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

Eugenio Souza Carvalho[1], **Hugo Santos Piauilino Neto**[1]

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

{hugos94,eugeniucarvalho}@gmail.com

Abstract. *This paper presents a performance analysis of bucketsort algorithm with different sorting sub-algorithms, in addition to a general overview of the history and operation of the algorithm.*

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

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 decidível 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*, *combsort*, *quicksort*, *mergesort*, *heapsort* e *shellsort*. Neste artigo, o algoritmo *quicksort* será analisado, explicando o seu funcionamento, suas peculiaridades e o comportamento do seu particionamento.

2. *Bucketsort*

3. Estratégia Utilizada

O *quicksort* adota a estratégia de divisão e conquista. Essa estratégia consiste em rearranjar as chaves do problema de modo que chaves ”menores”precedam chaves ”maiores”. Em seguida o *quicksort* ordena as duas sub-listas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada [**?**].

O algoritmo *quicksort* executa os seguintes passos:

1. Escolha um elemento da lista, denominado pivô;

2. Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverão duas sub-listas não-ordenadas. Essa operação é denominada particionamento;
3. Recursivamente ordene a sub-lista dos elementos menores e a sub-lista dos elementos maiores.

A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

3.1. Pseudo-Código

O Algoritmo 1 demonstra o pseudo-código para o algoritmo *quicksort*. Podemos verificar que a função *quicksort* recebe como parâmetros de entrada um *array* e suas posições inicial e final. Logo, o método de particionamento escolhido é chamado e como resultado retorna um elemento pivô. Este pivô é utilizado para realizar as chamadas recursivas das sub-listas à esquerda e direita do elemento pivô. Quando as listas se tornarem de tamanho 1, o algoritmo retorna o *array* devidamente ordenado.

O método *Partition* do Algoritmo 1 dependerá do particionamento escolhido para executar o algoritmo.

```
Function quicksort (A[], primeiro, ultimo)
    if primeiro < ultimo then
        pivo = Partition(A, primeiro, ultimo);
        quicksort(A, primeiro, pivo-1);
        quicksort(S, pivo+1, ultimo);
    end
Result: O algoritmo retorna o vetor ordenado.
```

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

3.2. Dimensão de Desempenho

Em uma base teórica, podemos determinar o número de comparações de elementos e trocas para comparar o desempenho. Além disso, o tempo de funcionamento real será influenciado por outros fatores, como desempenho de *caches* e escalonamento de *threads*.

Como será mostrado abaixo, os métodos possuem comportamento semelhante em permutações aleatórias, exceto pelo número de trocas. Aqui, o método de *Lomuto* necessita de três vezes mais trocas do que o particionamento de *Hoare*.

3.3. Número de Comparações

Ambos os métodos podem ser implementados utilizando $n - 1$ comparações para particionar um *array* de comprimento n. Isto é essencialmente ideal, uma vez que precisamos comparar cada elemento com o pivô para decidir onde colocá-lo.

3.4. Número de Trocas

O número de trocas é aleatório para ambos os algoritmos, dependendo dos elementos no *array*. Se assumirmos permutações aleatórias, ou seja, todos os elementos são distintos e cada permutação dos elementos é igualmente provável, podemos analisar o número esperado de trocas.

Como apenas a ordem relativa conta, assumimos que os elementos são os números $1, ..., n$. Isso faz com que a discussão abaixo se torne mais fácil pois a posição de um elemento e seu valor coincidem.

3.5. Método de *Lomuto*

A variável índice j escaneia o *array* completo e sempre que encontra um elemento $A[j]$ menor que o pivô x, a troca é realizada. Entre os elementos $1, ..., n$, exatamente $x - 1$ são menores que x, então nós teremos $x - 1$ trocas se o pivô for x.

A expectativa geral então resulta do cálculo da média de todos os pivôs. Segundo [**?**], cada valor em $\{1, ..., n\}$ tem a mesma probabilidade de $\frac{1}{n}$ de se tornar pivô, então serão realizadas

$$\frac{1}{n}\sum_{x=1}^{n}(x-1) = \frac{n}{2} - \frac{1}{2} \tag{1}$$

trocas, em média, para particionar um *array* de comprimento n com o método de *Lomuto*.

3.6. Método de *Hoare*

Para este método, a análise é um pouco mais complexa. Mesmo fixando o pivô x, o número de trocas permanece aleatório.

Os índices i e j correm um em direção ao outro até que eles se cruzem, que sempre acontece em x (por correção do algoritmo de particionamento de *Hoare*). Isto divide eficazmente o *array* em duas partes: a parte esquerda que é verificada pela variável índice i e uma parte direita que é verificada pela variável índice j.

Agora, uma troca é feita para cada par de elementos ”fora do lugar”, isto é, um elemento grande (maior do que x, que pertence a partição direita) que atualmente está localizado na partição esquerda e um elemento pequeno que esteja localizado na partição direita. Note-se que este par formado trabalha sempre para fora, ou seja, o número de pequenos elementos inicialmente na partição direita é igual ao número de grandes elementos na partição esquerda.

[**?**] mostra que o número destes pares é hiper geometricamente distribuído $Hyp(n-1, n-x, x-1)$: para os $n - x$ maiores elementos nós aleatoriamente traçamos suas posições no *array* e temos $x - 1$ posições na partição esquerda . Por conseguinte, o número esperado de pares é $(n - x)(x - 1)/(n - 1)$ dado que o pivô é x.

Segundo [**?**], a média de todos os valores dos pivôs é calcula para obter o número total esperado de trocas para o método de particionamento de *Hoare*:

$$\frac{1}{n}\sum_{x=1}^{n}\frac{(n-x)(x-1)}{n-1} = \frac{n}{6} - \frac{1}{3}. \tag{2}$$

Mais informações podem ser encontradas em [**?**, Pág. 29].

3.7. Padrão de Acesso a Memória

Ambos os métodos usam dois ponteiros que escaneiam o *array* sequencialmente. Portanto, ambos possuem comportamento quase ideal.

3.8. Elementos Iguais e Listas Ordenadas

A performance dos algoritmos diferem mais drasticamente para listas que não estão aleatoriamente permutadas.

Em um *array* já ordenado, o método de *Hoare* não realiza nenhuma troca, já que não existem pares mal posicionados, ao passo que o método de *Lomuto* realiza cerca de $\frac{n}{2}$ trocas.

A presença de elementos iguais requere cuidados especiais na utilização do algoritmo *quicksort*. Considere um exemplo extremo onde um *array* é preenchido apenas com elementos 0. Para este *array*, o método de *Hoare* realiza um troca para cada par de elementos - configurando o pior caso para o particionamento de *Hoare* - mas i e j sempre encontram-se no meio do *array*. Assim, temos um particionamento ideal e o tempo total de execução permanece em $\mathcal{O}(n \log n)$.

O método de *Lomuto* possui comportamento pior para o *array* apenas com elementos 0: a comparação $A[j] <= x$ sempre irá retornar verdadeira, então serão realizadas trocas para todos os elementos. Entretanto piora: após o *loop*, sempre teremos $i = n$, então observamos o pior caso de particionamento, fazendo com que a performance do método seja degradada para $\Theta(n^2)$.

4. Materiais

4.1. *Software*

O algoritmo *buicksort*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.

6. Conclusão

Podemos concluir que a escolha do método de particionamento tem impacto no resultado final, tal escolha deve levar em conta o tipo de entrada que será submetida ao algoritmo. Para entradas suficientemente grandes o método de particionamento de *Hoare* comporta-se melhor que o método de *Lomuto* obtendo menor tempo de execução.

Referências

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.

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.