# Lista de Exercícios 6 - CAP-241 2017

# Prof. Dr. Gilberto Ribeiro de Queiroz.

Aluno: Paulo Henrique Barchi[1a]

31 de maio de 2017

[1]paulobarchi@gmail.com

[a] Laboratório Associado de Computação e Matemática Aplicada (LAC)
Coordenação de Laboratórios Associados (CTE)
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos, SP - Brasil.

**Exercício 01.** Para a bateria de testes, foram implementados os métodos: *SelectionSort*, *InsertionSort*, *ShellSort*, *HeapSort*, *MergeSort*, *QuickSort* – os quais constam no arquivo `sort.hpp`. Além destes, também foram considerados os métodos do arquivo de cabeçalho `<algorithm>` da biblioteca `STL`: `std::sort`, `std::stable_sort` e `std::sort_heap`. As chamadas para os métodos da biblioteca STL também estão no arquivo `sort.hpp`.

Para cada um dos métodos de ordenação abordados neste trabalho, os testes envolveram vetores gerados de forma aleatória, vetores ordenados de forma crescente e decrescente. Os valores gerados aleatoriamente estão dentro do intervalo $[1, 100000]$. Foi implementado um menu para apresentação dos resultados de cada método.

No arquivo `exercicio01.cpp`, quando o usuário escolher um método, é chamada a função `sortTests` que aceita como parâmetro um funtor do respectivo método de ordenação escolhido. Esta função faz três chamadas à função `sortVectors`, cada chamada referente à ordenação dos vetores de entrada (aleatório, ordem crescente e decrescente). A função `sortVectors` tem dois funtores como parâmetro: o método de ordenação escolhido no menu, e a função para gerar o vetor de entrada.

Ainda na função `sortVectors`, o tamanho do vetor variou de $n = 8$ (isto é, $2^3$) a $n = 4096$ (isto é, $2^{12}$). Para cada tamanho $n$, foram feitas 100 execuções para obter-se o tempo médio em segundos de ordenação.

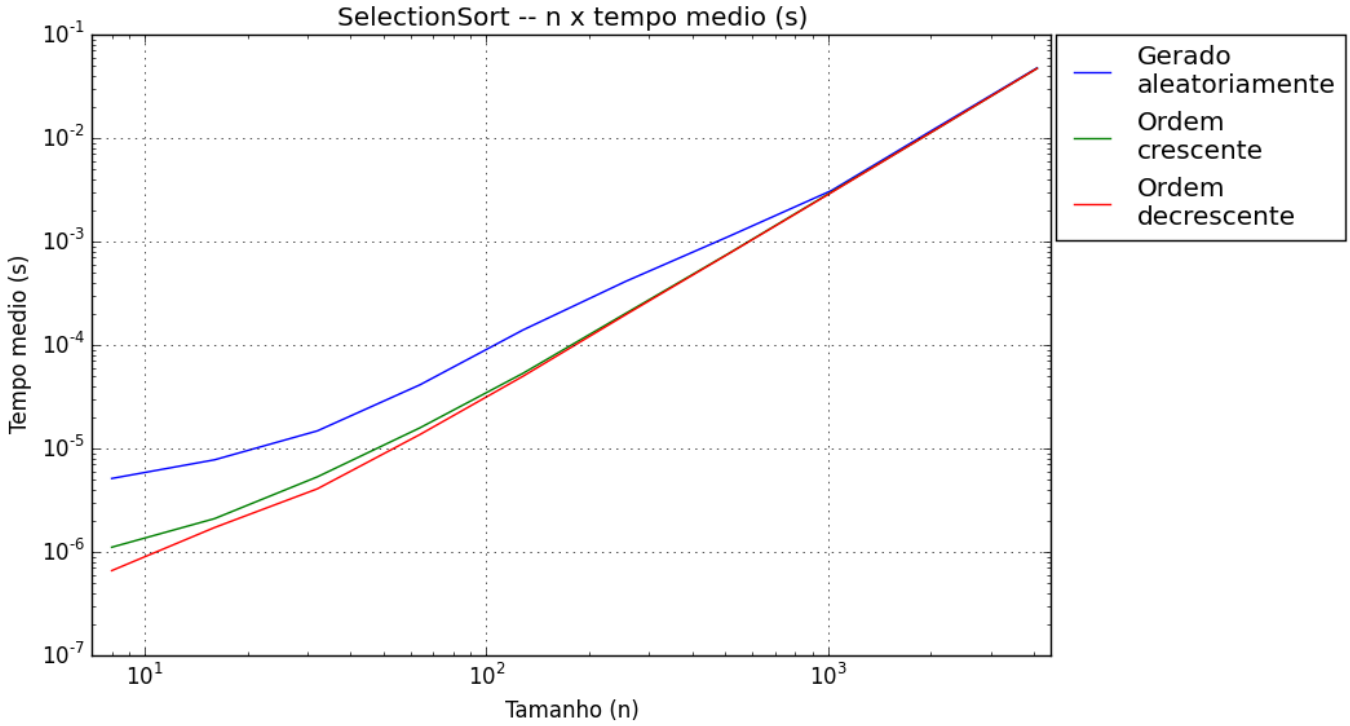
Nas próximas seções são apresentados os pontos chaves de cada método e os resultados obtidos nos experimentos deste trabalho com cada método. Nas penúltima seção, são apresentadas comparações entre os métodos e resultados obtidos. Na última seção, finalmente, são discutidas possibilidades de otimizações sobre os métodos. Para melhor visualização, todos os gráficos estão em escala $log \times log$.

# 1 *SelectionSort*

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita essas duas operações com $n-1$ itens restantes, depois com os $n-2$ itens, até que reste apenas um elemento.
- Custo linear no tamanho da entrada para o número de movimentos de registros, ou seja, é um método vantajoso quanto ao número de movimentos de registros, que é O(n).
- Deve ser usado quando os registros trabalhados são muito grandes, mas, com $n \leq 1.000$ elementos.
- O algoritmo não é estável pois não preserva a ordem de registros de chaves iguais.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 5.12802e-06 | 1.10988e-06 | 6.5962e-07 |
| $2^4 = 16$ | 7.76841e-06 | 2.09814e-06 | 1.72111e-06 |
| $2^5 = 32$ | 1.47915e-05 | 5.31706e-06 | 4.07311e-06 |
| $2^6 = 64$ | 4.13161e-05 | 1.58662e-05 | 1.36755e-05 |
| $2^7 = 128$ | 0.000139694 | 5.32759e-05 | 4.98731e-05 |
| $2^8 = 256$ | 0.000412375 | 0.000202498 | 0.00019642 |
| $2^9 = 512$ | 0.00111865 | 0.000767044 | 0.000760447 |
| $2^{10} = 1024$ | 0.00310944 | 0.00300972 | 0.00297457 |
| $2^{11} = 2048$ | 0.0122338 | 0.0118354 | 0.0118248 |
| $2^{12} = 4096$ | 0.0476476 | 0.0471404 | 0.0470924 |

Tabela 1: Resumo dos experimentos com *SelectionSort*.

