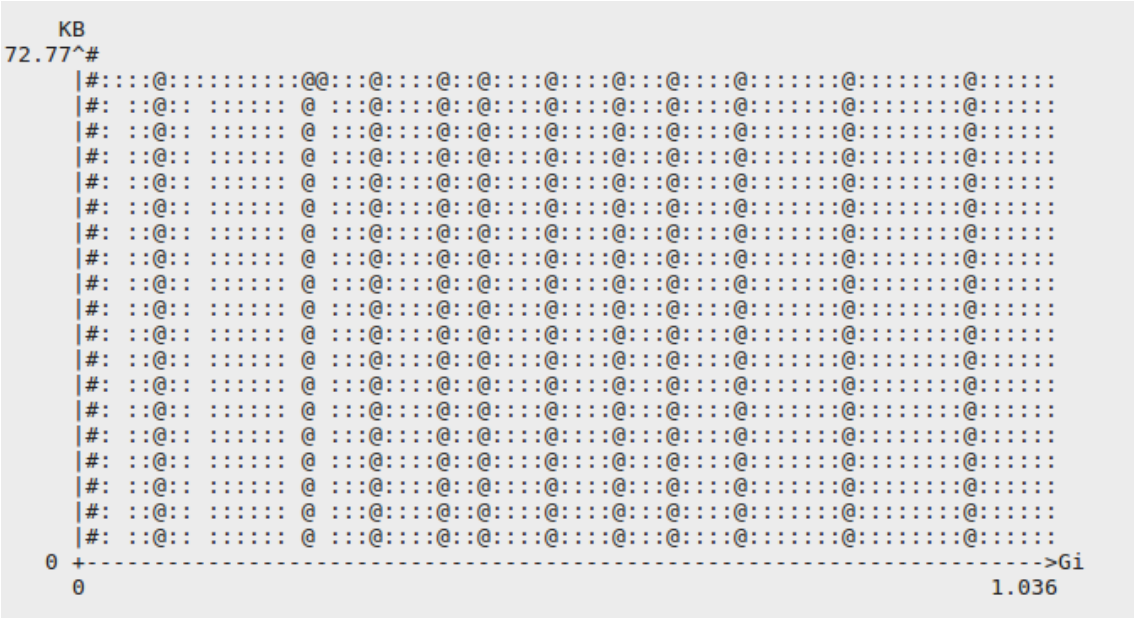
time	seconds	seconds	calls	ns/call	ns/call	name
89.82	0.08	0.08	1000000	80.84	80.84	fibonacci(int)
11.23	0.09	0.01				main
0.00	0.09	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonaccii
0.00	0.09	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

time	seconds	seconds	calls	ns/call	ns/call	name
101.05	0.29	0.29	1000000	293.05	293.05	fibonacci(int)
0.00	0.29	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonaccii
0.00	0.29	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

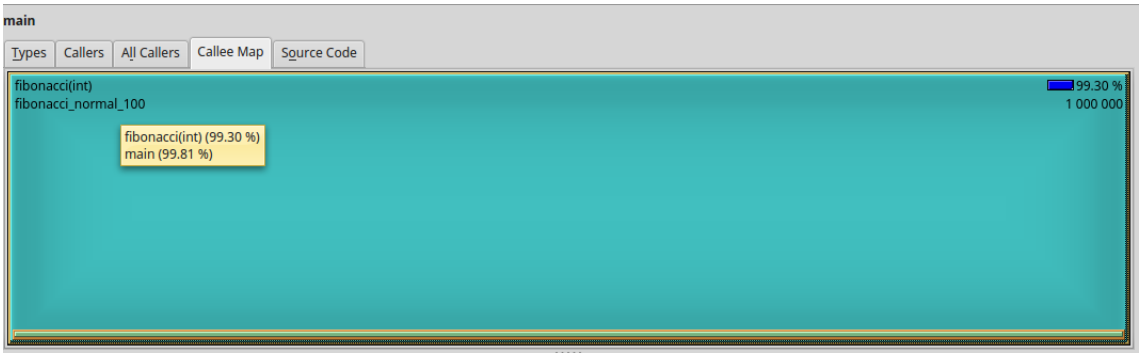
Abaixo estão apresentados o uso de memória para cada um dos três parâmetros do método Fibonacci, nota-se que o uso de memória não tende a mudar mesmo aumentando os parâmetros, diferente do número de instruções.







