

Análise de desempenho do algoritmo *bucketsort* para diferentes subalgoritmos de ordenação

Eugenio Souza Carvalho¹, Hugo Santos Piauilino Neto¹

¹Departamento de Computação
Universidade Federal do Piauí (UFPI)
Teresina – PI – Brazil

{eugeniucarvalho, hugos94}@gmail.com

Abstract. *This work presents a performance analysis of bucketsort algorithm for different sorting subalgorithms, in addition to a general overview of the history and operation of the algorithms.*

Resumo. *Este trabalho apresenta uma análise de desempenho do algoritmo de ordenação bucketsort para diferentes subalgoritmos de ordenação, além de apresentar um resumo geral sobre a história e funcionamento dos algoritmos.*

1. Introdução

Problemas são questões propostas em busca de uma solução. Algoritmos são utilizados com o propósito de conceder uma solução para certo problema. Para todo problema decível existe um algoritmo que determina uma solução para as instâncias desse problema.

Algoritmos descrevem passo a passo os procedimentos para chegar a uma solução de um problema e podem ser representados de três formas: descrição narrativa, fluxograma e a linguagem algorítmica. Neste trabalho focaremos na utilização da última forma.

Algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem. Em outras palavras efetua sua ordenação completa ou parcial de acordo com uma necessidade pré-estabelecida. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista.

Os mais populares algoritmos de ordenação são: *insertionsort*, *selectionsort*, *bubblesort*, *bucketsort*, *quicksort*, *mergesort*, *heapsort* e *shellsort*. Neste artigo, o algoritmo *bucketsort* será analisado, explicando seu funcionamento, suas peculiaridades e seu comportamento quando os subalgoritmos *insertionsort*, *mergesort*, *heapsort* e *quicksort* são utilizados.

2. Bucketsort

O *bucketsort* ou algoritmo de ordenação por baldes, funciona em tempo linear quando a entrada é gerada a partir de uma distribuição uniforme. O *bucketsort* é rápido porque pressupõe que a entrada é gerada por um processo aleatório que distribui elementos uniformemente sobre o intervalo $[0, 1)$ [Cormen et al. 2009].

O algoritmo *bucketsort* originou-se do método *countingsort* (ordenação por contagem), pois o *bucketsort* torna-se *countingsort* quando os baldes possuem tamanho 1. O *countingsort* funciona separando cada número do *array* original e colocando em sua devida posição ao final do processo. O *bucketsort* realiza o mesmo procedimento, exceto por manipular múltiplos números ao mesmo tempo.

2.1. Estratégia Utilizada

O *bucketsort* adota a estratégia de dividir o intervalo $[0, 1)$ em n subintervalos (baldes) de igual tamanho e depois distribuir os n números de entrada entre os baldes. Tendo em vista que as entradas são uniformemente distribuídas sobre $[0, 1)$, não esperamos que grandes quantidade de números caiam em cada balde. Para produzir a saída, simplesmente ordenamos os números em cada balde, e depois percorremos os baldes em ordem, listando os elementos contidos em cada um [Cormen et al. 2009].

O algoritmo *bucketsort* funciona da seguinte maneira:

1. Inicialize um vetor de "baldes", inicialmente vazios;
2. Percorra o vetor original, incluindo cada elemento em um balde;
3. Ordene todos os baldes não vazios;
4. Coloque os elementos dos baldes que não estão vazios no vetor original.

O passo 3 do funcionamento do algoritmo *bucketsort* pode ser realizado de duas maneiras. A primeira é chamar recursivamente o algoritmo *bucketsort* para realizar a ordenação dos baldes. A segunda maneira é utilizar qualquer outro algoritmo de ordenação para ordenar os baldes não vazios. Este artigo analisa o desempenho obtido com a utilização de subalgoritmos de ordenação para ordenar os baldes internamente.

Na análise realizada foram utilizados 4 algoritmos de ordenação diferentes. Os algoritmos escolhidos foram: *insertionsort*, *mergesort*, *heapsort* e *quicksort*.

2.2. Pseudo-Código

O Algoritmo 1 demonstra o pseudo-código para o algoritmo *bucketsort*. Podemos verificar que a função *bucketsort* recebe como parâmetro apenas o *array* a ser ordenado.

Este pseudo-código pressupõe que a entrada é um arranjo de n elementos A , e que cada elemento $A[i]$ no arranjo satisfaz a $0 \leq A[i] \leq 1$. O código exige um *array* auxiliar $B[0..n - 1]$ de listas ligadas (baldes) e pressupõe que existe um mecanismo para manter tais listas.