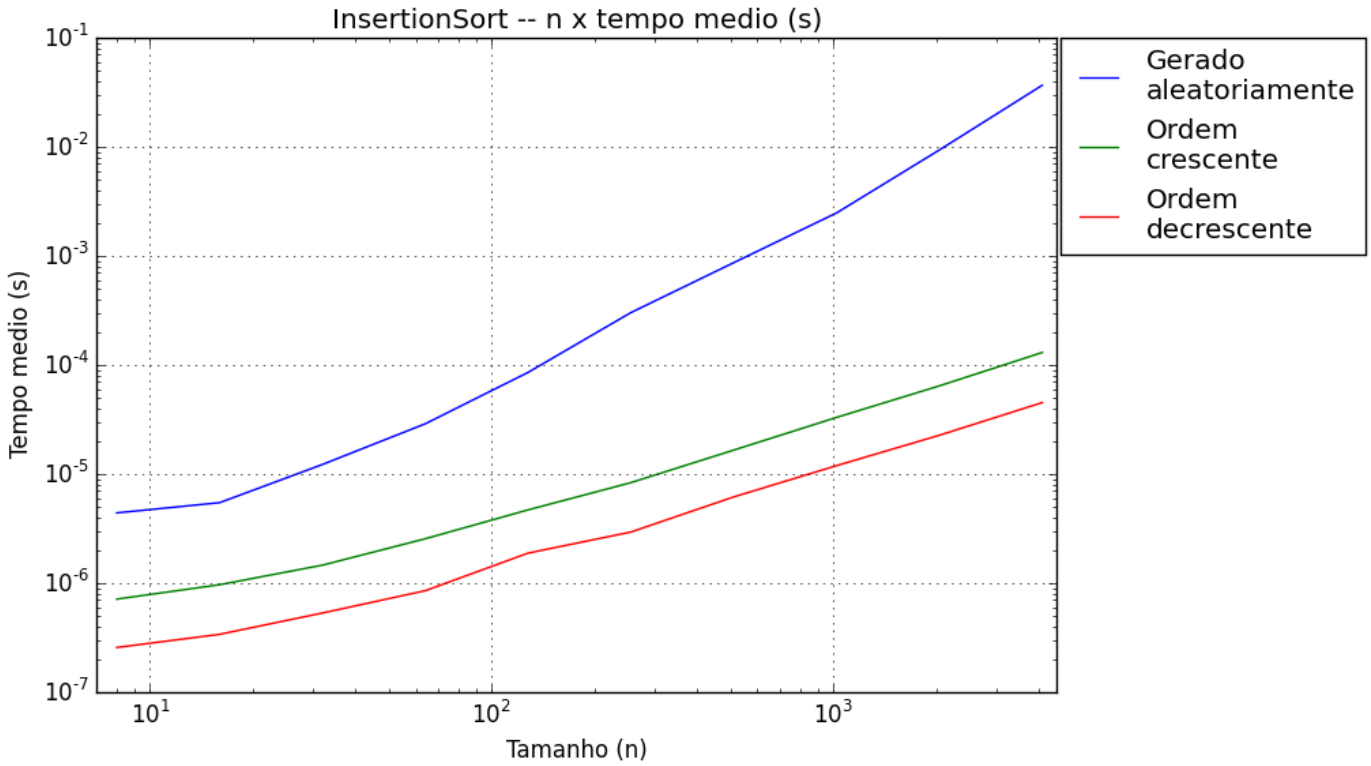

SelectionSort -- n x tempo medio (s)
Tempo medio (s)
Tamanho (n)
Gerado aleatoriamente
Ordem crescente
Ordem decrescente

# 2 *InsertionSort*

- Algoritmo:
  - Em cada passo a partir de $i = 2$ faça:
    - Selecione o i-ésimo item da sequência fonte.
    - Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.
- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o vetor está ordenado (ou "quase").
- É o mais interessante para $n \leq 20$.
- O método é estável pois preserva a ordem de registros de chaves iguais.
- Sua implementação é tão simples quanto o *SelectionSort*.
- Para vetores já ordenados, o método é O(n).
- O custo é linear para adicionar alguns elementos em um vetor já ordenado.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 4.40498e-06 | 7.1367e-07 | 2.5706e-07 |
| $2^4 = 16$ | 5.46593e-06 | 9.6681e-07 | 3.3913e-07 |
| $2^5 = 32$ | 1.224e-05 | 1.46122e-06 | 5.3212e-07 |
| $2^6 = 64$ | 2.89121e-05 | 2.55243e-06 | 8.5232e-07 |
| $2^7 = 128$ | 8.55862e-05 | 4.69559e-06 | 1.88507e-06 |
| $2^8 = 256$ | 0.00030248 | 8.34791e-06 | 2.94251e-06 |
| $2^9 = 512$ | 0.000867717 | 1.6589e-05 | 6.17797e-06 |
| $2^{10} = 1024$ | 0.002481 | 3.30821e-05 | 1.19496e-05 |
| $2^{11} = 2048$ | 0.00937856 | 6.44014e-05 | 2.27093e-05 |
| $2^{12} = 4096$ | 0.036745 | 0.000130339 | 4.52609e-05 |

Tabela 2: Resumo dos experimentos com *InsertionSort*.

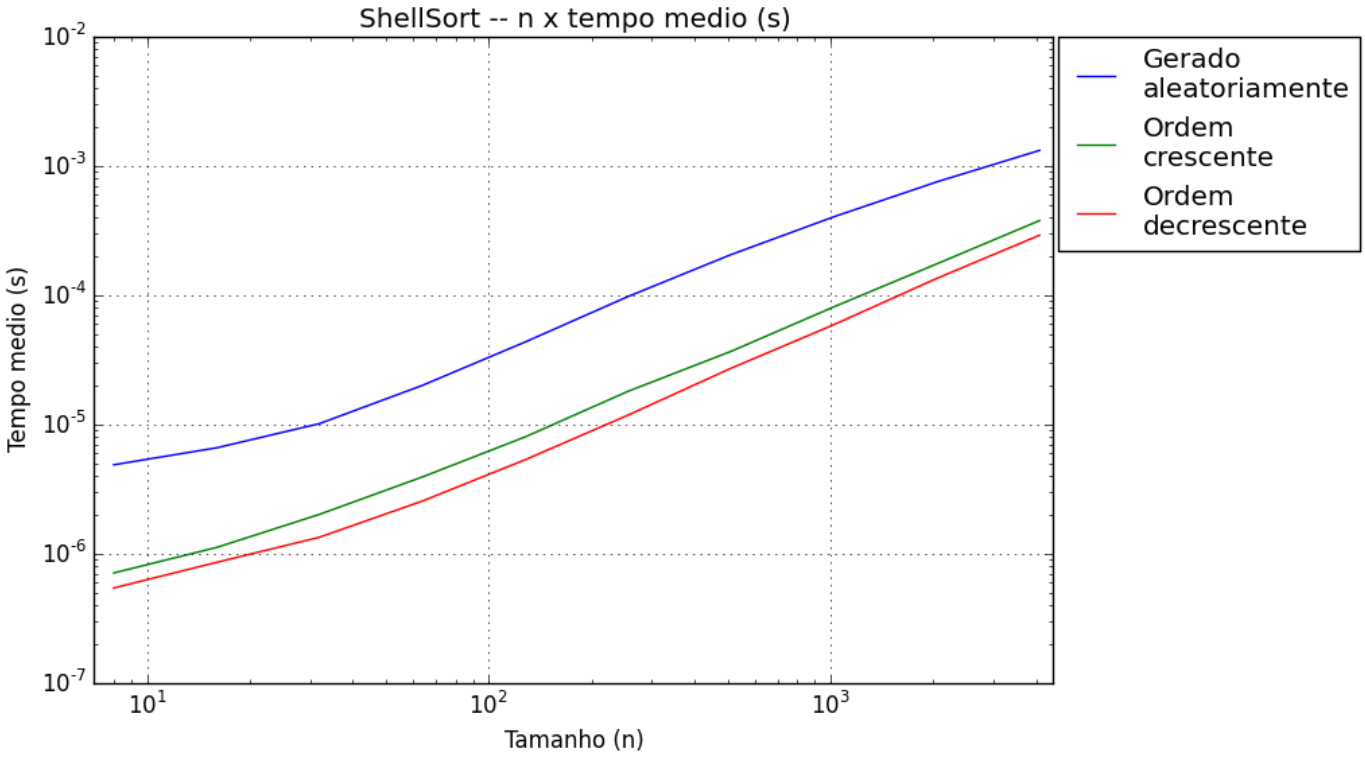

InsertionSort -- n x tempo medio (s)
Tempo medio (s)
Tamanho (n)
Gerado aleatoriamente
Ordem crescente
Ordem decrescente

