

**COMPARAÇÃO E ANÁLISE DO DESEMPENHO DE ALGORITMOS DE ORDENAÇÃO**

**Integrantes:** Matheus G. de Paula, RA: 122.043

**Data**: 30/04/2019

São José dos Campos

**Resumo**

O intuito do relatório é analisar o desempenho de diversos algoritmos de ordenação, ou *sort*, com dados previamente determinados ou não. Como método de teste iremos utilizar 3 (três) tipos de dados, sendo estes ordenados de forma crescente, decrescente e não ordenados (aleatório). Além dos 3 tipos de dados teremos ainda 4 tamanhos para os dados, sendo estes 1.000 (mil), 10.000 (dez mil), 100.000 (cem mil), 1.000.000 (um milhão). O requisito mínimo para atender o relatório era de que todos estes dados deveriam estar em um intervalo de -2.000.000.000 (menos dois bilhões) a 2.000.000.000 (dois bilhões).

Os algoritmos selecionados para o grupo I foram: Bubble Sort e Selection Sort; Para o grupo II foram: Mergesort e Heapsort; Para o grupo III foram Counting Sort e Radix Sort;

Após a coleta dos tempos irei comparar os resultados entre os diversos algoritmos de ordenação e abordar as principais diferenças**.**

**Introdução**

O método de medição de tempo será feito pela biblioteca <time.h>, com a função clock() para calcular o tempo inicial (antes de começar a aplicação) e o tempo final (após calcular a aplicação), tendo o código entre esses medidores.



figura 1. Exemplo de leitura dos números

Note que isso retorna o tempo como um tipo ponto flutuante. Assim, sendo mais preciso que um segundo. A precisão depende da arquitetura do sistema, em sistemas modernos você consegue facilmente 10ms ou menos, mas em sistemas Windows antigos (Windows 98) eles podem chegar a 60ms.

**Amostragem do Tempo**

Para se obter o tempo, farei o uso da média entre 3 testes para cada tipo de dado e para cada tamanho também.

**Leitura dos Arquivos**

Os arquivos com os dados foram colocados em arquivos .txt, eles então são lidos através do seguinte comando:



figura 2. Exemplo de leitura dos números

onde primeiro crio o arquivo FILE, depois abro o arquivo desejado para leitura com o fopen(). Caso a abertura falhe eu retorno 1 como aviso de erro. Após eu crio o array com alocamento de memória dinâmica (malloc). Em seguida eu leio todos os valores do arquivo .txt com o fscanf e coloco no array, após o término eu fecho o arquivo e executo a função desejada.

**Especificações Técnicas da Máquina**

* **Processador:** Intel® Core™ i5-5200U Dual Core 2.2 GHz
* **Sistema Operacional:** Windows 10 Home
* **Tela:** Tela LED HD Widescreen, com resolução de 1366 x 768
* **Cache:** 3 mb L3
* **Memória RAM:** 8 GB DDR3L 1600 MHz
* **HD:** 1 TB 5400 RPM + SSD M2 2280 240gb
* **PLACA DE VÍDEO**: NVIDIA® GeForce® 920MX Graphics 2 GB gDDR3 Dedicada

**GRUPO I**

**Bubble Sort**

O Bubble Sort, ou ordenação por flutuação é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, e a cada passagem fazer “flutuar” para o topo o maior elemento da sequência. Esse movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vetor. No pior caso, são feitas n² operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Bubble Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,130 | 11,972 | 1234,705 |

Tabela 1. Bubble Sort - Valores obtidos para ordenação crescente

Os seguintes resultados, em segundos, são para o método de Bubble Sort com os dados ordenados de forma decrescente. Existe um porém nesse algoritmo, ele não comportou usar um milhão de números, não foi possível obter o valor médio de um milhão:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,004 | 0,32 | 30 | ? |

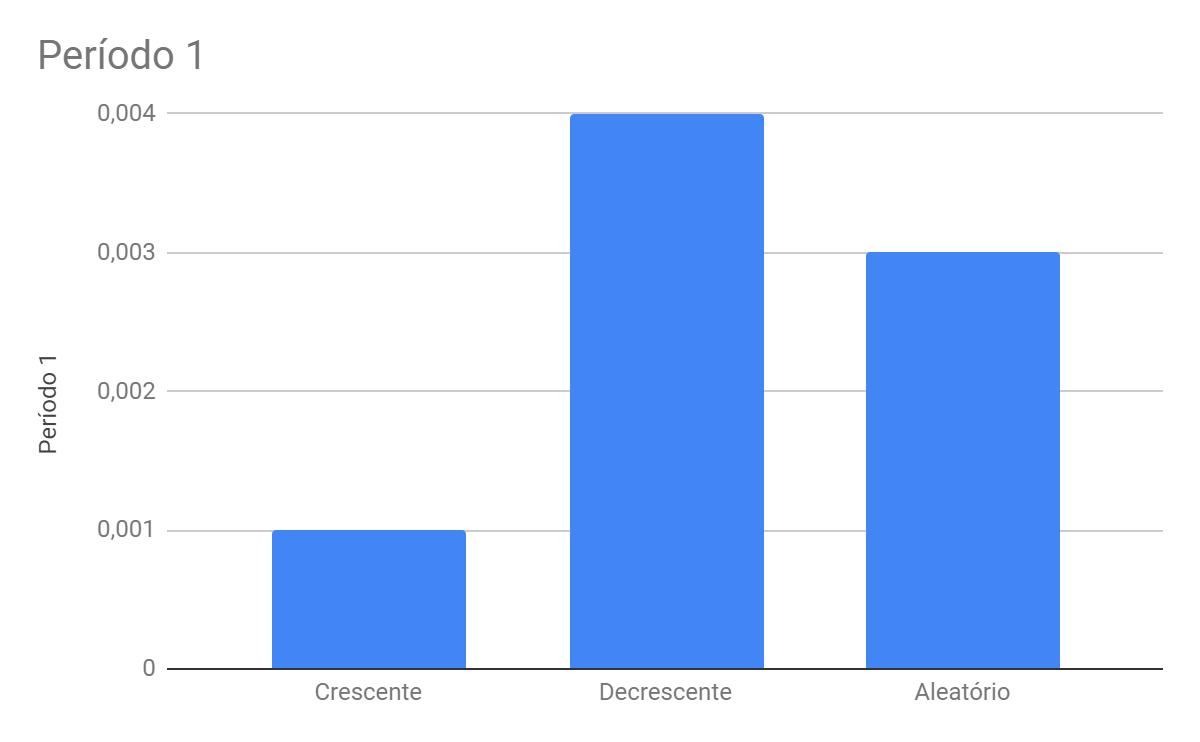
Tabela 2. Bubble Sort - Valores obtidos para ordenação decrescente

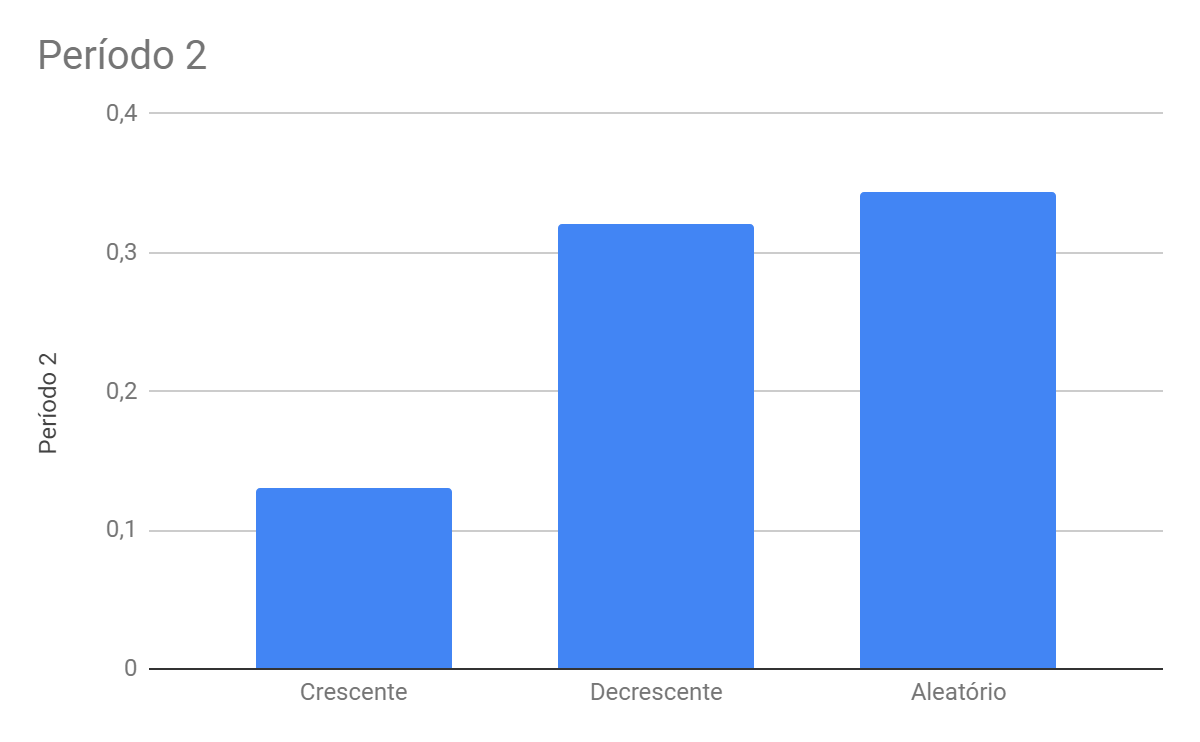
Os seguintes resultados, em segundos, são para o método de Bubble Sort com os dados aleatórios:

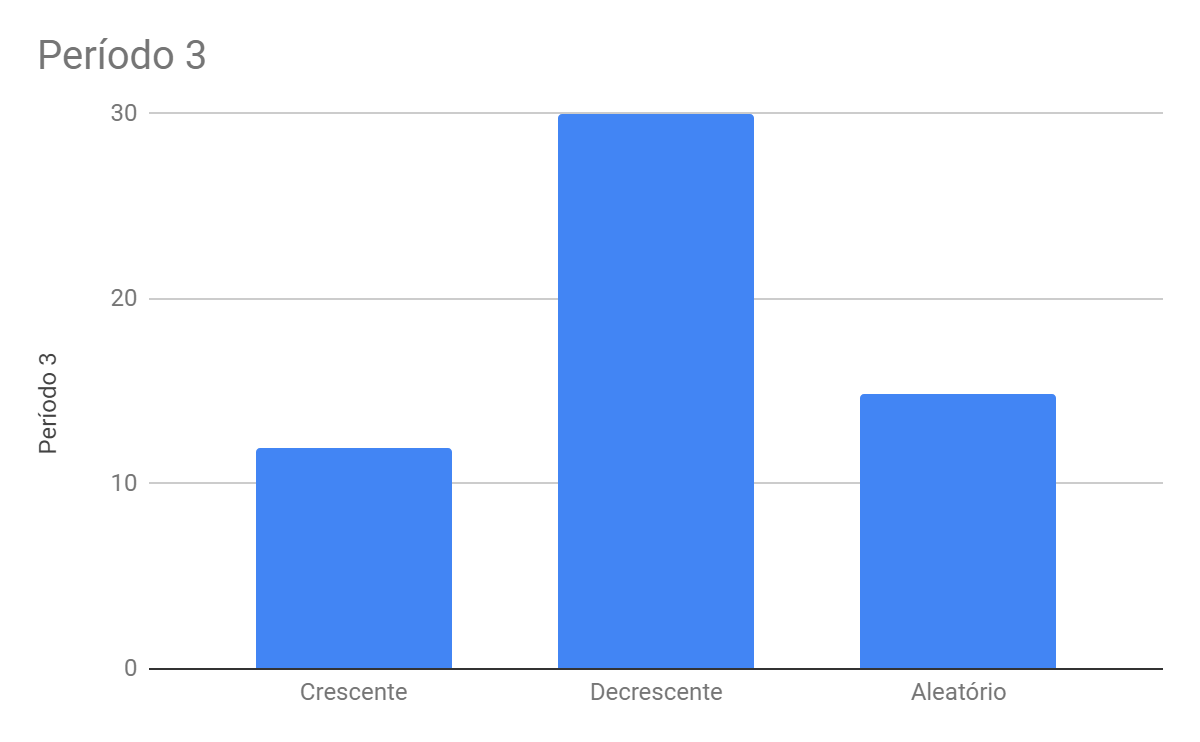
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,003 | 0,345 | 14,022 | 1234,875 |

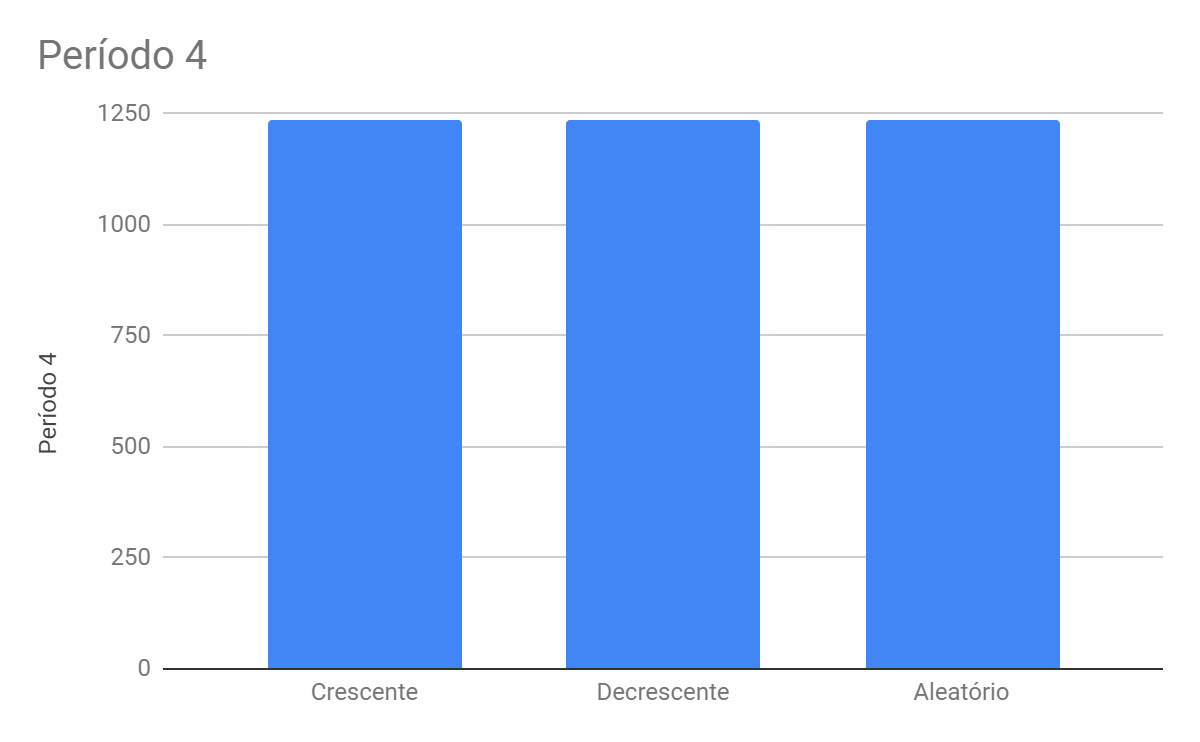
Tabela 3. Bubble Sort - Valores obtidos para valores aleatórios

Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.









Podemos perceber que em geral o Bubble sort possui uma dificuldade extrema quando se trata de dados Decrescentes, pois é seu pior caso, levando notadamente muito mais tempo.

**Insertion Sort**

Insertion Sort, ou ordenação por inserção, é o algoritmo de ordenação que, dado uma estrutura (*array*, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Podemos fazer uma comparação do Insertion Sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas. Esta é a ideia por trás da ordenação por inserção. Percorra as posições do *array*, começando com o índice 1 (um). Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto no *subarray* ordenado à esquerda daquela posição.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Insertion Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,000 | 0,003 | 0,033 | 0,360 |

Tabela 1. Insertion Sort - Valores obtidos para ordenação crescente

Os seguintes resultados, em segundos, são para o método de Bubble Sort com os dados ordenados de forma decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,150 | 14,150 | ? |

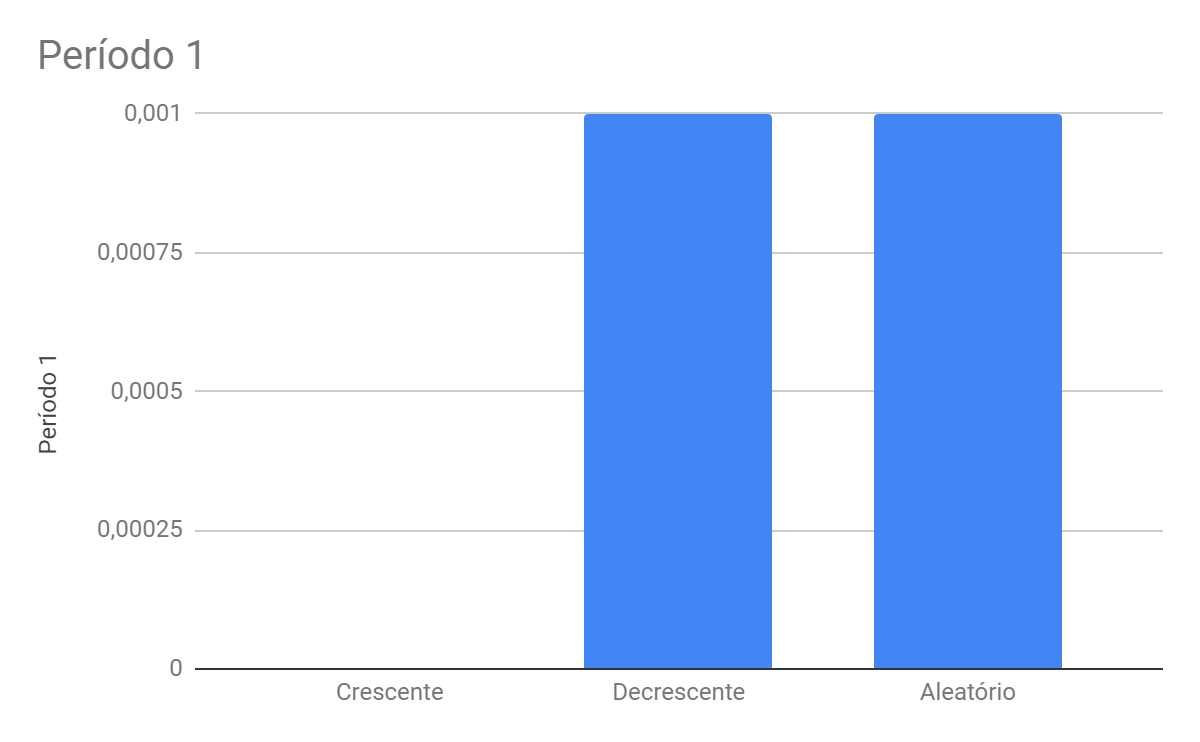
Tabela 2. Insertion Sort - Valores obtidos para ordenação decrescente

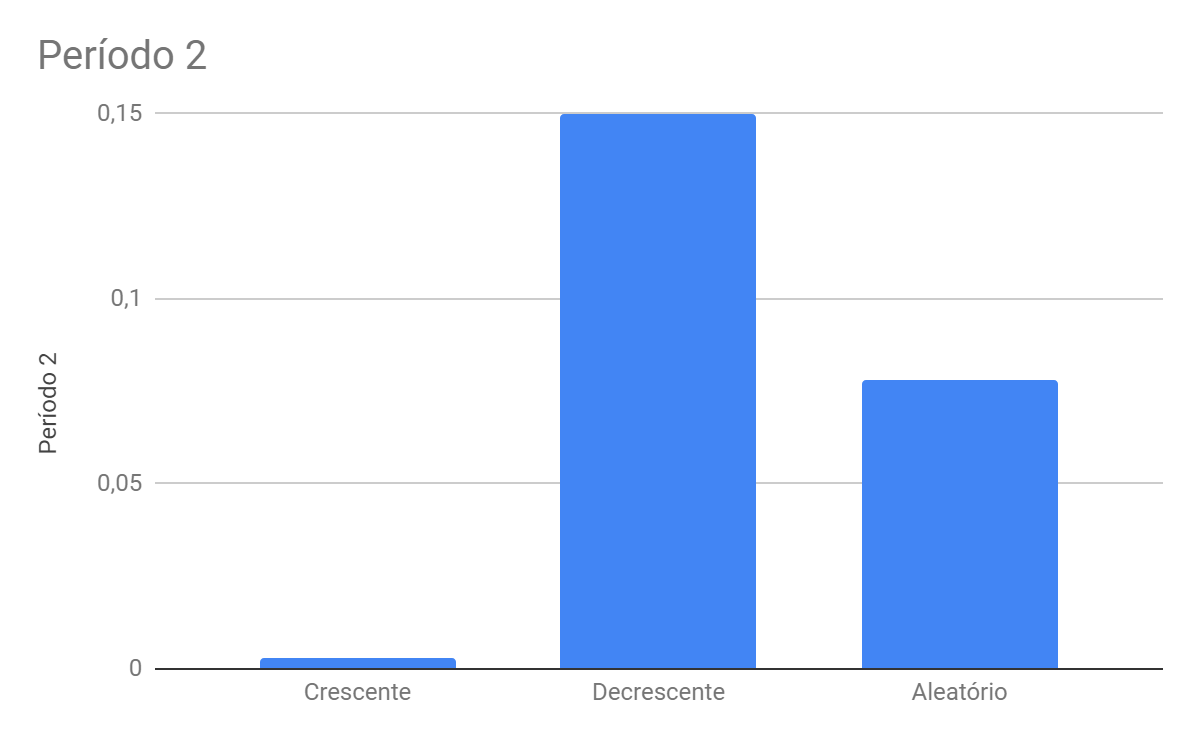
Os seguintes resultados, em segundos, são para o método de Insertion Sort com os dados aleatórios:

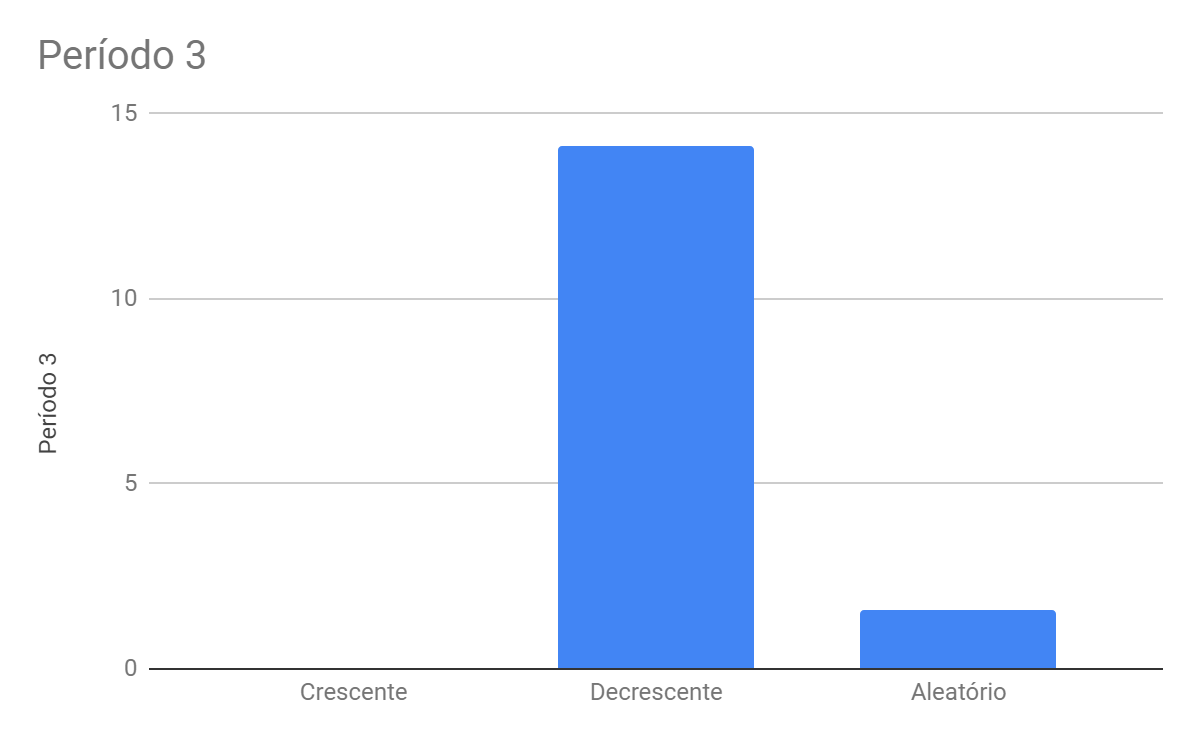
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,078 | 1,594 | 16,5 |

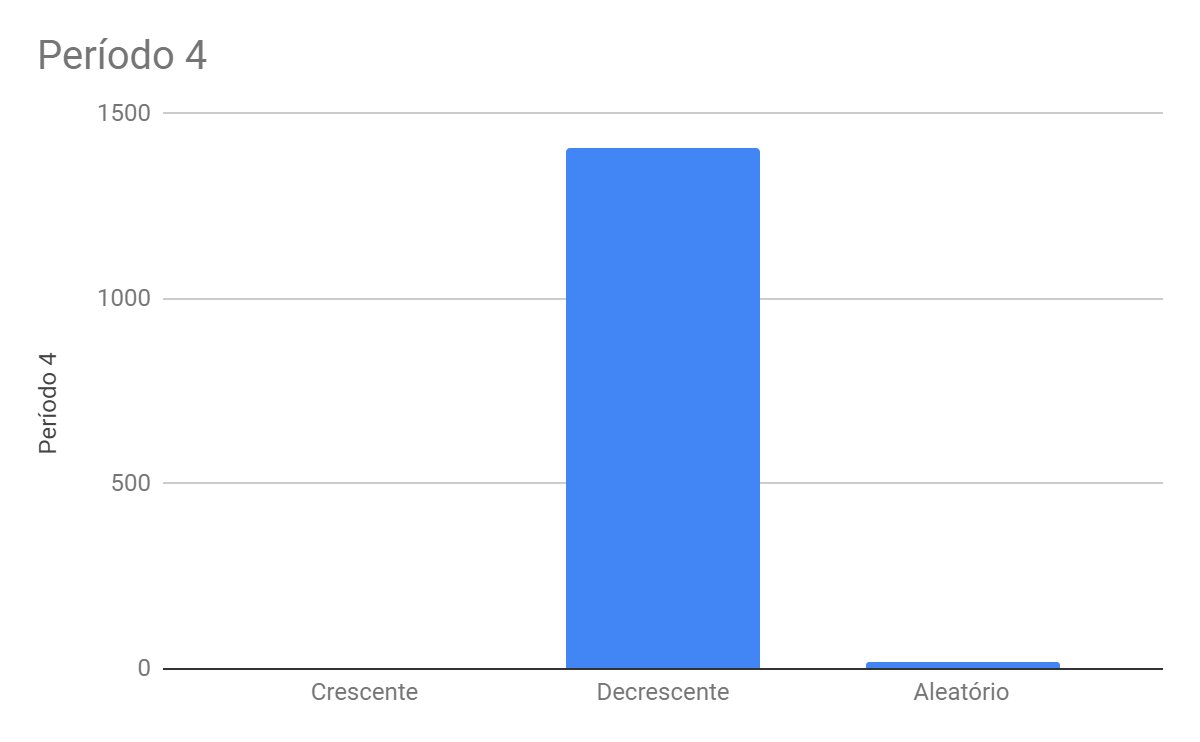
Tabela 3. Insertion Sort - Valores obtidos para valores aleatórios

Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.









Podemos perceber que assim como o Bubble sort, o Insertion sort possui uma dificuldade extrema quando se trata de dados Decrescentes, pois é seu pior caso, levando notadamente muito mais tempo.

**GRUPO II**

**Merge Sort**

O Merge sort, ou ordenação por mistura. é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

Neste algoritmo não foi possível obter o valor médio para o N = 1.000.000, acredito que tenha sido o tamanho que tenha impactado.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Merge Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,0 | 0,005 | 0,045 | ? |

Tabela 1. Merge Sort - Valores obtidos

Os seguintes resultados, em segundos, são para o método de Merge Sort com os dados ordenados de forma decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,003 | 0,046 | ? |