Abaixo está apresentado o tempo gasto em cada parte do algoritmo.



Métodos Fibonacci Com Recursividade

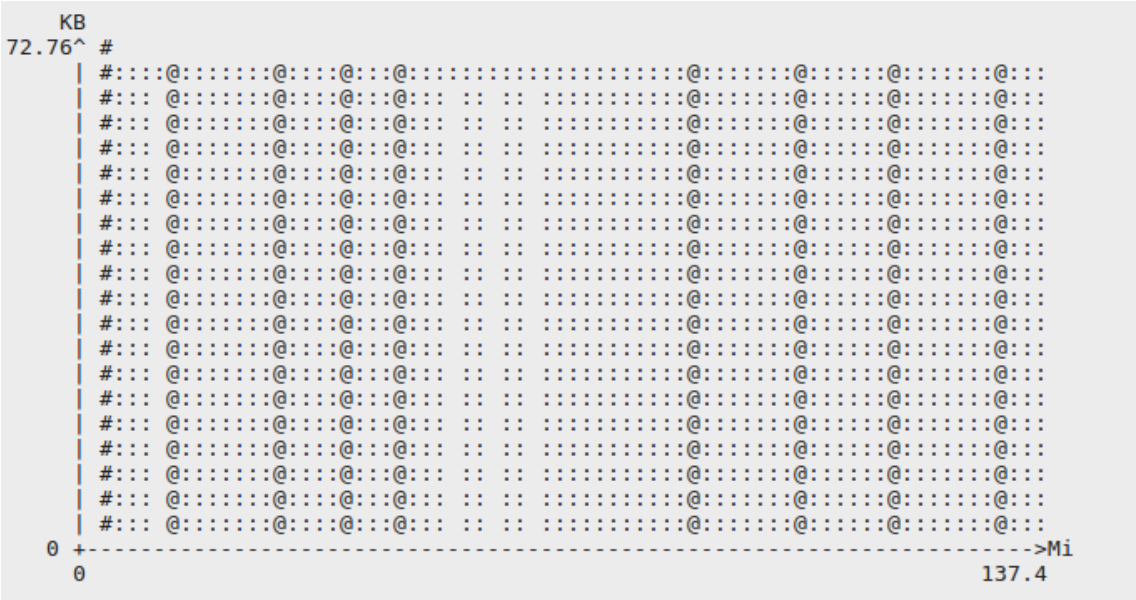
Abaixo estarão apresentados os tempos de execução para os três parâmetros diferentes do método Fibonacci, nota-se que se comparado ao método Sem Recursividade, é muito mais lento, chegando, em níveis mais altos a quase triplicar o tempo de execução.