# 3 *ShellSort*

- É uma extensão do algoritmo de ordenação por inserção proposto por Shell em 1959.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas $n - 1$ comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.
- Itens separados de $h$ posições são rearranjados.
- Todo $h$-ésimo item leva a uma sequência ordenada.
- Tal sequência é dita estar $h$-ordenada.
- Quando $h = 1$, *ShellSort* corresponde ao algoritmo de inserção.
- Knuth mostrou experimentalmente em 1973 que a seguinte sequência para $h$ é difícil de ser superada por mais de 20% de eficiência
  - $h(s) = 3h(s - 1) + 1$, para $s > 1$
  - $h(s) = 1$, para $s = 1$.
- É o método a ser escolhido para a maioria das aplicações por ser muito eficiente para $n$ moderado.
- Mesmo para $n$ grande, o método é cerca de apenas duas vezes mais lento do que o QuickSort.
- Sua implementação é simples e geralmente resulta em um programa pequeno.
- Não possui um pior caso ruim e, quando encontra um arquivo parcialmente ordenado, trabalha menos.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 4.89186e-06 | 7.1259e-07 | 5.4502e-07 |
| $2^4 = 16$ | 6.618e-06 | 1.12191e-06 | 8.6069e-07 |
| $2^5 = 32$ | 1.01806e-05 | 2.02228e-06 | 1.3456e-06 |
| $2^6 = 64$ | 2.01153e-05 | 3.93854e-06 | 2.5633e-06 |
| $2^7 = 128$ | 4.36065e-05 | 8.0404e-06 | 5.35389e-06 |
| $2^8 = 256$ | 9.79744e-05 | 1.81001e-05 | 1.18509e-05 |
| $2^9 = 512$ | 0.000206485 | 3.68765e-05 | 2.72756e-05 |
| $2^{10} = 1024$ | 0.000405379 | 8.14972e-05 | 5.93425e-05 |
| $2^{11} = 2048$ | 0.000754266 | 0.000175053 | 0.00013517 |
| $2^{12} = 4096$ | 0.00132223 | 0.000378851 | 0.000291863 |

Tabela 3: Resumo dos experimentos com *ShellSort*.

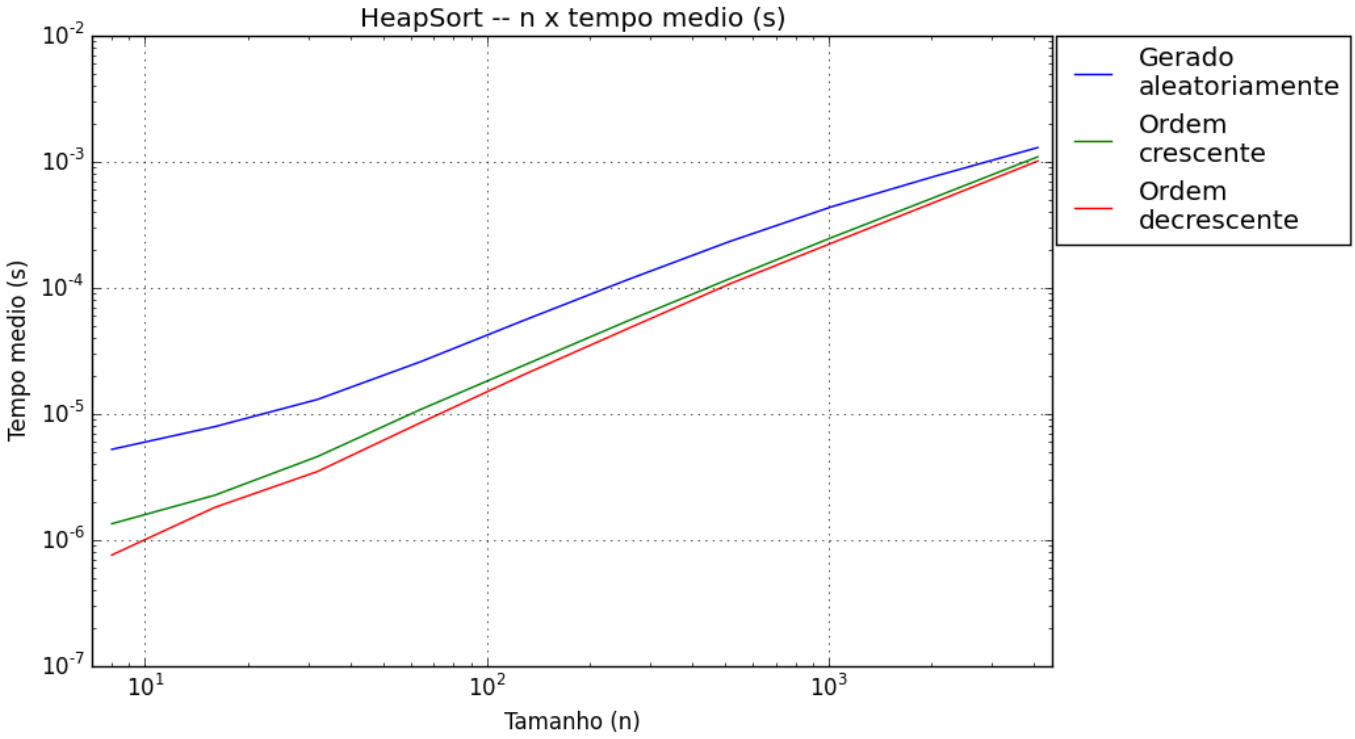

ShellSort -- n x tempo medio (s)
Tempo medio (s)
Tamanho (n)
Gerado aleatoriamente
Ordem crescente
Ordem decrescente

# 4 *HeapSort*

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Faz uso de Heaps para realizar a ordenação.
- Heaps
  - Representação extramente compacta.
  - Permite caminhar pelos nós da árvore facilmente.
  - Os filhos de um nó $i$ estão nas posições $2i$ e $2i + 1$.
  - O pai de um nó $i$ está na posição $i \div 2$.
  - Na representação do *Heap* em um arranjo, o maior valor está sempre na posição 1 do vetor.
  - Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.
  - Um algoritmo elegante para construção do *heap* foi proposto por Floyd em 1964.
- É um método de ordenação elegante e eficiente.
- Não necessita de nenhuma memória adicional.
- Executa sempre em tempo proporcional a $n\log n$.
- O anel interno do algoritmo é bastante complexo se comparado com o do *QuickSort*.
- Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o HeapSort.
- Não recomendado para $n$ pequeno por causa do tempo necessário para construção do *heap*.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 5.24562e-06 | 1.34737e-06 | 7.6196e-07 |
| $2^4 = 16$ | 7.93664e-06 | 2.26854e-06 | 1.81585e-06 |
| $2^5 = 32$ | 1.30587e-05 | 4.60247e-06 | 3.50246e-06 |
| $2^6 = 64$ | 2.5973e-05 | 1.08532e-05 | 8.52211e-06 |
| $2^7 = 128$ | 5.51224e-05 | 2.42456e-05 | 2.04582e-05 |
| $2^8 = 256$ | 0.000115131 | 5.41338e-05 | 4.6966e-05 |
| $2^9 = 512$ | 0.000232928 | 0.000117867 | 0.000107215 |
| $2^{10} = 1024$ | 0.000441655 | 0.000251636 | 0.000227411 |
| $2^{11} = 2048$ | 0.000769092 | 0.000524759 | 0.000479393 |
| $2^{12} = 4096$ | 0.00129926 | 0.00109709 | 0.00101582 |

Tabela 4: Resumo dos experimentos com *HeapSort*.