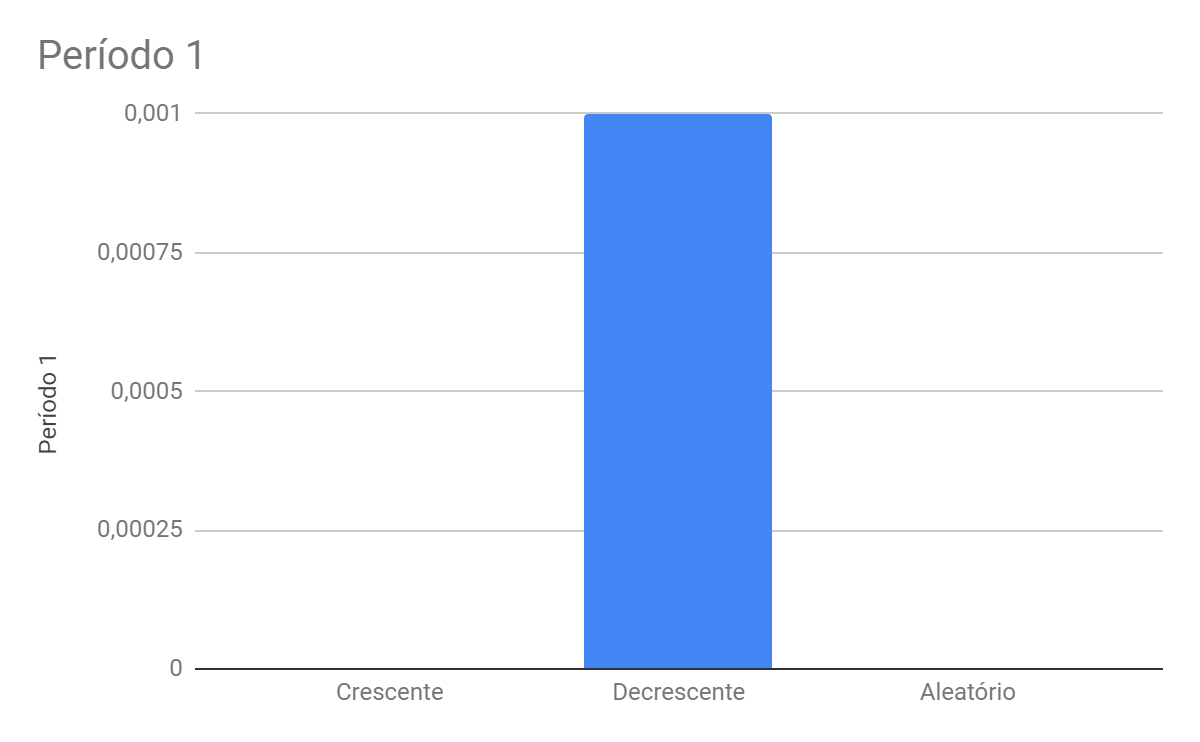
Tabela 2. Merge Sort - Valores obtidos

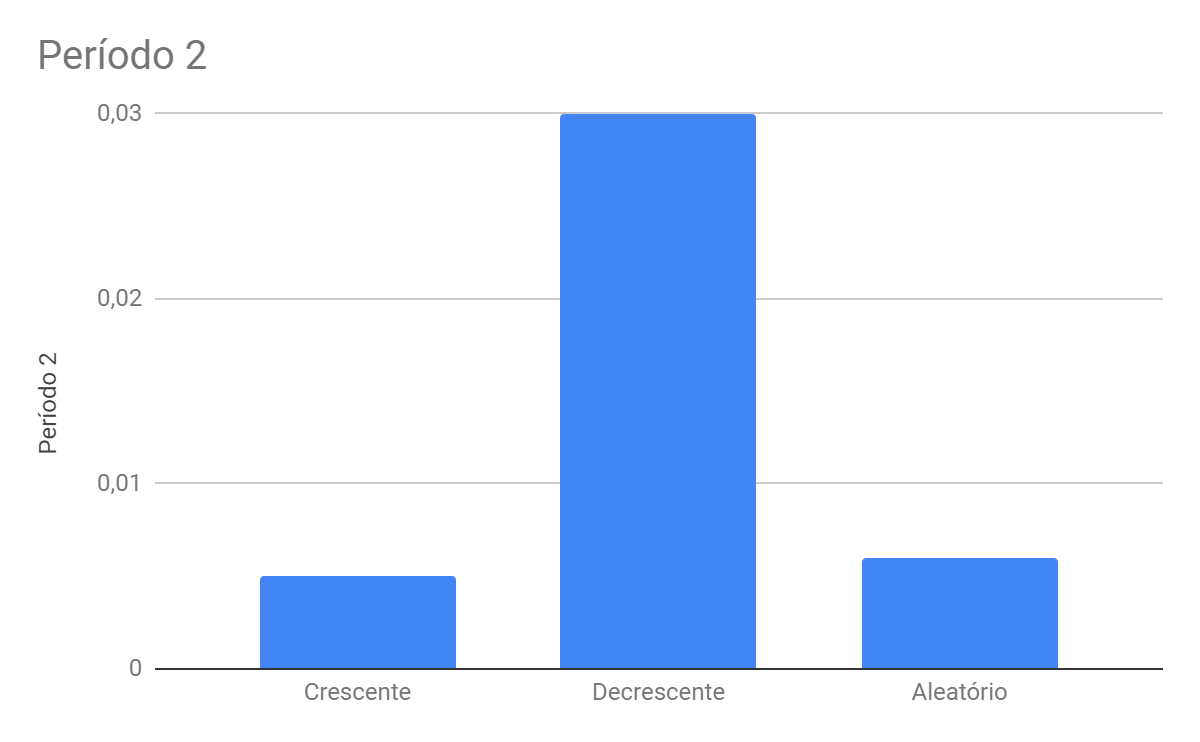
Os seguintes resultados, em segundos, são para o método de Merge Sort com os dados aleatórios:

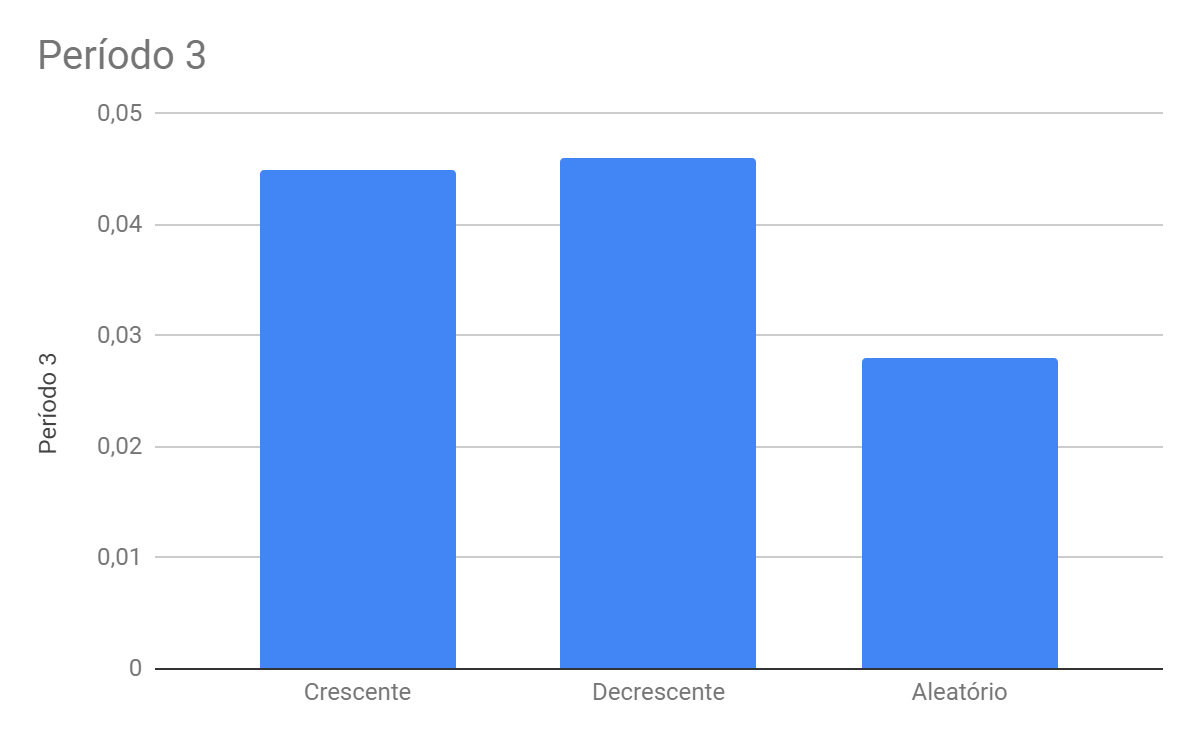
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,0 | 0,006 | 0,028 | ? |

Tabela 3. Merge Sort - Valores obtidos

Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.







Podemos perceber que em geral o Merge sort possui uma linearidade na diferença de tempo entre os períodos, não possuindo quase nenhuma diferença de um para o outro.

**Heap Sort**

O algoritmo heapsort é um algoritmo de ordenação generalista e faz parte da família de algoritmos de ordenação por seleção.

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio. Alguns algoritmos de ordenação rápidos têm desempenhos espetacularmente ruins no pior cenário, quer em tempo de execução, quer no uso da memória. O heapsort trabalha no lugar e o tempo de execução em pior cenário para ordenar *n* elementos é de O(*n* lg *n*). Lê-se logaritmo (ou log) de "n" na base 2. Para valores de *n*, razoavelmente grandes, o termo log *n* é quase constante, de modo que o tempo de ordenação é quase linear com o número de itens a ordenar.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Heap Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,000 | 0,004 | 0,056 | 0,635 |

Tabela 1. Heap Sort - Valores obtidos para ordenação crescente

Os seguintes resultados, em segundos, são para o método de Heap Sort com os dados ordenados de forma decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,00 | 0,005 | 0,058 | 0,642 |

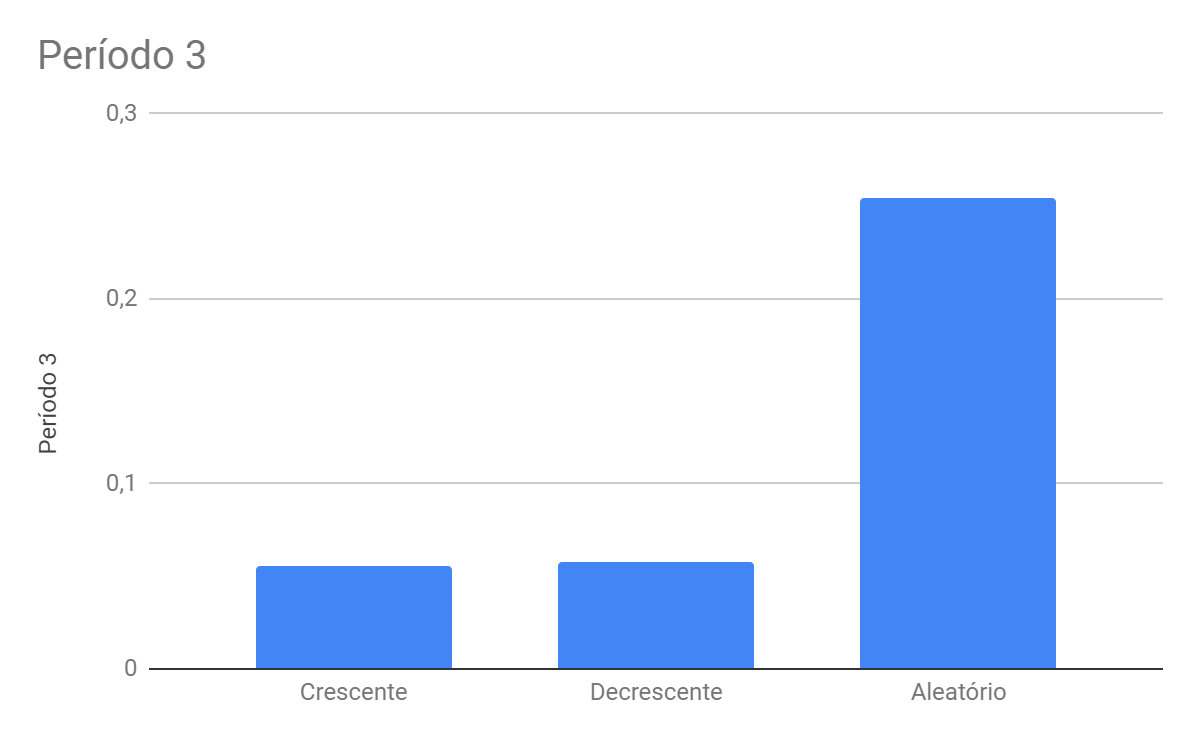
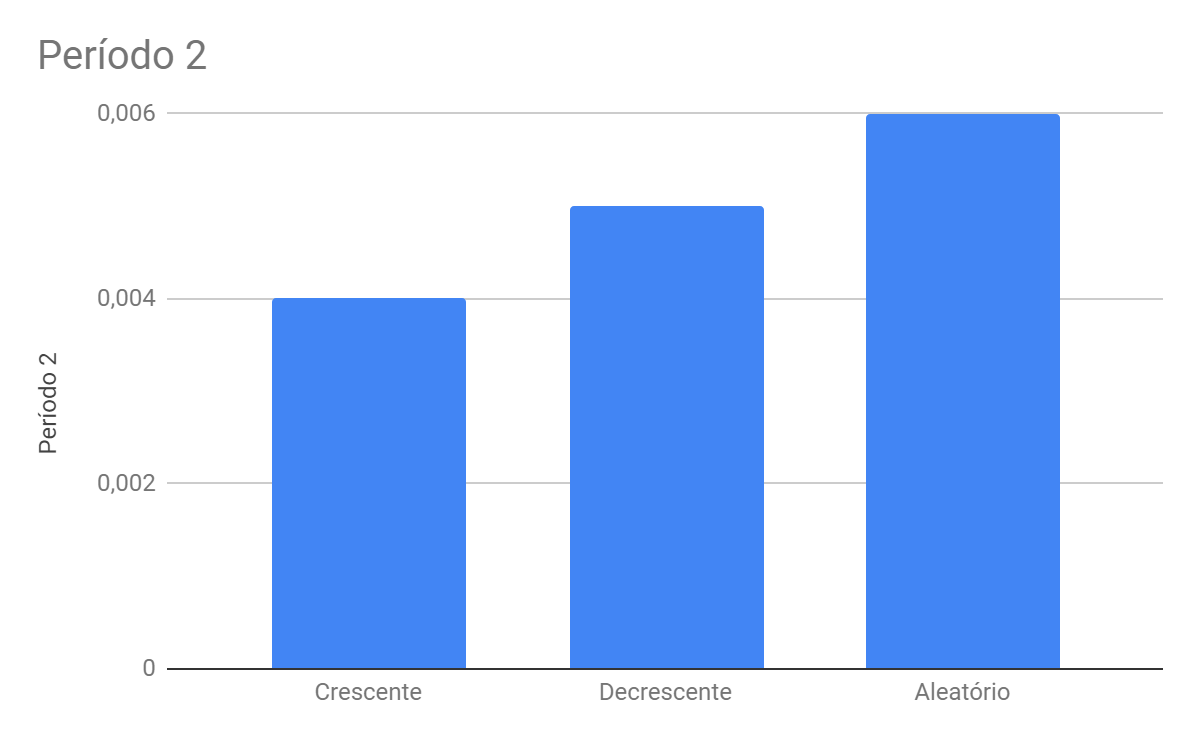
Tabela 2. Heap Sort - Valores obtidos para ordenação decrescente