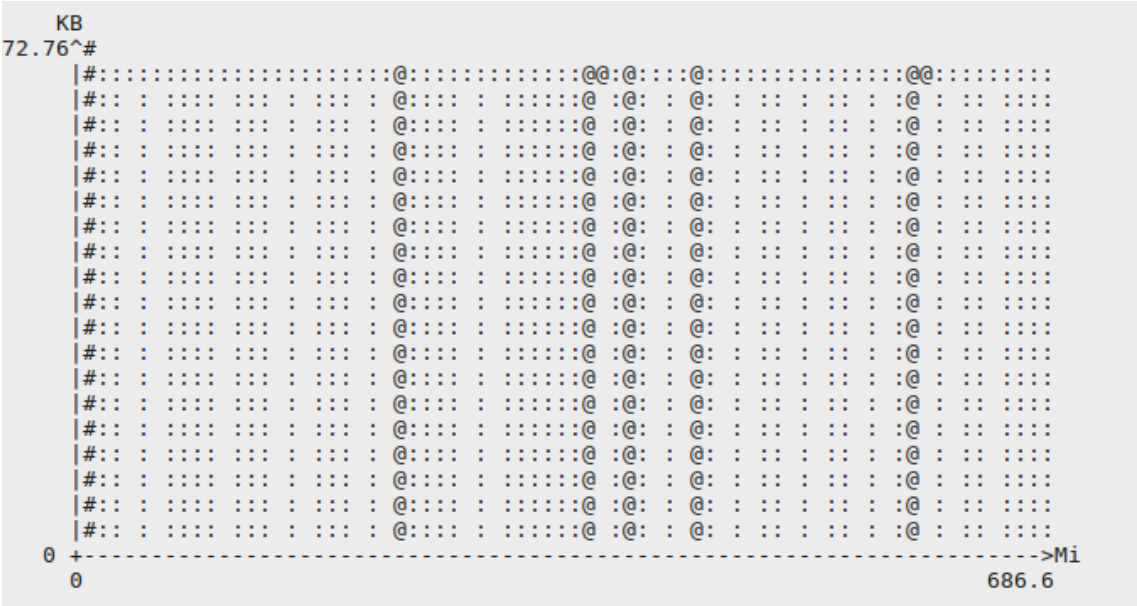
time	seconds	seconds	calls	ns/call	ns/call	name
101.01	0.03	0.03	1000000	30.30	30.30	fibonacci(int, int, int, int)
0.00	0.03	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonacciiiii
0.00	0.03	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

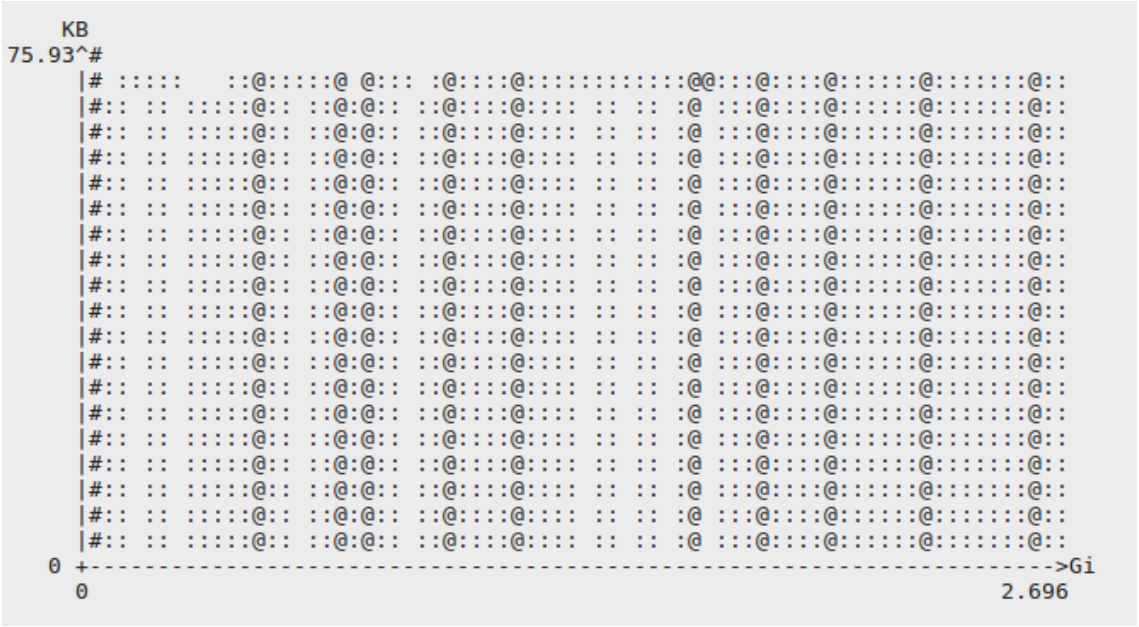
time	seconds	seconds	calls	ns/call	ns/call	name
87.24	0.10	0.10	1000000	95.96	95.96	fibonacci(int, int, int, int)
13.77	0.11	0.02				main
0.00	0.11	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonacciiiii
0.00	0.11	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