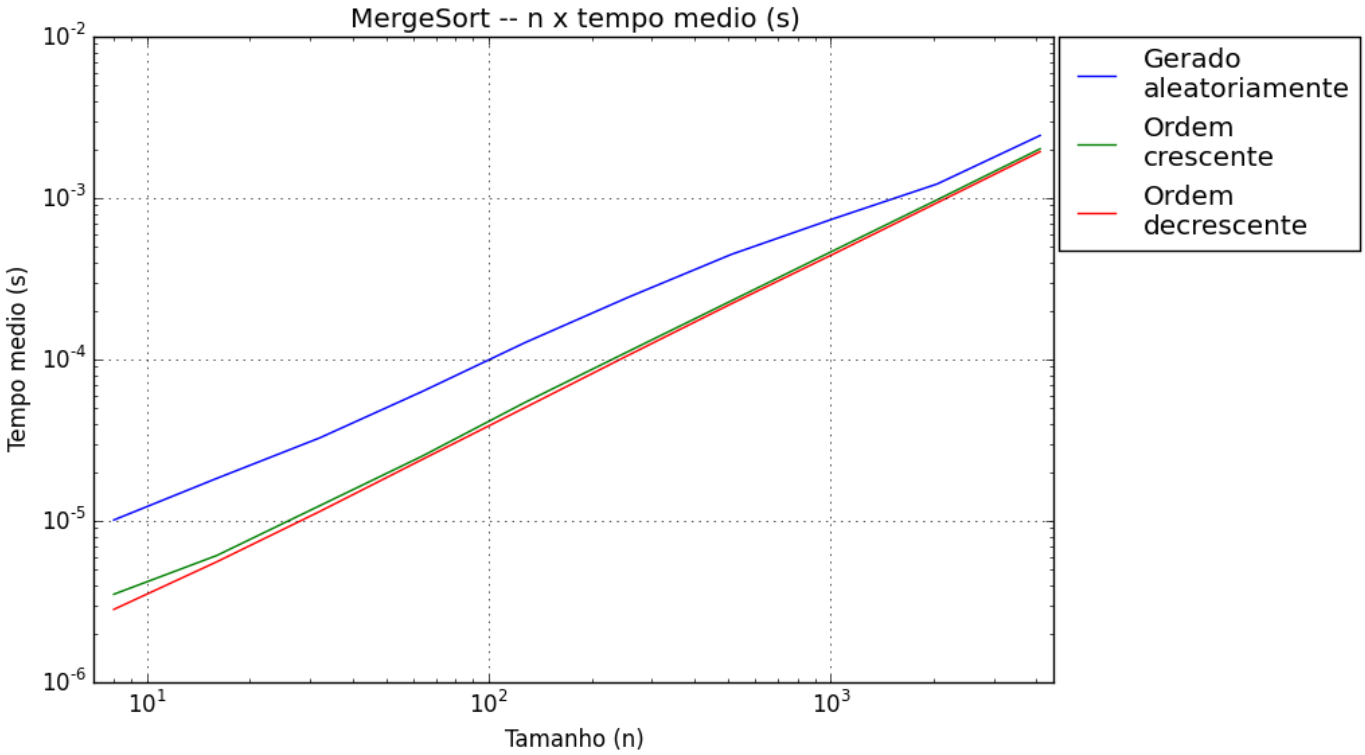

HeapSort -- n x tempo medio (s)
Tempo medio (s)
Tamanho (n)
Gerado aleatoriamente
Ordem crescente
Ordem decrescente

# 5 *MergeSort*

- Utiliza princípio de Divisão e Conquista:
  - Divida o vetor A em dois subconjuntos A1 e A2.
  - Solucione os sub-problemas associados a A1 e A2, isto é, ordene cada subconjunto separadamente (chamadas recursivas). Recursão pára quando atinge sub-problemas de tamanho 1.
  - Combine as soluções de A1 e A2 em uma solução para A: intercale os dois sub-vetores A1 e A2 e obtenha o vetor ordenado.
- Operação chave é a intercalação (*merge*).
- Algoritmo estável pois preserva a ordem de registros de chaves iguais.
- Demanda uso de vetor auxiliar para realização do merge, isto é, não é *in-place*.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 1.01302e-05 | 3.51091e-06 | 2.82803e-06 |
| $2^4 = 16$ | 1.83363e-05 | 6.10271e-06 | 5.57708e-06 |
| $2^5 = 32$ | 3.26664e-05 | 1.2412e-05 | 1.14336e-05 |
| $2^6 = 64$ | 6.35065e-05 | 2.52535e-05 | 2.40816e-05 |
| $2^7 = 128$ | 0.000127629 | 5.44212e-05 | 5.05182e-05 |
| $2^8 = 256$ | 0.000243191 | 0.000112234 | 0.000106321 |
| $2^9 = 512$ | 0.000447903 | 0.000231017 | 0.000221353 |
| $2^{10} = 1024$ | 0.000748083 | 0.00047531 | 0.000452735 |
| $2^{11} = 2048$ | 0.00122342 | 0.000975966 | 0.000935896 |
| $2^{12} = 4096$ | 0.00244268 | 0.00201968 | 0.00193757 |

Tabela 5: Resumo dos experimentos com *MergeSort*.


MergeSort -- n x tempo medio (s)
Tempo medio (s)
Tamanho (n)
Gerado aleatoriamente
Ordem crescente
Ordem decrescente

# 6 *QuickSort*

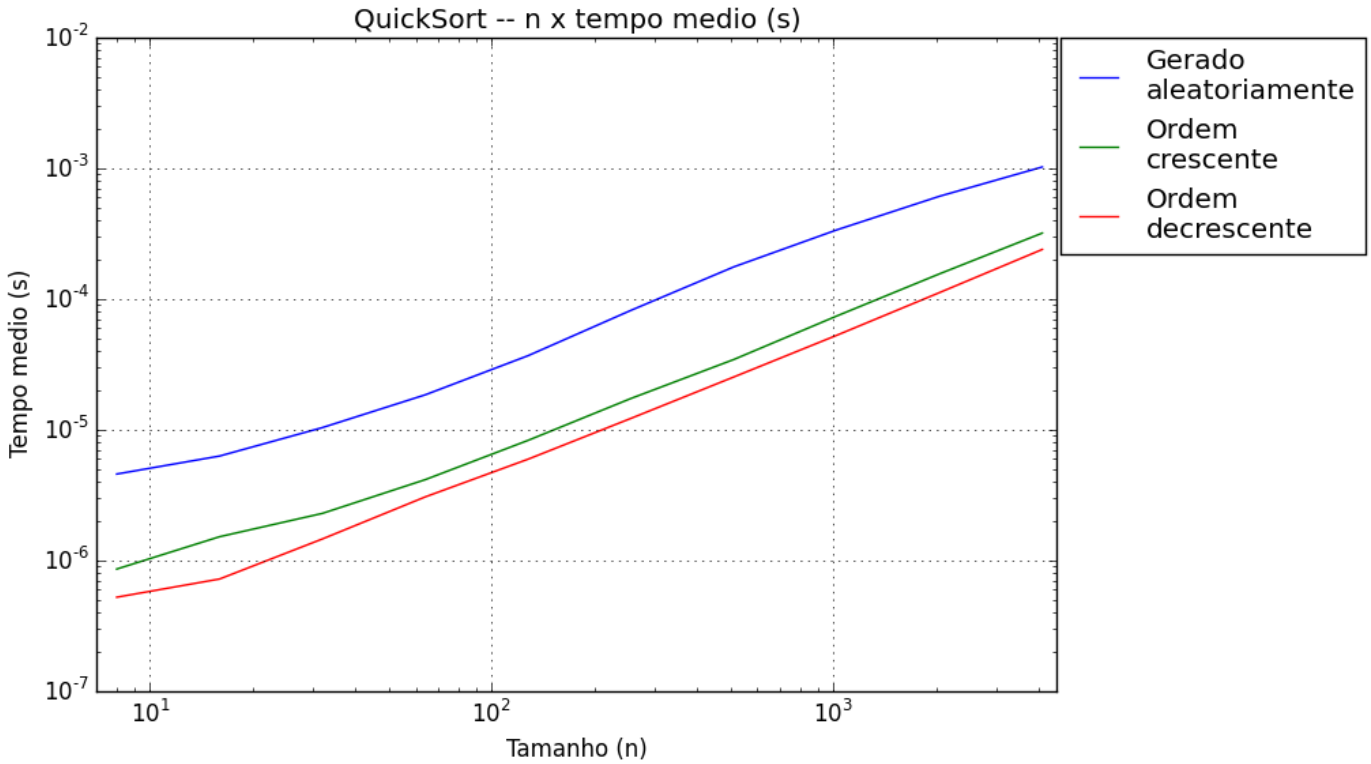
- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo mais eficiente que existe para uma grande variedade de situações.
- A idéia básica consiste de dividir o problema de ordenar um conjunto com $n$ itens em dois problemas menores.
  - Dividir: particiona o arranjo A[l..r] em dois sub-arranjos $A[l..p-1]$ e $A[p+1..r]$, de forma que os elementos de $A[l..p-1] \leq A[p]$ e $A[p] \leq A[p+1..r]$. (Forma de divisão apresentada por Bentley em 1984 diferente da original de Hoare.)
  - Conquistar: ordena de forma recursiva as duas sequências utilizando o próprio algoritmo *QuickSort*.
  - Combinar: como as duas subsequências já se encontram ordenadas não é preciso realizar mais trabalhos.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.
- A parte mais delicada do método é o processo de partição.
- É um método bastante frágil no sentido de que qualquer erro de implementação pode ser difícil de ser detectado.
- O algoritmo é recursivo, o que demanda uma pequena quantidade de memória adicional.
- O anel interno do procedimento de Partição é extremamente simples, razão pela qual o algoritmo *QuickSort* é tão rápido.
- No geral, é um algoritmo de ordenação muito eficiente.
- Requer cerca de $n\log n$ comparações em média para ordenar $n$ itens.
- Seu desempenho é da ordem de $O(n^2)$ operações no pior caso.
- O principal cuidado a ser tomado é com relação à escolha do pivô para realizar a partição.
- A escolha do elemento do meio do arranjo melhora muito o desempenho quando o arquivo está total ou parcialmente ordenado.
- O pior caso tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios.