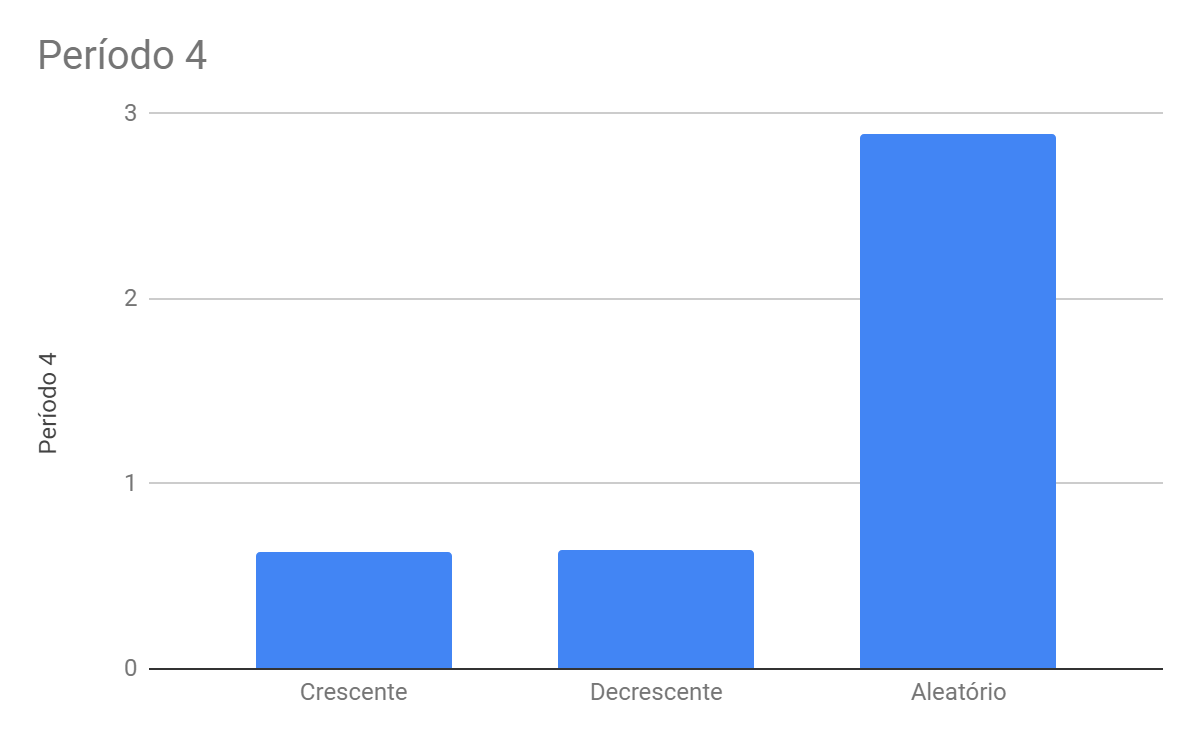
Os seguintes resultados, em segundos, são para o método de Heap Sort com os dados aleatórios:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,000 | 0,006 | 0,254 | 2,889 |

Tabela 3. Heap Sort - Valores obtidos para valores aleatórios

Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.





Podemos perceber que o Heap Sort é um bom algoritmo de ordenação entre os vistos até o momento, levando menos de 3 segundos no pior dos casos.

**GRUPO III**

**Counting Sort**

Counting sort é um algoritmo de ordenação estável cuja complexidade é O(n). As chaves podem tomar valores entre 0 e M-1. Se existirem k0 chaves com valor 0, então ocupam as primeiras k0 posições do vetor final: de 0 a k0-1.

A ideia básica do counting sort é determinar, para cada entrada x, o número de elementos menor que x. Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no array de saída. Por exemplo, se há 17 elementos menor que x, então x pertence a posição 18. Esse esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor, uma vez que nós não queremos colocar eles na mesma posição.

Para o Counting sort eu tive que desenvolver um arquivo exclusivo para leitura (Eles possuem o seguinte formato: numbers\_1000\_RAND\_2, variando o número para cada N), pois a natura do algoritmo impede que os números aleatórios tenham uma diferença muito alta entre eles, assim eu diminui o intervalo de valor para até 100.000 (cem mil) unidades nos casos aleatórios. Excluindo esse caso e o do Radix Sort, os demais foram testados usando o mesmo conjunto de dados que os anteriores.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Counting Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,00 | 0,002 | 0,034 | 0,386 |

Tabela 1. Counting Sort - Valores obtidos

Os seguintes resultados, em segundos, são para o método de Counting Sort com os dados ordenados de forma decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,00 | 0,003 | 0,036 | 0,38 |

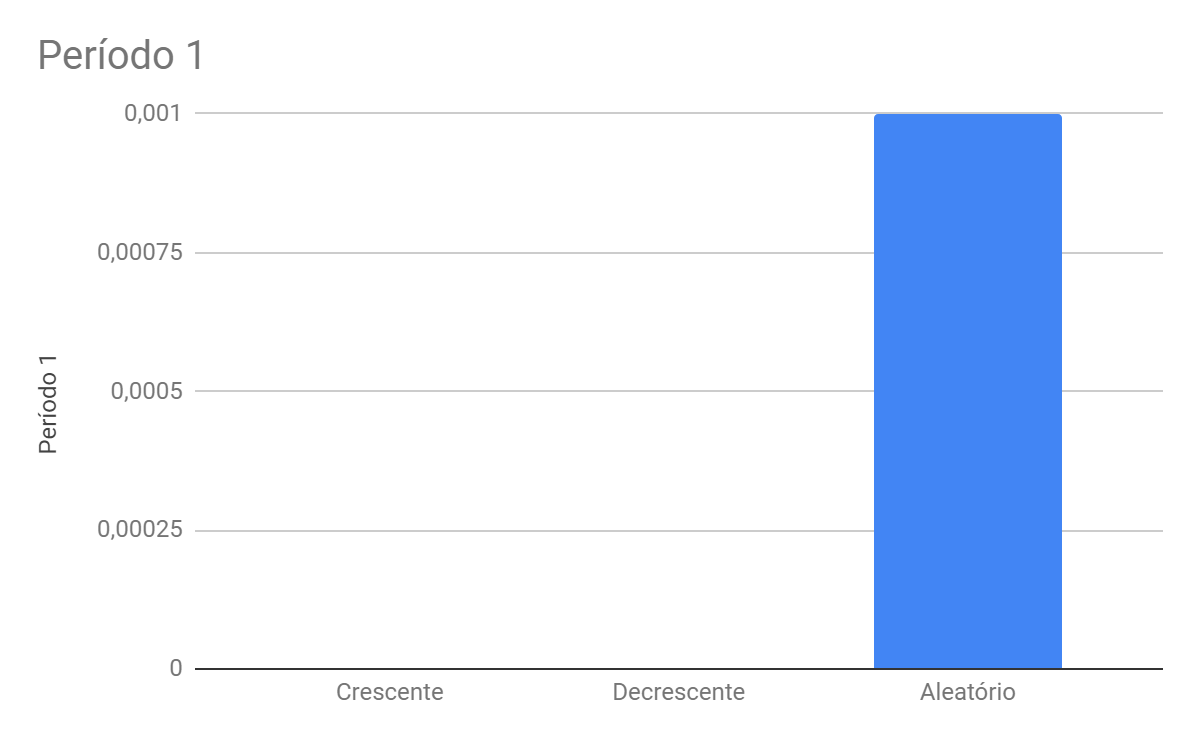
Tabela 2. Counting Sort - Valores obtidos

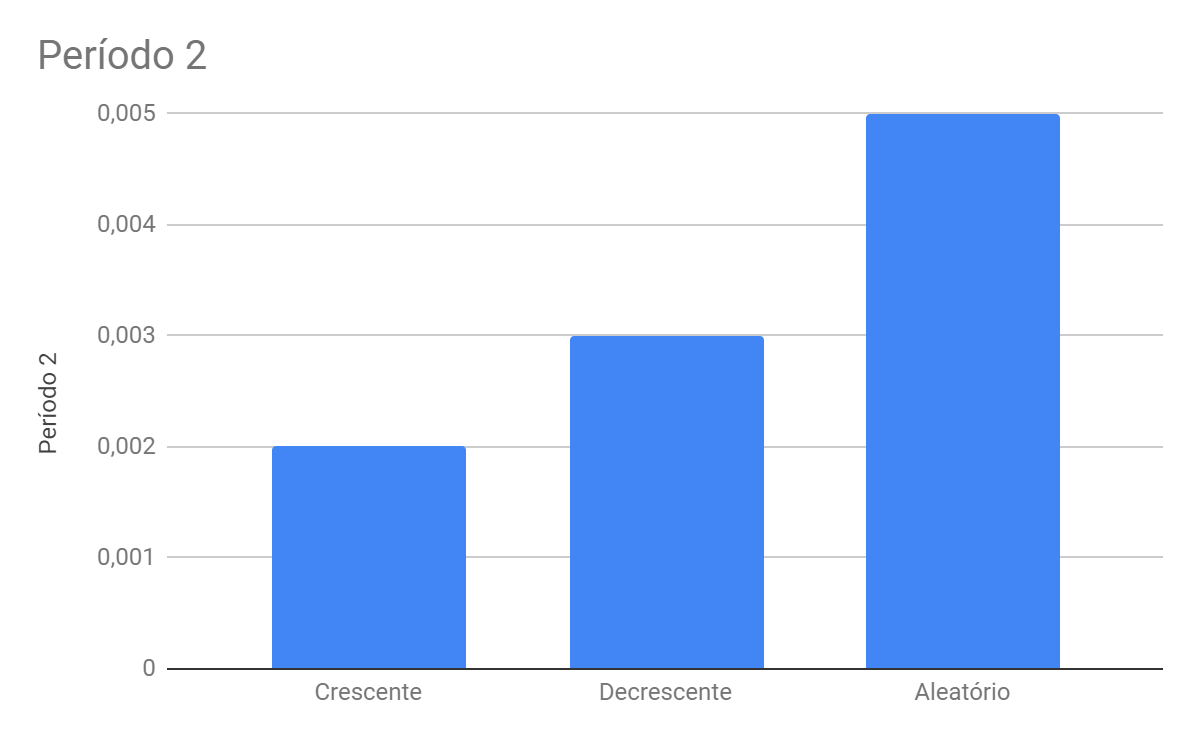
Os seguintes resultados, em segundos, são para o método de Counting Sort com os dados aleatórios:

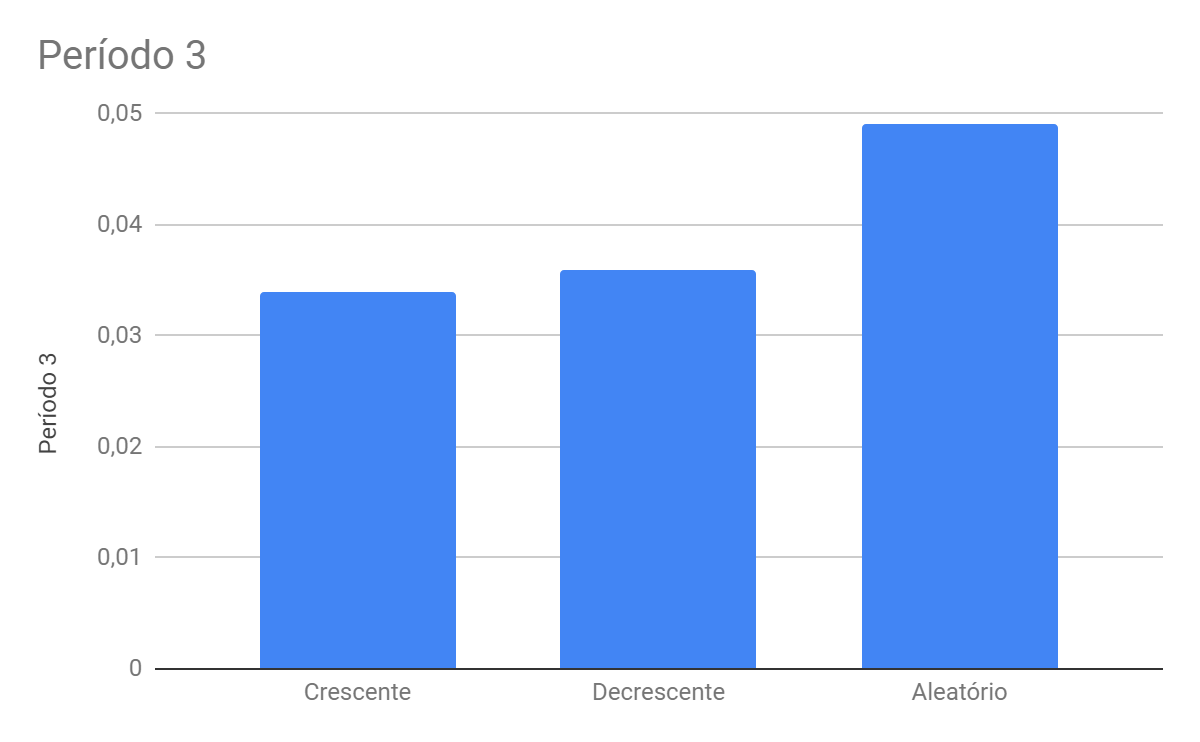
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,005 | 0,049 | 0,726 |

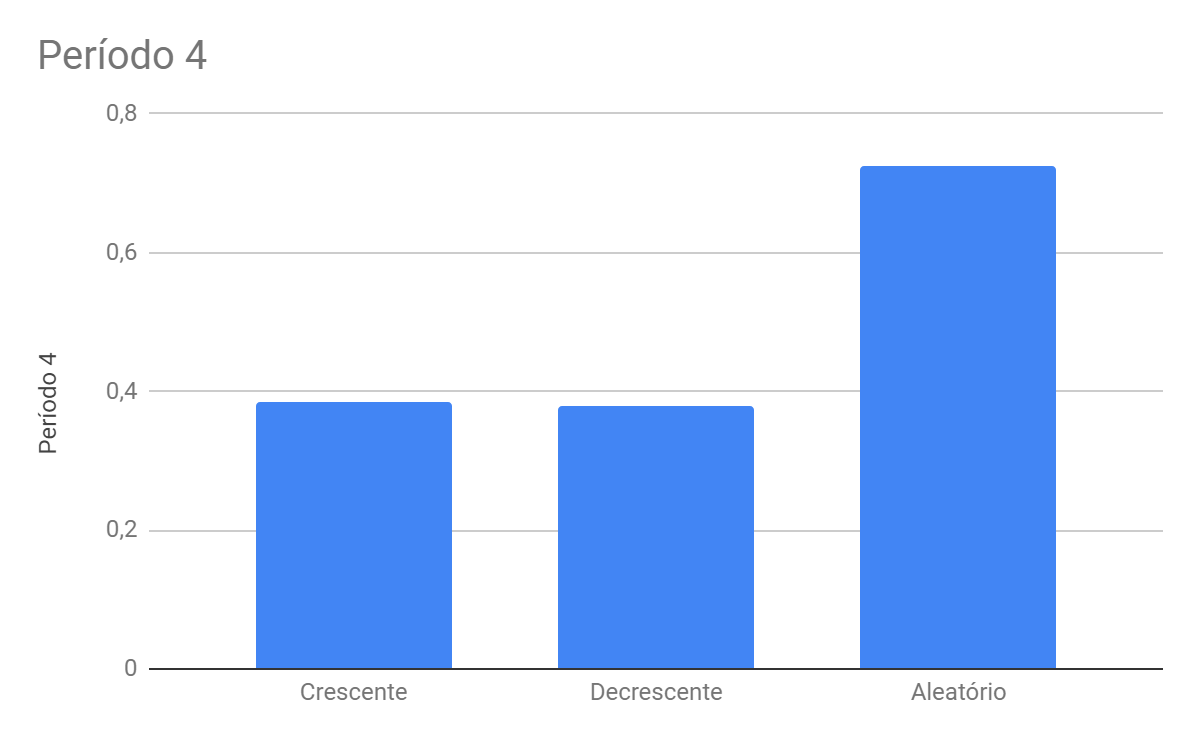
Tabela 3. Counting Sort - Valores obtidos

Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.









Podemos perceber que em geral o Counting sort possui uma linearidade na diferença de tempo entre os períodos, sendo um excelente algoritmo de ordenação, vemos também que a diferença fica na casa de centésimos de segundo, sendo assim pouco importante na visão geral. Percebi através dos meus teste que o modo aleatório de dados é o que mais consumiu tempo do código em média.

**Radix Sort**

O Radix sort é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o *radix sort* ordena estas chaves em qualquer ordem relacionada com a lexicografia.

Na ciência da computação, radix sort é um algoritmo de ordenação 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.

Da mesma forma que no Counting Sort eu utilizei o arquivo especial com um intervalo de dados reduzido para os casos de dados aleatórios, isso por conta da natureza do algoritmo.

Após seguir a metodologia descrita anteriormente obtive os seguintes resultados em segundos para o método de Radix Sort com os dados ordenados de forma crescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,001 | 0,005 | 0,074 | 0,706 |

Tabela 1. Radix Sort - Valores obtidos para ordenação crescente

Os seguintes resultados, em segundos, são para o método de Radix Sort com os dados ordenados de forma decrescente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,00 | 0,003 | 0,040 | 0,44 |