Function *bucketsort* ($A[]$)

```
n = comprimento[A];  
for  $i = 0; i < n; i = i + 1$  do  
    | inserir  $A[i]$  na lista  $B[\lfloor A[i] \rfloor]$ ;  
end  
for  $i = 0; i < n - 1; i = i + 1$  do  
    | ordenar lista  $B[i]$  com qualquer algoritmo de ordenação (inclusive o  
    | próprio bucketsort);  
end  
concatenar as listas  $B[0], B[1], \dots, B[n - 1]$  juntas em ordem
```

Result: O algoritmo retorna o vetor ordenado.

Algorithm 1: Pseudo-código do algoritmo *bucketsort*.

O primeiro passo do Algoritmo 1 é armazenar o tamanho do *array*. Logo após, um **for** percorre todo o *array* A e insere os elementos visitados no *array* auxiliar de listas ligadas B .

No próximo passo, um outro **for** percorre todo o *array* auxiliar de listas ligadas B e aplica um algoritmo de ordenação em cada lista. Qualquer algoritmo de ordenação

pode ser utilizado nessa etapa, inclusive o próprio algoritmo *bucketsort* recursivamente. Para esta análise de desempenho, foram utilizados os algoritmos descritos na Seção 3.

Por último, o *array* auxiliar de listas ligadas *B* é concatenado em ordem e gera como resultado os mesmos elementos do *array* *A* ordenados.

A Figura 1 demonstra um exemplo de funcionamento do algoritmo *bucketsort* para entradas uniformemente distribuídas entre $[0, 1)$.

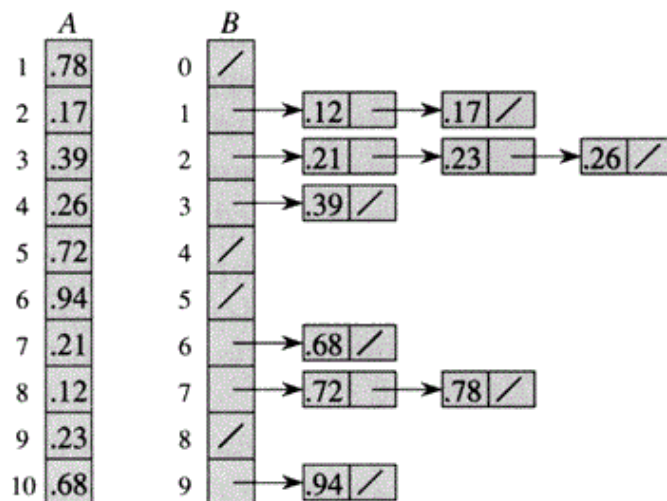


Figura 1. Exemplo do funcionamento do algoritmo *bucketsort* utilizando dois *arrays* para entradas uniformemente distribuídas.

2.3. Complexidade

A complexidade do algoritmo *bucketsort* depende do subalgoritmo utilizado em seu interior para realizar a ordenação das listas ligadas (baldes).

Segundo Cormen [Cormen et al. 2009], quando o subalgoritmo utilizado for o próprio *bucketsort*, o tempo esperado a partir de uma distribuição uniforme é $\Theta(n) + n * O(2 - 1/n) = \Theta(n)$, dessa forma, funcionando em tempo linear.

Mesmo que a entrada não seja obtida a partir de uma distribuição uniforme, o *bucketsort* ainda pode ser executado em tempo linear, pois a entrada tem a propriedade de que a soma dos quadrados dos tamanhos de baldes é linear no número total de elementos.

Porém, quando outros algoritmos de ordenação forem utilizados para realizar a ordenação das listas ligadas (baldes), a complexidade do *bucketsort* dependerá da complexidade do subalgoritmo utilizado.

3. Subalgoritmos

Esta seção apresenta uma descrição geral dos subalgoritmos utilizados para realizar a análise de desempenho do algoritmo *bucketsort*.

3.1. Insertion Sort

O *insertionsort*, ou ordenação por inserção, é um simples algoritmo de ordenação, bastante eficiente quando aplicado para pequenas quantidades de elementos. Em termos

gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados [Knuth 1998a]. O *insertion sort* funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho, como o pôquer.

Possui o menor número de trocas e comparações entre os algoritmos de ordenação $\mathcal{O}(n)$ quando o vetor está ordenado e em seu pior caso possui complexidade de tempo $\mathcal{O}(n^2)$.

3.2. Merge Sort

O *mergesort*, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Foi inventado em 1945 por John von Neumann [Knuth 1998b].

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

Possui complexidade de tempo $\Theta(n \log_2 n)$ e complexidade de espaço $\Theta(n)$ para todos os casos.

3.3. Heap Sort

O algoritmo *heapsort* é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. Foi inventado em 1964 por J.W.J Williams [Williams 1964].