# 7 `std::sort`

A biblioteca padrão C++ do GNU utiliza um algoritmo de ordenação híbrido para este algoritmo `std::sort`: começa com IntroSort (que é um híbrido de QuickSort e HeapSort), até uma profundidade máxima dada por $2\log n$, onde $n$ é o número de elementos, seguido de um InsertionSort no resultado.

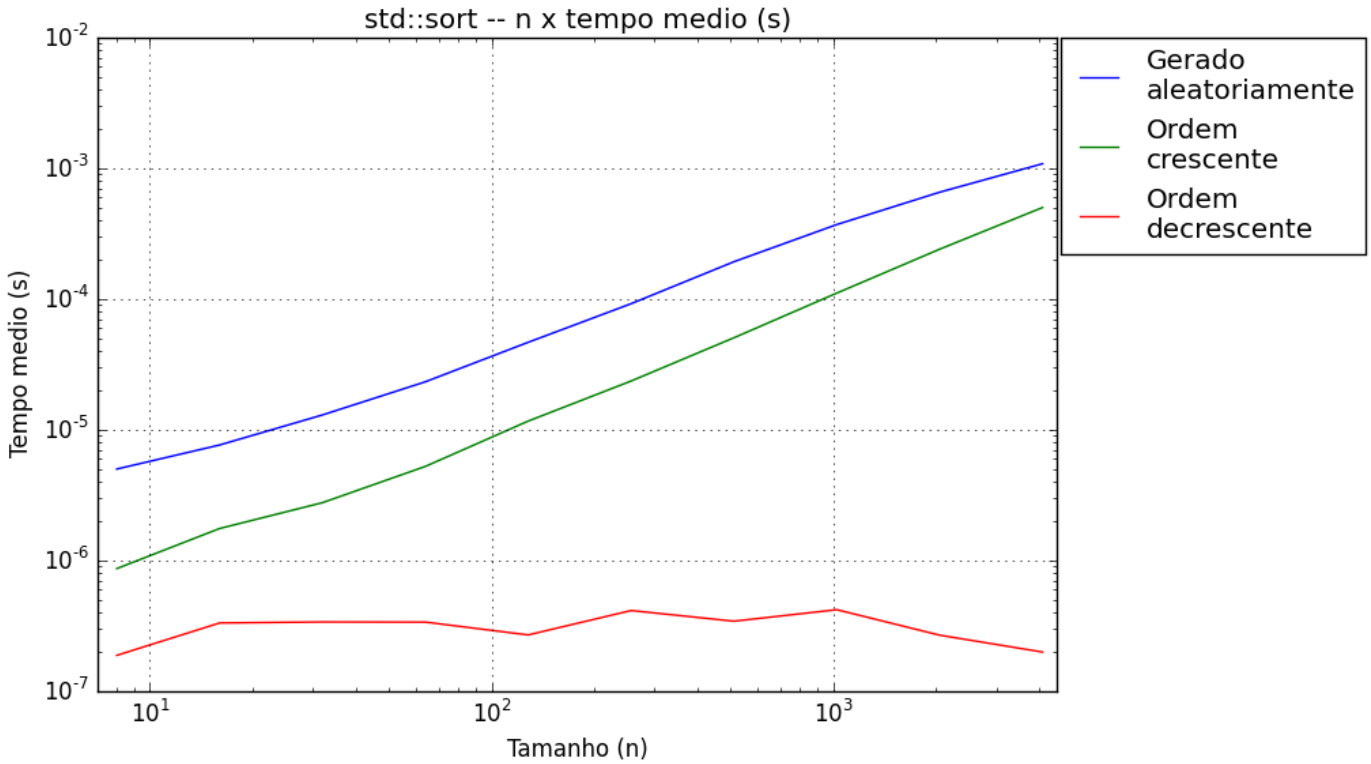
| n | tempo médio (s) Gerado aleatoriamente | tempo médio (s) Ordem crescente | tempo médio (s) Ordem decrescente |
|---|---|---|---|
| $2^3 = 8$ | 4.59146e-06 | 8.6123e-07 | 5.2447e-07 |
| $2^4 = 16$ | 6.30445e-06 | 1.52233e-06 | 7.2274e-07 |
| $2^5 = 32$ | 1.04094e-05 | 2.29656e-06 | 1.4638e-06 |
| $2^6 = 64$ | 1.85401e-05 | 4.16033e-06 | 3.07011e-06 |
| $2^7 = 128$ | 3.69372e-05 | 8.31275e-06 | 5.96729e-06 |
| $2^8 = 256$ | 8.20867e-05 | 1.73835e-05 | 1.22417e-05 |
| $2^9 = 512$ | 0.000176527 | 3.45331e-05 | 2.53833e-05 |
| $2^{10} = 1024$ | 0.000338384 | 7.42359e-05 | 5.28574e-05 |
| $2^{11} = 2048$ | 0.000612341 | 0.000156117 | 0.000112214 |
| $2^{12} = 4096$ | 0.00102614 | 0.000320261 | 0.000240065 |

Tabela 6: Resumo dos experimentos com *QuickSort*.



| n | tempo médio (s) Gerado aleatoriamente | tempo médio (s) Ordem crescente | tempo médio (s) Ordem decrescente |
|---|---|---|---|
| $2^3 = 8$ | 5.01365e-06 | 8.6813e-07 | 1.8829e-07 |
| $2^4 = 16$ | 7.66009e-06 | 1.75926e-06 | 3.3314e-07 |
| $2^5 = 32$ | 1.30051e-05 | 2.77361e-06 | 3.3924e-07 |
| $2^6 = 64$ | 2.33166e-05 | 5.25218e-06 | 3.3839e-07 |
| $2^7 = 128$ | 4.6762e-05 | 1.16712e-05 | 2.6994e-07 |
| $2^8 = 256$ | 9.22663e-05 | 2.36395e-05 | 4.1477e-07 |
| $2^9 = 512$ | 0.000193354 | 5.06586e-05 | 3.4375e-07 |
| $2^{10} = 1024$ | 0.00037208 | 0.000111231 | 4.2091e-07 |
| $2^{11} = 2048$ | 0.000657953 | 0.000241159 | 2.6873e-07 |
| $2^{12} = 4096$ | 0.00108608 | 0.000502364 | 1.9962e-07 |

Tabela 7: Resumo dos experimentos com `std::sort`.