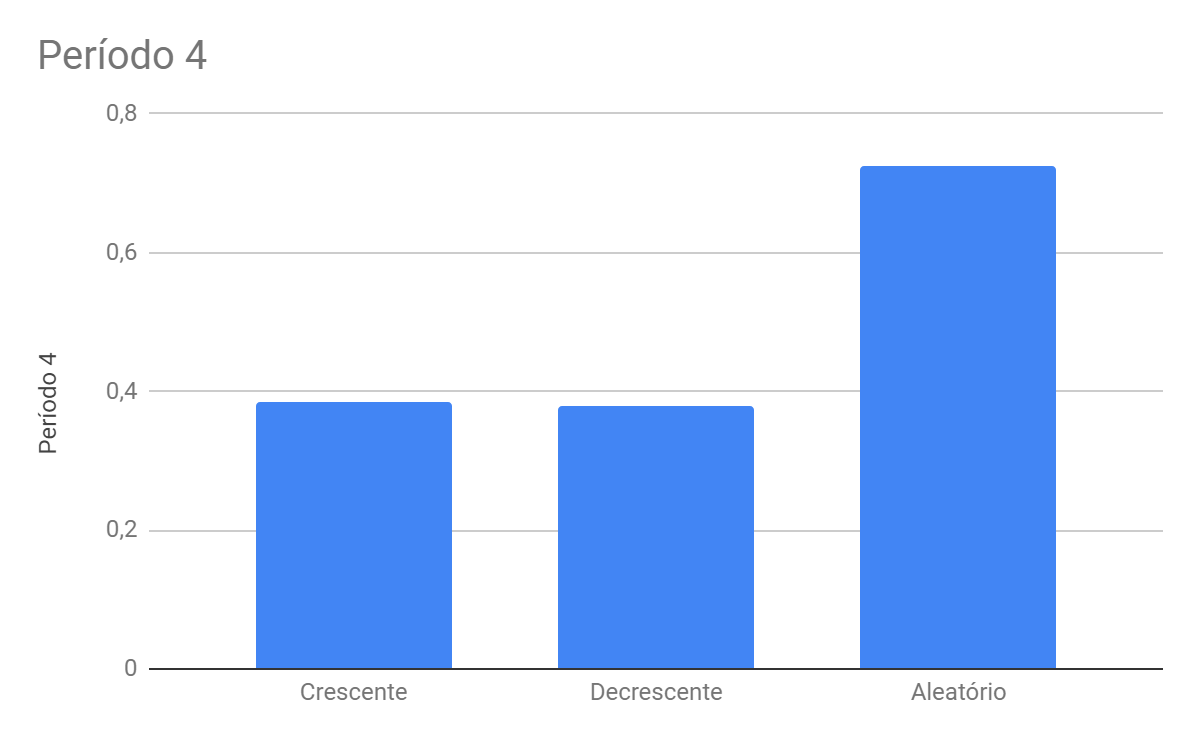
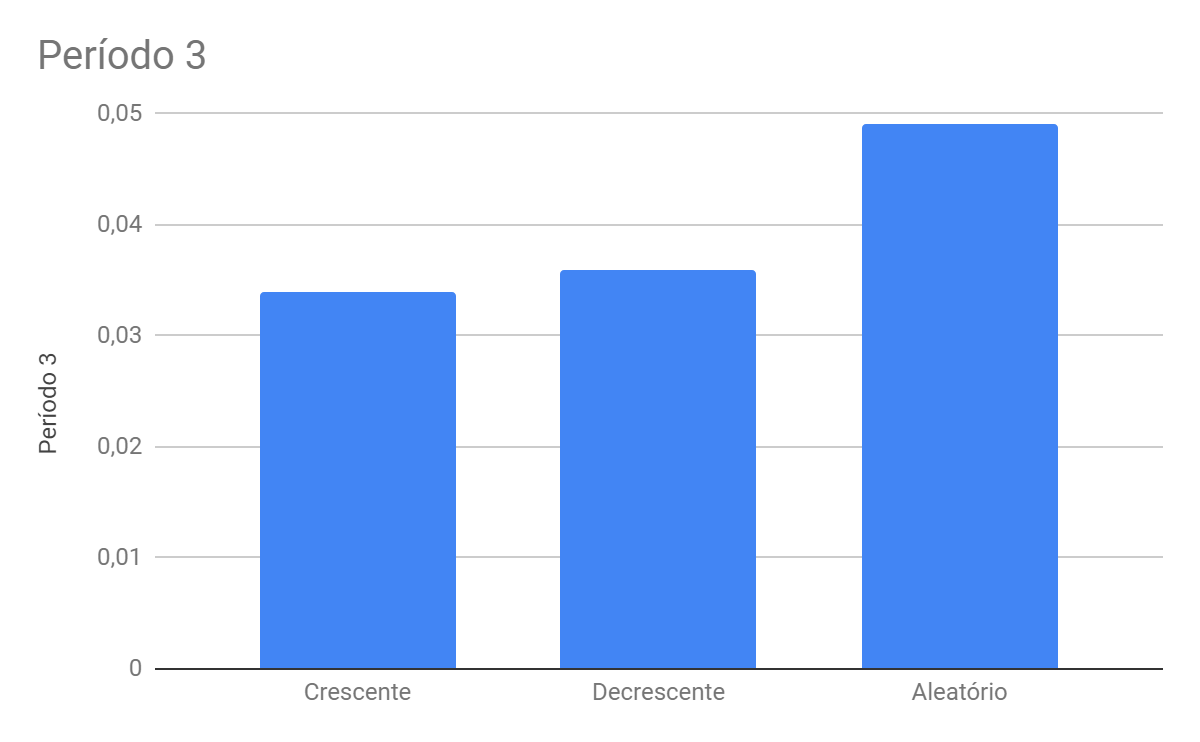
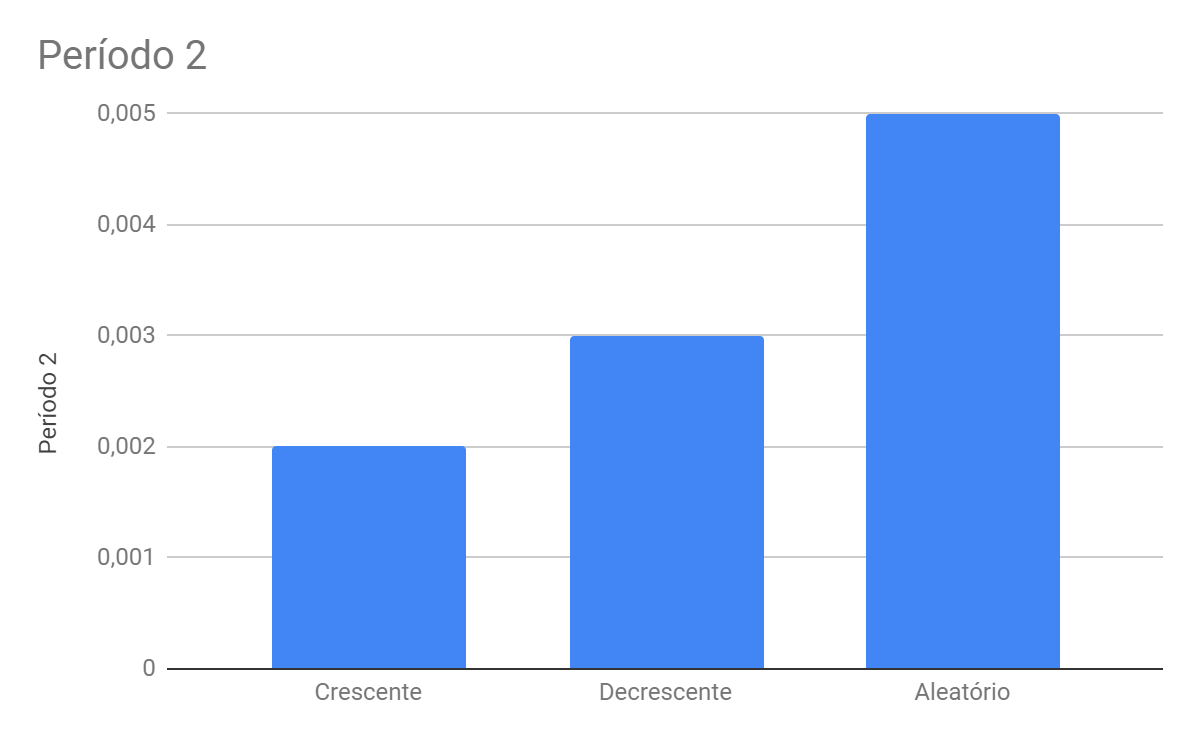
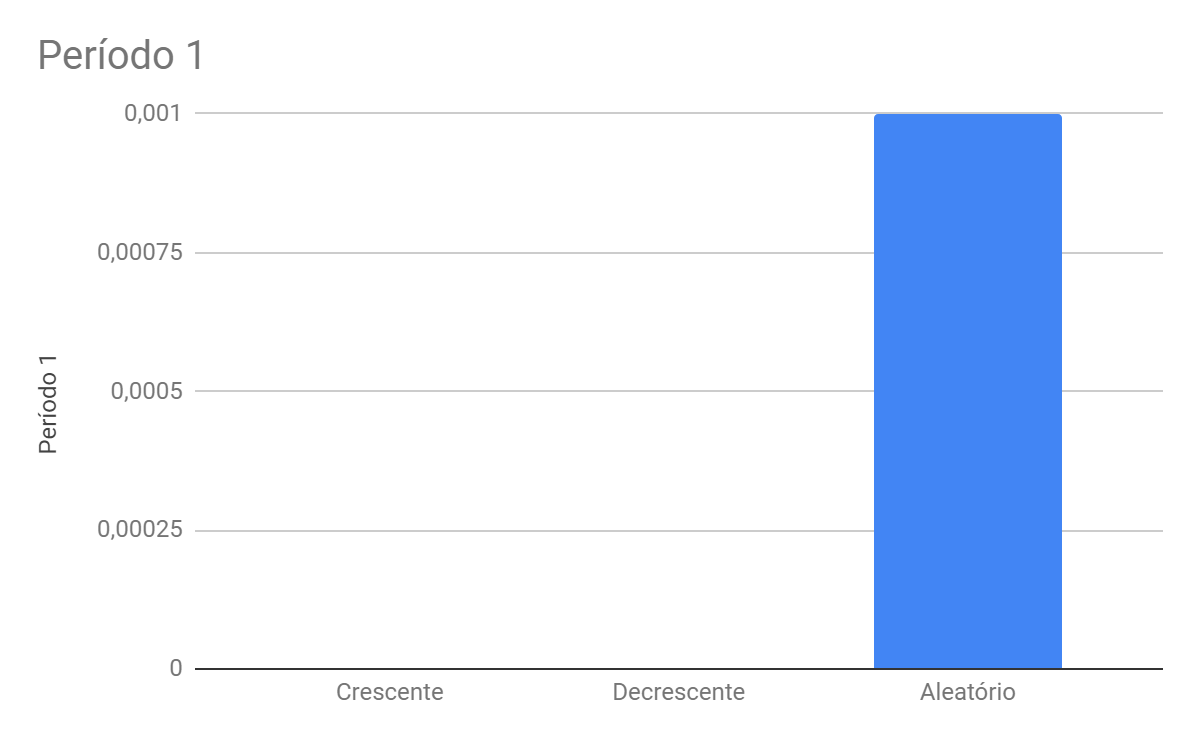
Tabela 2. Radix Sort - Valores obtidos para ordenação decrescente

Os seguintes resultados, em segundos, são para o método de Radix Sort com os dados aleatórios:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | 1.000 | 10.000 | 100.000 | 1.000.000 |
| MÉDIA | 0,000 | 0,007 | 0,064 | 0,7 |

Tabela 3. Radix Sort - Valores obtidos para valores aleatórios

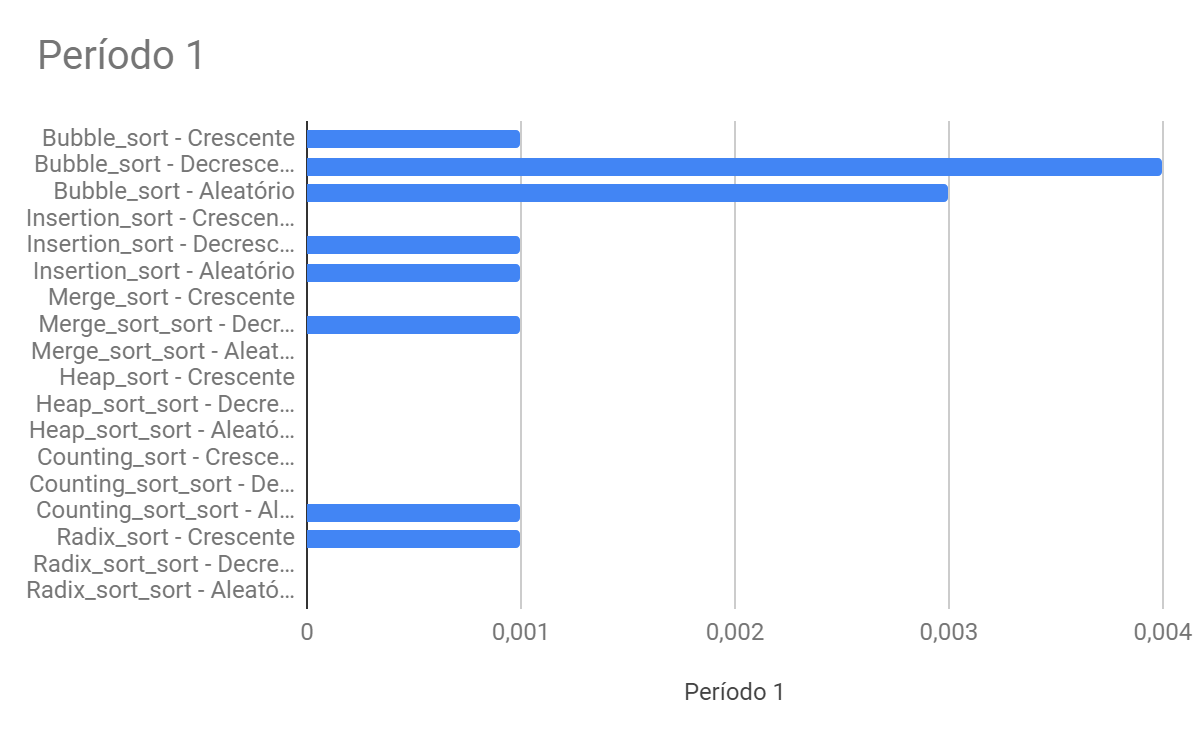
Abaixo estão dispostos os dados para cada N (período), sendo Período 1 igual a N de 1.000, Período 2 igual a N de 10.000 e assim por diante.