O *heapsort* utiliza uma estrutura de dados chamada *heap*, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da *heap*, na ordem desejada, lembrando-se sempre de manter a propriedade de *max-heap*.

A *heap* pode ser representada como uma árvore binária com propriedades especiais ou como um vetor [Baase 1988]. Para uma ordenação decrescente, deve ser construída uma *heap* mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma *heap* máxima (o maior elemento fica na raiz).

Possui complexidade de tempo $\Theta(n \log_2 n)$ e complexidade de espaço $\Theta(n)$ para todos os casos.

3.4. Quick Sort

O *quicksort* é um método de ordenação muito rápido, eficiente. É um algoritmo não estável e adota a estratégia de divisão e conquista. Foi inventado em 1961 por C.A.R. Hoare [Hoare 1961].

A estratégia consiste em rearranjar as chaves de modo que chaves menores precedam chaves maiores. Em seguida o *quicksort* ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Possui complexidade de tempo $\Theta(n \log_2 n)$ e complexidade de espaço $\Theta(\log_2 n)$ para o melhor caso e o caso médio. Para o pior caso, possui complexidade de tempo e espaço $\Theta(n^2)$.

4. Materiais

Nesta seção serão apresentados os materiais utilizados para realizar as simulações propostas, tanto em termos de *software*, quanto de *hardware*.

4.1. Software

O algoritmo *bucketsort* foi implementado utilizando a linguagem de programação C. Para a compilação, foi utilizado o compilador gcc (TDM-2 mingw32) versão 4.4.1 2009 [Mingw 2009].

O ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*) utilizado foi o Code::Blocks versão 13.12 [Code:Blocks 2016].

O sistema operacional utilizado para realizar as simulações foi o *Windows* 10 de 64 bits versão *Professional* [Microsoft 2015].

4.2. Hardware

A máquina utilizada para realizar as simulações possui processador AMD Phenom(tm) II X4 B97 Processor 3.20 GHz com três pentes de memória RAM de 4 GB DDR3 2000Mhz, totalizando 12 GB de memória RAM.

5. Resultados

Para comparar os métodos, foram escolhidos dez diferentes tamanhos para o *array*: 100, 500, 1.000, 5.000, 30.000, 80.000, 100.000, 150.000 e 200.000 elementos.

Para cada tamanho especificado foram gerados *arrays* de números aleatórios, permitindo valores repetidos. Foram realizadas 20 simulações para cada tamanho em cada método. A média dos tempos de execuções foram utilizadas para realizar a análise comparativa.

Podemos observar na Figura 2 que o subalgoritmo *insertionsort* apresentou os piores tempos médios de execução em relação aos outros algoritmos. Para o tamanho máximo analisado, com 200000 elementos, o algoritmo chegou a utilizar 60 segundos em sua execução, em média.

Como os tempos médios do algoritmo *insertionsort* foram bastante elevados, não foi possível identificar os tempos de execução médios dos outros subalgoritmos utilizando a Figura 2. Para fins de comparação entre os métodos, todos os algoritmos, com exceção do *insertionsort*, foram comparados.

Podemos observar na Figura 3 que os tempos médios dos algoritmos *mergesort*, *heapsort* e *quicksort* não ultrapassaram a faixa de 0,09 segundos. O *quicksort* foi o subalgoritmo com melhores tempos médios de execução, não ultrapassando a barreira de 0,04 segundos para todos os tamanhos de *arrays* analisados.

Os algoritmos *heapsort* e *mergesort* ficaram com os segundos e terceiros melhores tempos médios de execução para ordenar os baldes no interior do algoritmo *bucketsort*, não ultrapassando a barreira de 0,06 e 0,08 segundos, respectivamente.