time	seconds	seconds	calls	ns/call	ns/call	name
84.72	0.53	0.53	1000000	525.25	525.25	fibonacci(int, int, int, int)
13.03	0.61	0.08				main
3.26	0.63	0.02				frame_dummy
0.00	0.63	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z9fibonacciiiii
0.00	0.63	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

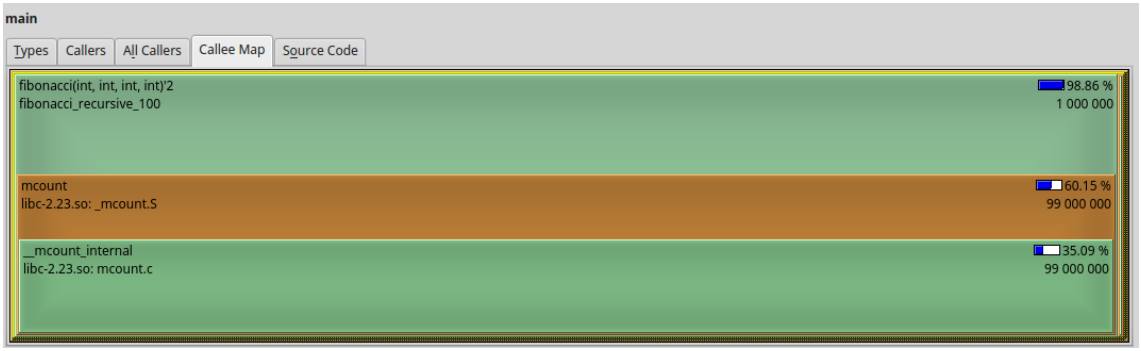
Abaixo estão apresentados o uso de memória para cada um dos três parâmetros do método Fibonacci, nota-se que o uso de memória não tende a mudar mesmo aumentando os parâmetros, mas o número de instruções aumentou consideravelmente.







Abaixo está apresentado o tempo gasto em cada parte do algoritmo.



Considerações Finais

Levando em consideração todos os dados apresentados nesse trabalho pode-se concluir que é dificil escolher um **melhor algoritmo de ordenação** pois embora o Merge Sort seja o mais eficiente, ele tem um gasto de memória mais alto, mesmo que seja pequena essa diferença, e possui uma alta complexidade, então, depende muito da situação, se o problema exigir um nivel de eficácia alto e não se importar com a complexidade do algoritmo de ordenação, então Merge Sort pode ser uma ótima solução, mas se for um problema simples e de baixa quantidade de dados, pode-se observar que em vetores pequenos os algoritmos se igualam, podendo-se assim levar como caracteristica principal a facilidade de implementação, e nisso o Bubble Sort e o Insertion Sort ganham disparadamente, embora o Bubble Sort seja altamente ineficaz, ele é muito simples de ser implementado, o Insertion estaria em um nivel mais médio, mesmo tendo também baixa complexidade de implementação, seria então em grande parte dos casos a melhor opção por manter um balanceamento entre otimização e baixa complexidade.

Os **métodos Fibonacci** mostraram que a versão sem recursividade se mostra melhor em todos os sentidos, embora o algoritmo utilizado nos testes foi que somente retorna um elemento do Fibonacci e não um vetor com todos elementos até aquele ponto, foi o bastante para mostrar que o método sem recursividade se mostra mais eficiente, mas também devemos levar em consideração que o método com recursividade apresenta uma solução mais elegante e legível, então novamente, depende muto da situação, se a otimização for um requisito muito importante, então a melhor opção seria sem recursividade, mas caso contrário, a solução com recursividade deixaria o código com entendimento mais fácil.