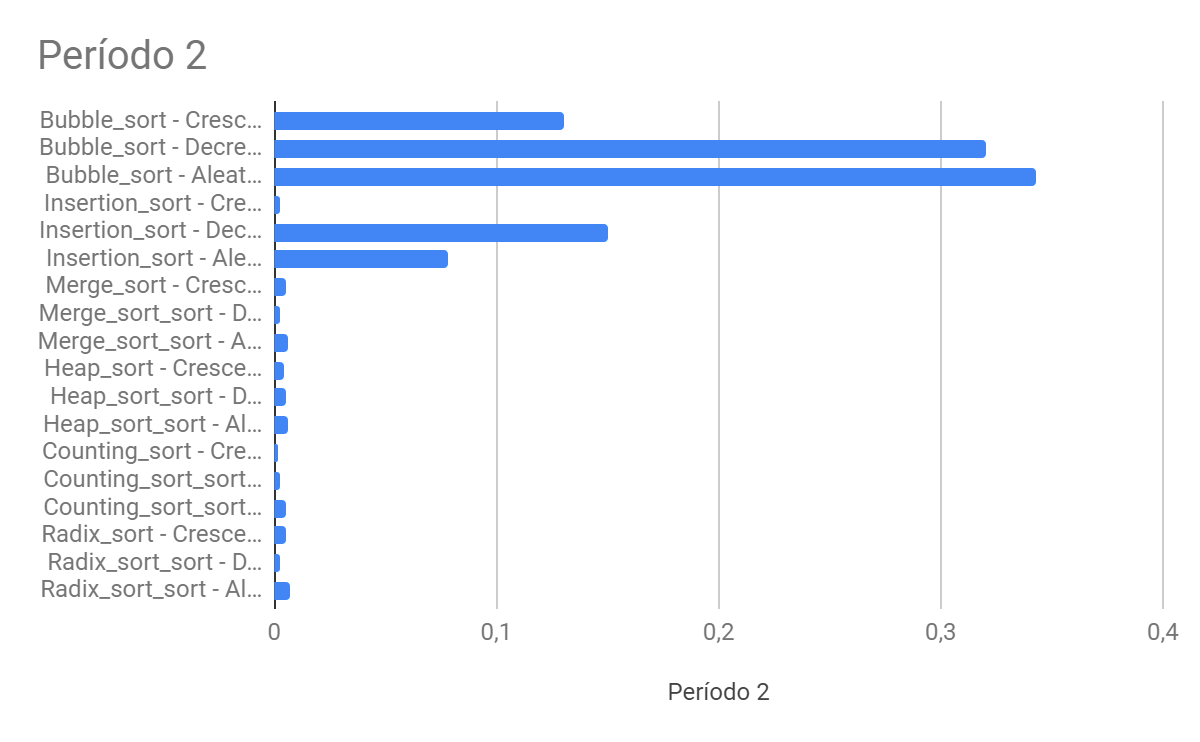
Podemos perceber que o Radix Sort é um excelente algoritmo de ordenação, possivelmente o melhor entre os que foram apresentados até o momento,levando menos de 1 segundo no pior dos casos, sendo que a diferença de um caso para o outro é irrisória, sendo na casa de décimos ou centésimos.

**Comparação entre os algoritmos**

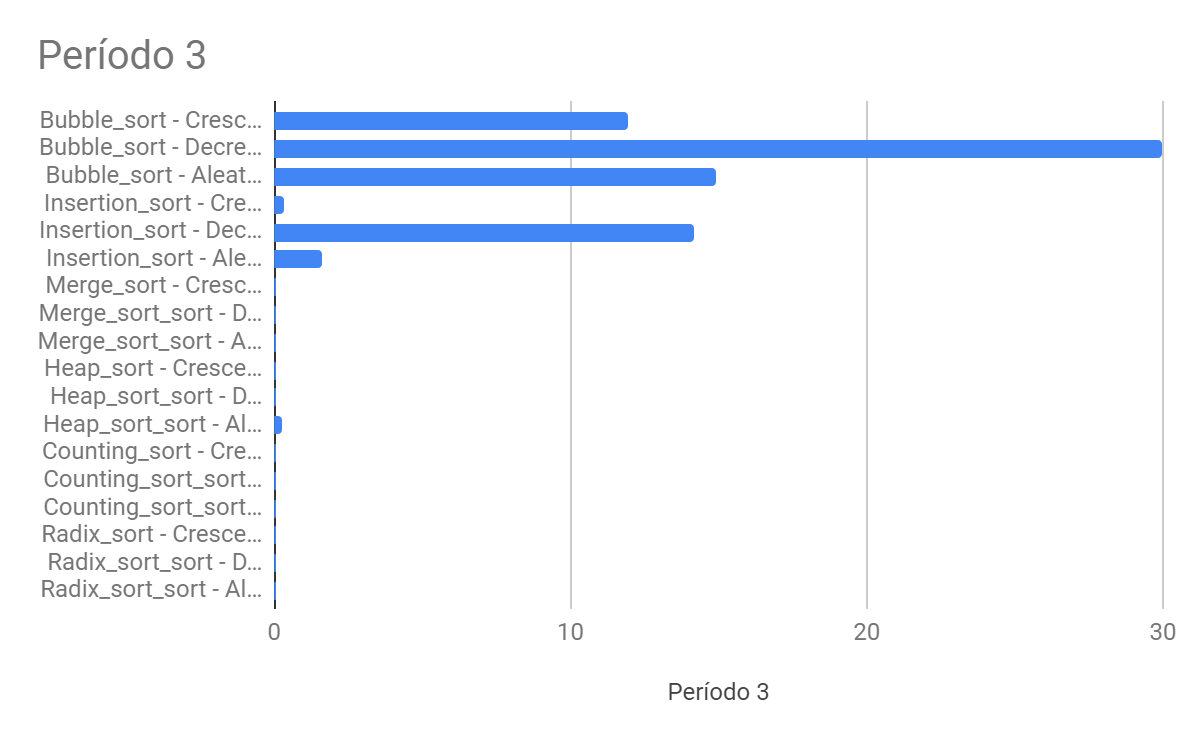
Para o período 1 (N = 1.000) temos as seguintes comparações:



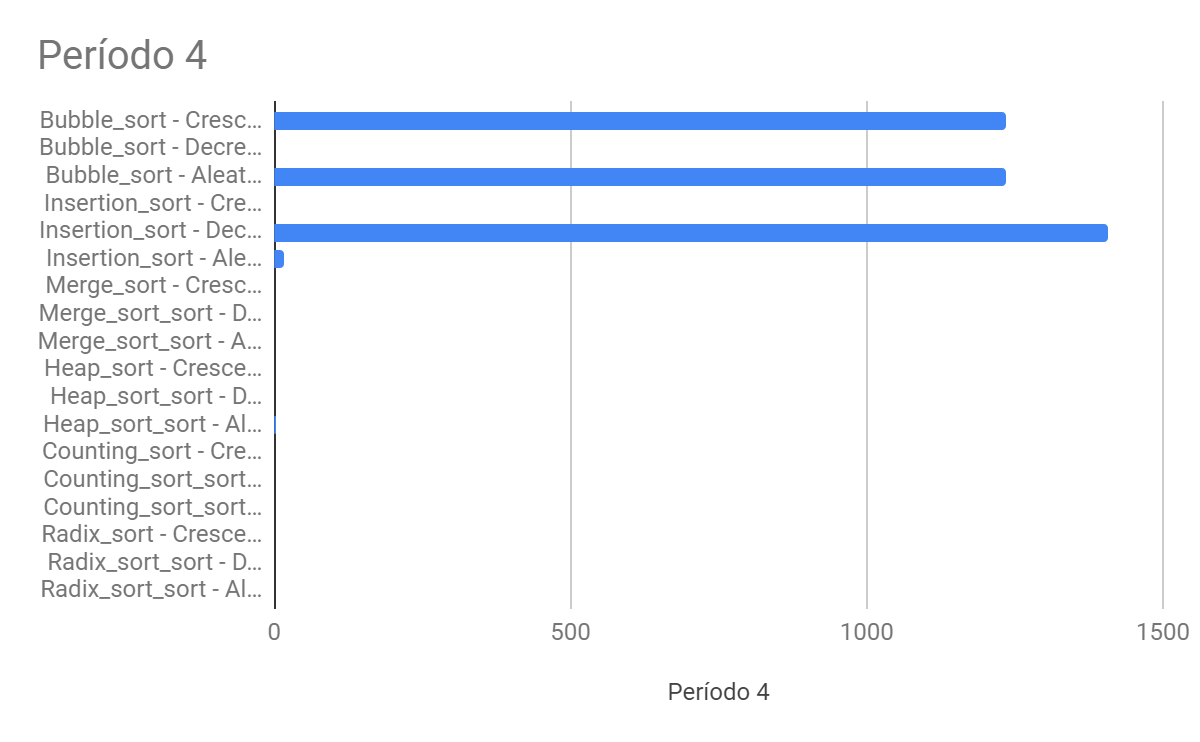
Para o período 2 (N = 10.000) temos as seguintes comparações:



Para o período 3 (N = 100.000) temos as seguintes comparações:



Para o período 4 (N = 1.000.000) temos as seguintes comparações:



**Conclusão**

Percebemos que no decorrer das escolhas dos grupos vemos uma melhora no desempenho dos algoritmos, sendo que a partir do merge sort temos uma melhora considerável. Podemos dizer que os algoritmos do grupo 3 são os melhores por serem os mais eficientes, vemos isso claramente no período 4 (N = 1.000.000).