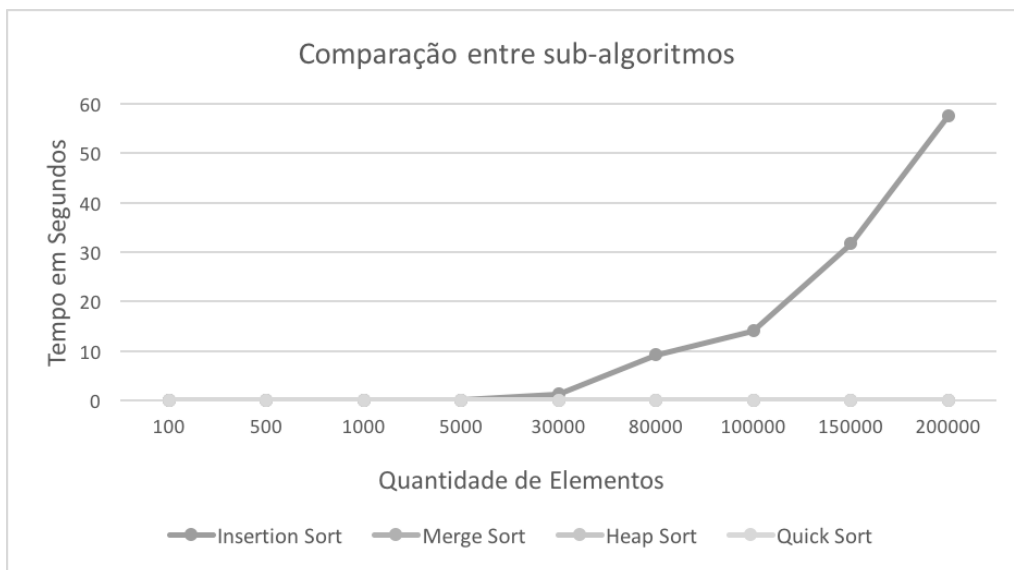


Figura 2. Gráfico comparativo entre os tempos médios de execução dos sub-algoritmos de ordenação *Insertion*, *Merge*, *Heap* e *Quick*.

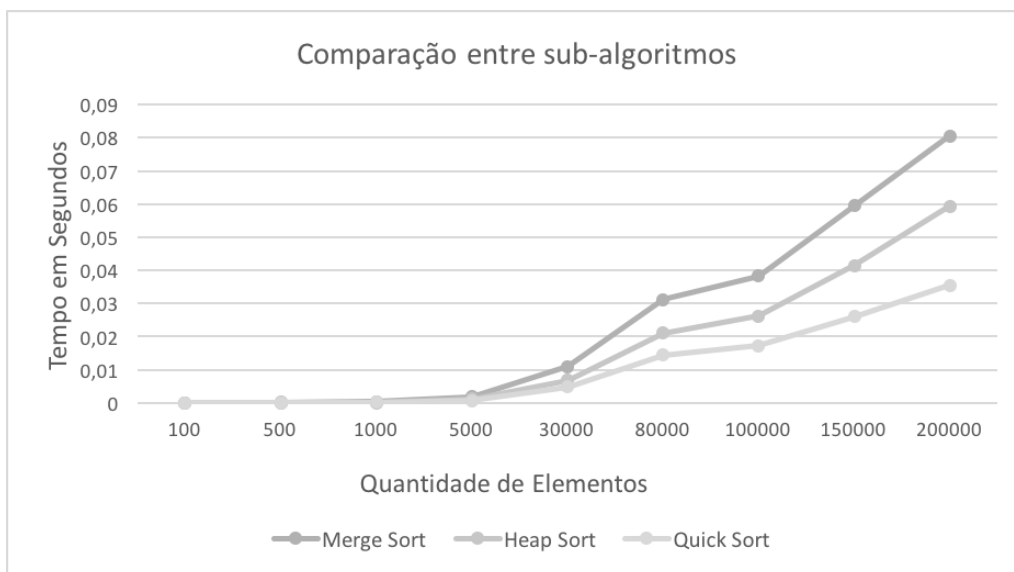


Figura 3. Gráfico comparativo entre os tempos médios de execução dos sub-algoritmos de ordenação *Merge*, *Heap* e *Quick*.

Podemos verificar na Figura 3 que todos os algoritmos parecem obedecer um padrão médio de tempos de execução.

6. Conclusão

Podemos concluir que a escolha do subalgoritmo utilizado no interior do algoritmo de ordenação *bucket sort* gera grande impacto nos tempos finais médios de execução. Portanto, quanto melhor for o desempenho do subalgoritmo de ordenação escolhido para ordenar os baldes internamente, melhor será o desempenho geral do *bucket sort*.

Referências

- Baase, S. (1988). *Computer algorithms : introduction to design and analysis*. Addison-Wesley series in computer science. Addison-Wesley, Reading Mass.
- Code:Blocks (2016). Code::blocks. <https://www.codeblocks.org/>. Acessado em: 11-06-2016.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.
- Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–.
- Knuth, D. (1998a). Section 5.2.1: Sorting by insertion. *Sorting and Searching. The Art of Computer Programming 3 (2nd ed.)*.
- Knuth, D. (1998b). Section 5.2.4: Sorting by merging. *Sorting and Searching. The Art of Computer Programming 3 (2nd ed.)*.
- Microsoft (2015). Windows 10. <https://www.microsoft.com/pt-br/windows/>. Acessado em: 11-06-2016.
- Mingw (2009). Mingw. <https://www.mingw.org/>. Acessado em: 11-06-2016.
- Williams, J. W. J. (1964). Algorithm 232 - heapsort. *Communications of the ACM*, 7(6):347–349.