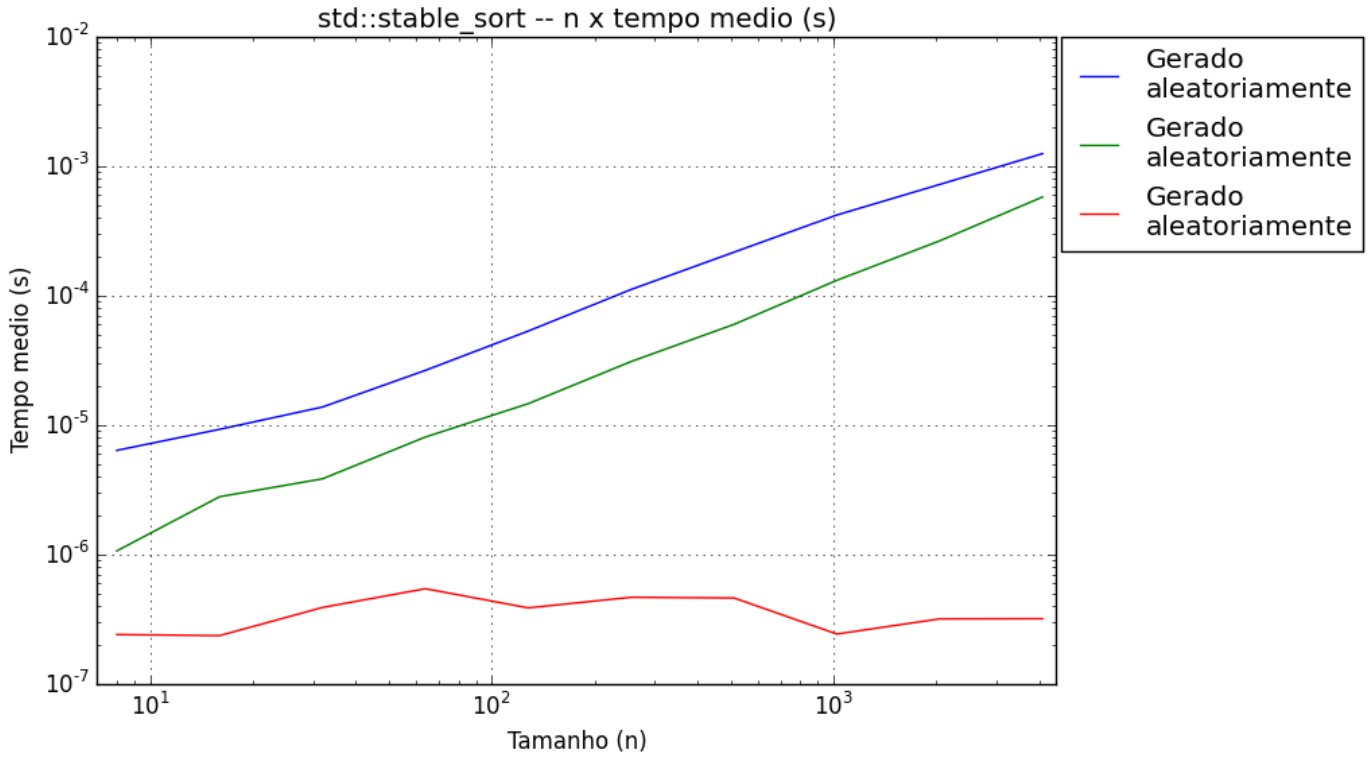


# 8 `std::stable_sort`

Em relação ao `std::sort`, `std::stable_sort` preserva a ordem original para elementos iguais, enquanto `std::sort` não preserva. Essa função tenta alocar um buffer temporário do mesmo tamanho da sequência (vetor) a ser ordenado. Se a alocação falhar, o algoritmo menos eficiente é escolhido.

| **n** | **tempo médio (s) Gerado aleatoriamente** | **tempo médio (s) Ordem crescente** | **tempo médio (s) Ordem decrescente** |
|---|---|---|---|
| $2^3 = 8$ | 6.38588e-06 | 1.06912e-06 | 2.4151e-07 |
| $2^4 = 16$ | 9.29756e-06 | 2.80297e-06 | 2.3683e-07 |
| $2^5 = 32$ | 1.38248e-05 | 3.85238e-06 | 3.9091e-07 |
| $2^6 = 64$ | 2.65229e-05 | 8.10406e-06 | 5.4566e-07 |
| $2^7 = 128$ | 5.33692e-05 | 1.46721e-05 | 3.8793e-07 |
| $2^8 = 256$ | 0.000111856 | 3.10421e-05 | 4.6839e-07 |
| $2^9 = 512$ | 0.000217153 | 6.00941e-05 | 4.63e-07 |
| $2^{10} = 1024$ | 0.000420481 | 0.000131718 | 2.4371e-07 |
| $2^{11} = 2048$ | 0.000724692 | 0.000265696 | 3.1905e-07 |
| $2^{12} = 4096$ | 0.00125373 | 0.000579898 | 3.2016e-07 |

Tabela 8: Resumo dos experimentos com `std::stable_sort`.

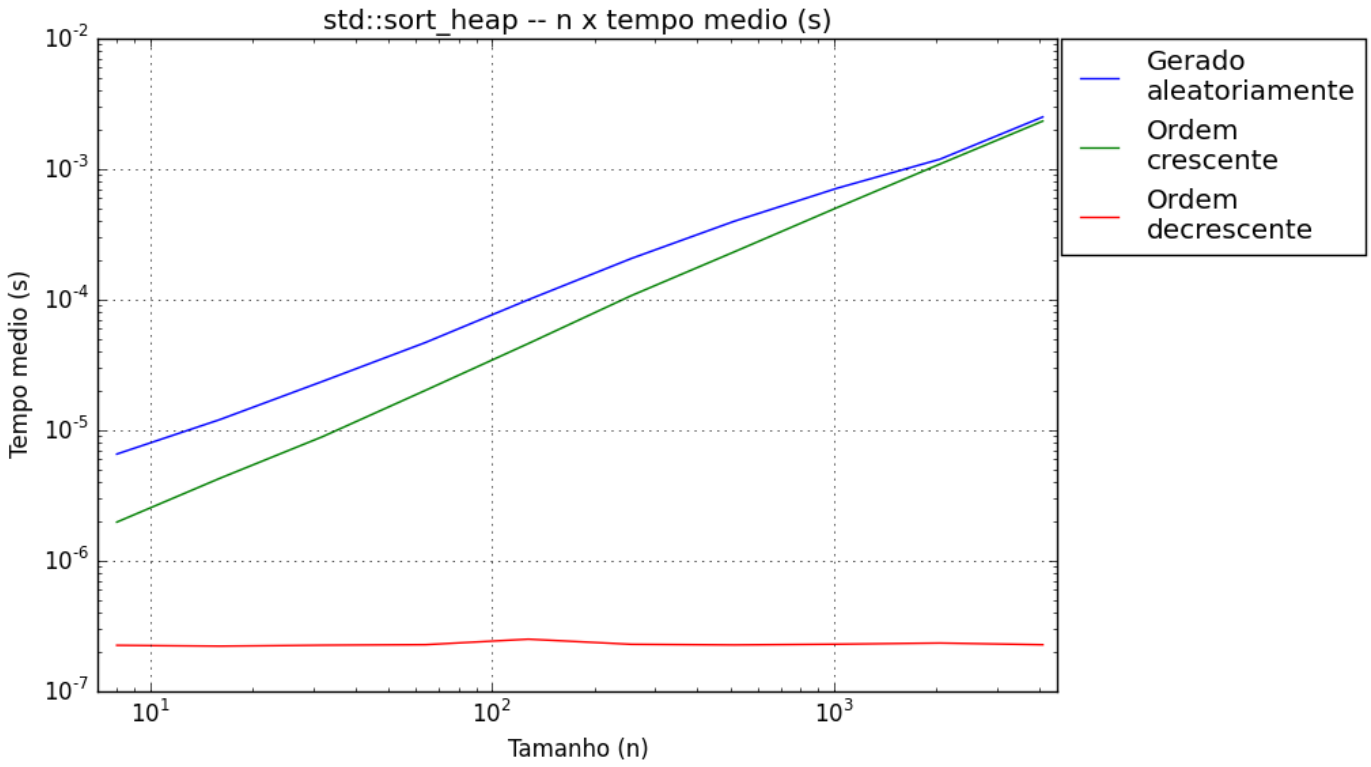


# 9 `std::sort_heap`

Aplica ordenação sobre uma estrutura heap. Necessita criação de heap como pré-requisito.

| n | tempo médio (s) Gerado aleatoriamente | tempo médio (s) Ordem crescente | tempo médio (s) Ordem decrescente |
|---|---|---|---|
| $2^3 = 8$ | 6.57068e-06 | 1.9793e-06 | 2.2547e-07 |
| $2^4 = 16$ | 1.20609e-05 | 4.28126e-06 | 2.2214e-07 |
| $2^5 = 32$ | 2.36837e-05 | 8.93542e-06 | 2.2584e-07 |
| $2^6 = 64$ | 4.70189e-05 | 2.02221e-05 | 2.2794e-07 |
| $2^7 = 128$ | 0.000100015 | 4.61806e-05 | 2.5116e-07 |
| $2^8 = 256$ | 0.000206777 | 0.000107412 | 2.2946e-07 |
| $2^9 = 512$ | 0.000398741 | 0.000232412 | 2.2691e-07 |
| $2^{10} = 1024$ | 0.000714148 | 0.000507439 | 2.2995e-07 |
| $2^{11} = 2048$ | 0.00119157 | 0.00109401 | 2.3461e-07 |
| $2^{12} = 4096$ | 0.00251225 | 0.00233573 | 2.2772e-07 |

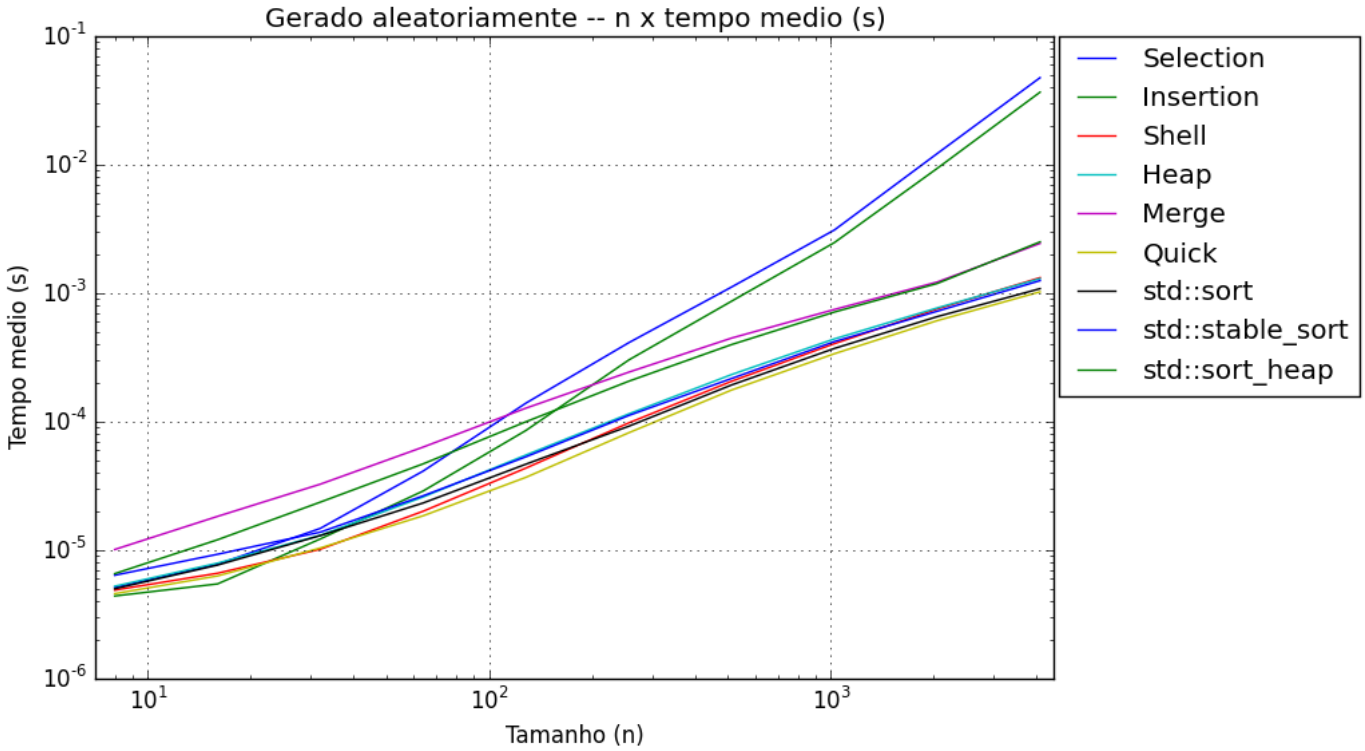
Tabela 9: Resumo dos experimentos com `std::sort_heap`.



# 10 Comparação de todos os métodos

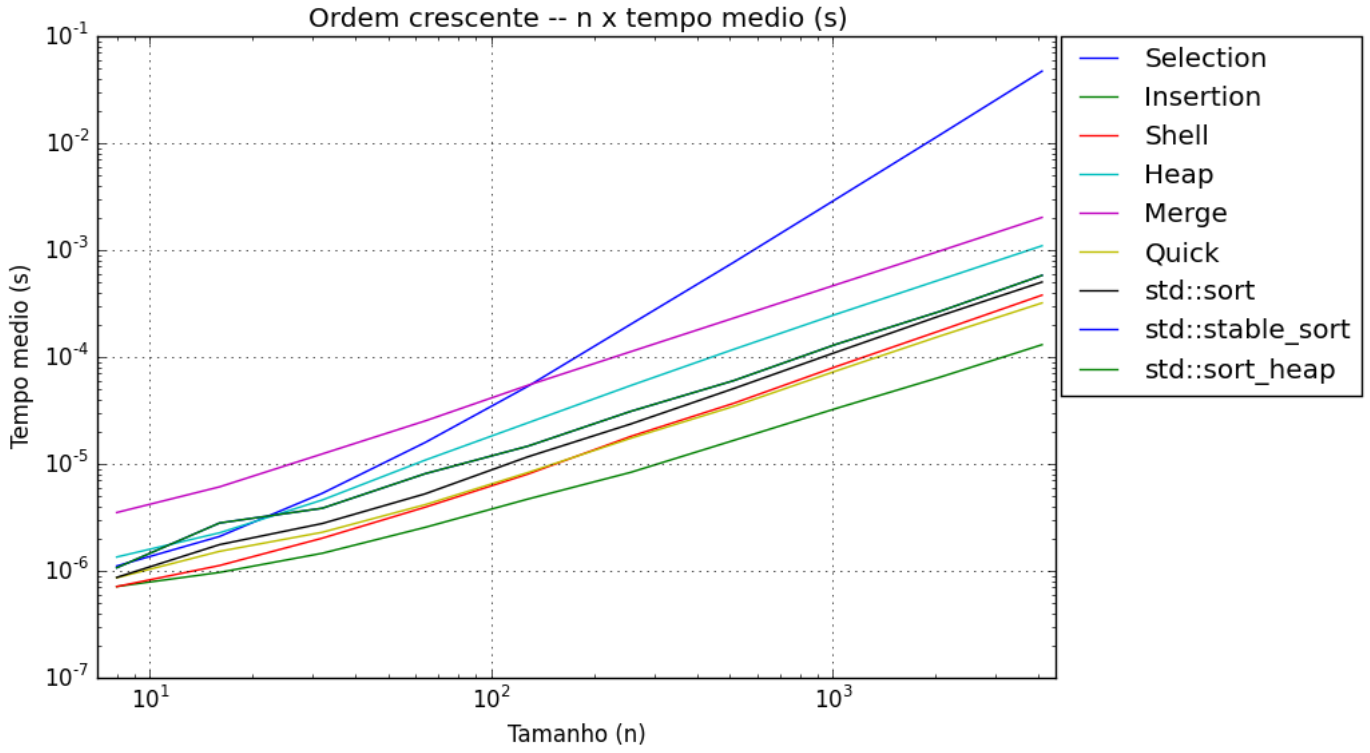
| Método | Complexidade |
|---|---|
| *SelectionSort* | $O(n^2)$ |
| *InsertionSort* | $O(n^2)$ |
| *ShellSort* | $O(n\log n)$ |
| *HeapSort* | $O(n\log n)$ |
| *MergeSort* | $O(n\log n)$ |
| *QuickSort* | $O(n\log n)$ |
| `std::sort` | $O(n\log n)$ |
| `std::stable_sort` | $O(n\log n)$ |
| `std::sort_heap` | $O(n\log n)$ |

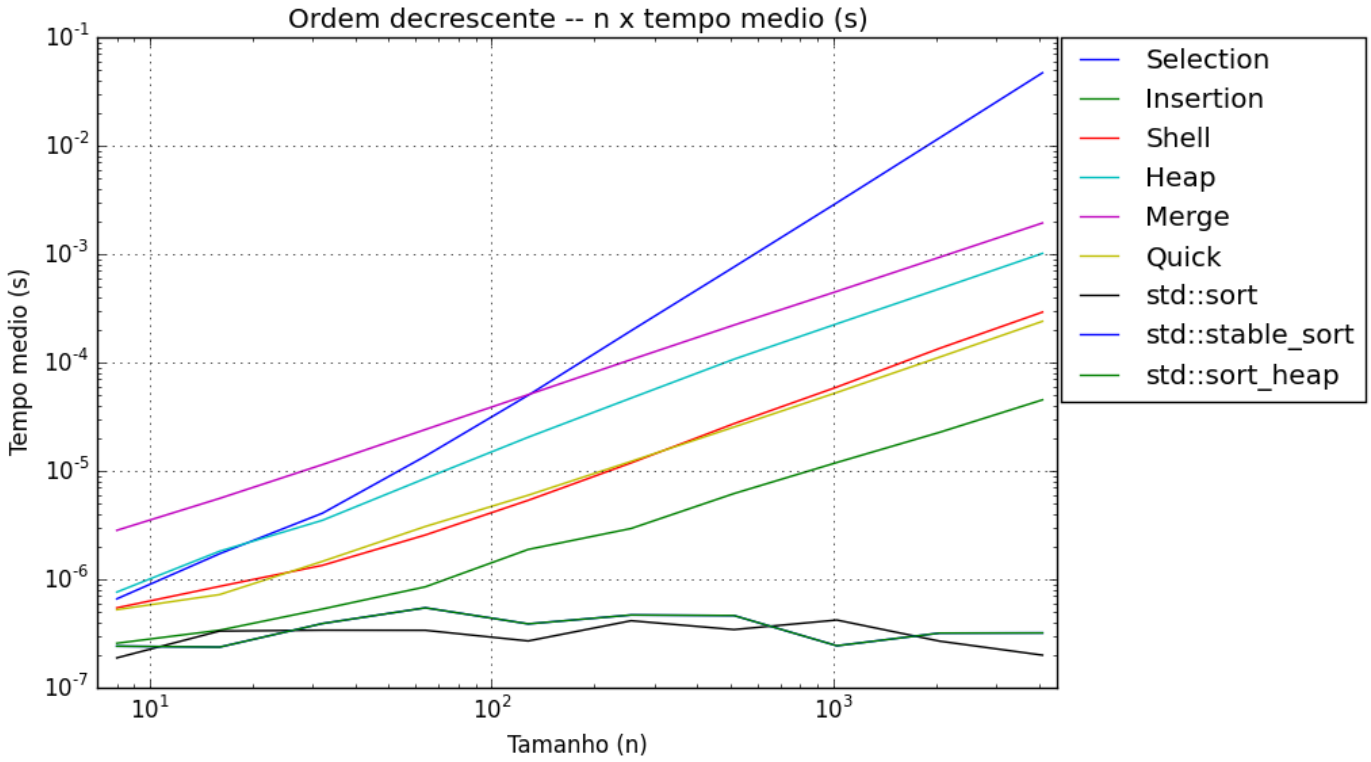
Tabela 10: Resumo das complexidades dos métodos de ordenação utilizados nos experimentos.




Observações sobre os métodos:

1. Apesar de não se conhecer analiticamente o comportamento do *ShellSort*, ele é considerado um método eficiente.
2. *ShellSort*, *QuickSort* e *HeapSort* têm a mesma ordem de grandeza.
3. *QuickSort* é o mais rápido para todos os tamanhos aleatórios experimentados.
4. A relação *HeapSort*/*QuickSort* se mantém constante para todos os tamanhos.
5. A relação *ShellSort*/*QuickSort* aumenta à medida que o número de elementos aumenta.




6. MergeSort é mais demorado quando comparado aos métodos de mesma complexidade.
7. Para $n$ pequeno (até 500), *ShellSort* é mais rápido que o *HeapSort*.
8. Quando o tamanho da entrada cresce, o *HeapSort* é mais rápido que o *ShellSort*.

9. *InsertionSort* é o mais rápido para qualquer tamanho se os elemento estão ordenados.

10. Entre os algoritmos de custo O($n^2$), *InsertionSort* é o melhor para todos os tamanhos experimentados.

11. Para $n$ iguais, *QuickSort* executa mais rápido para vetores ordenados.

12. *QuickSort* é o mais rápido para vetores ordenados de forma crescente.

13. `std::sort` pode ser mais lento que `std::make_heap` e `std::sort_heap` em $N/(3+\log N)$ vezes nos pior caso. Mas, na média, `std::sort` é mais rápido.

14. `std::sort` pode ser mais lento que `std::make_heap` e `std::sort_heap` em $N/(3+\log N)$ vezes nos pior caso. Mas, na média, `std::sort` é mais rápido.

15. Para vetores ordenados de forma decrescente, `std::sort` e `sort_heap` são os mais eficientes.

# 11 Otimizações

Otimizações para os métodos *SelectionSort*, *InsertionSort* e *ShellSort* não foram encontradas.

Os métodos que utilizam recursividade – *HeapSort*, *MergeSort* e *QuickSort* poderiam ser otimizados ao serem paralelizados, isto é, poderíamos utilizar um processo diferente para cada chamada recursiva. Para vetores grandes ($n$ grande), o ganho pode ser significativo.

O *QuickSort* ainda tem outras possíveis otimizações.

- O pior caso pode ser evitado empregando pequenas modificações no algoritmo. Para isso, basta escolher três itens quaiquer do vetor e usar a mediana dos três como pivô. Mais genericamente: mediana de $k$ elementos.

- Interromper as partições para vetores pequenos, utilizando um dos algoritmo básicos para ordená-los (*SelectionSort* ou *InsertionSort*).

- Remover a recursão: *QuickSort* não-recursivo. Uma possível implementação envolve uso da estrutura de dados pilha.

- Planejar ordem em que subvetores são processados a fim de se obter maior eficiência.