Comparação experimental entre Algoritmos de Ordenação

Rafael Barros Pereira

Resumo

O objetivo desse trabalho é estudar o problema de ordenação, implementar alguns dos algoritmos tradicionais para esse problema, e analisar o seu desempenho computacional. Os seguintes algoritmos são analisados:

- BubbleSort
- SelectionSort
- InsertionSort
- ShellSort
- MergeSort
- HeapSort
- QuickSort
- Método das Caixas

1. Introdução

A ordenação é o processo de dispor um determinado número de elementos em uma seqüência em alguma ordem pré-estabelecida. A ordenação é utilizada em muitas aplicações e até mesmo outros algoritmos (como buscas), e por isso é um dos aspectos mais importantes em programação [Knuth 98].

Neste trabalho serão avaliados oito algoritmos tradicionais de ordenação, de funcionamento e complexidade variados. São eles: ordenação pelo método da bolha (*BubbleSort*), ordenação por seleção (*SelectionSort*), ordenação por inserção (*InsertionSort*), *ShellSort*, *HeapSort*, *MergeSort*, *QuickSort* e a ordenação pelo método das caixas.

Resultados experimentais são executados com o objetivo de comparar o seu desempenho e confirmar a complexidade teórica dos algoritmos.

2.1 BubbleSort

O método da bolha ou *Bubblesort* é um dos algoritmos de ordenação mais simples. Ele percorre o vetor inteiro comparando elementos adjacentes (dois a dois), efetuando trocas sucessivas se eles estiverem fora de ordem. O pseudocódigo da Figura 2.1 ilustra o seu funcionamento.

A complexidade de pior caso do método da bolha é a seguinte: o primeiro laço “Para i de 0 até n” é executado *n* vezes, onde *n* é o tamanho do vetor. Em seguida o laço interno “Para j de 1 até n-1” é executado (n-1) vezes. A operação “Se” e as operações de troca possuem complexidade O(1). O “Se” que verifica se houve trocas pertence ao laço externo, e

também possui complexidade O(1). A complexidade de pior caso do algoritmo é de O(n) x O(n) x O(1) = $O(n^2)$.

O pior caso ocorre quando o vetor de entrada está ordenado de forma inversa (decrescente). Nesse caso o primeiro elemento sofrerá (n-1) trocas, o segundo elemento (n-2), e assim por diante. Haverá trocas em todas as iterações do laço externo, portanto a otimização que retorna o vetor ordenado, caso não haja trocas, não será executada.

```
bubbleSort(vetor[] a)
        para (i de 1 até a.tamanho)
                trocou = falso;
                para (j de 1 até a.tamanho - 1)
                        se (a[j] > a[j + 1])
                                temp = a[j];
                                a[j] = a[j + 1];
                                a[j + 1] = temp;
                                trocou = verdadeiro;
                se (!trocou)
                        retorna; // vetor já ordenado
        retorna;
```

Figura 2.1: Pseudocódigo do método da bolha (*Bubblesort*)

2.2 SelectionSort

A ordenação por seleção ou *SelectionSort* é um método que seleciona a cada passo o menor valor da partição ainda não ordenada do vetor, trocando esse elemento para a partição ordenada. O pseudocódigo da Figura 2.2 ilustra o funcionamento da ordenação por seleção.

Na primeira iteração o algoritmo busca o menor elemento do vetor, e troca com o elemento da primeira posição (caso este não seja o menor). Na segunda iteração o segundo menor elemento do vetor será trocado com o elemento da segunda posição (também caso já não esteja na posição correta). A cada iteração seguinte do algoritmo o próximo menor valor será buscado, e trocado com o próximo elemento da partição "esquerda" do vetor, que é onde estão armazenados os elementos já ordenados.

A complexidade de pior caso da ordenação por seleção é a seguinte: o laço externo executa *n* vezes, onde *n* é o tamanho do vetor. A cada iteração desse é atribuído o mínimo como o elemento atual do laço (complexidade O(1)). O próximo passo é varrer todos os próximos elementos do vetor buscando o menor valor. Na primeira iteração esse laço interno é executado (n-1) vezes. Na iteração seguinte são (n-2) vezes. E, portanto, a complexidade do laço interno é da ordem de O(n) (progressão aritmética). Após encontrar o menor valor, é efetuada uma troca para cada iteração do laço externo. As trocas possuem complexidade de O(1). A complexidade total do algoritmo é de O(n) x O(n) x O(1) = $O(n^2)$.

```
selectionSort (vetor[] a)
        para (i de 1 até a.tamanho)
                min = i;
                para (j de i + 1 até a.tamanho)
                        se (a[j] < a[min])
                                min = j;

                temp = a[min];
                a[min] = a[i];
                a[i] = temp;
        retorna;
```

Figura 2.2: Pseudocódigo do método de seleção (*SelectionSort*)

2.3 InsertionSort

A ordenação por inserção ou *InsertionSort* é um método simples de ordenação, que funciona de maneira como muitas pessoas ordenam as cartas em um jogo de baralho [Cormen 04]. Em cada passo um elemento é colocado em sua posição adequada, invertendo a sua posição com o elemento anterior, caso necessário.

O algoritmo varre o vetor colocando cada elemento em sua posição correta. O pseudocódigo da Figura 2.3 mostra o seu funcionamento.

A complexidade do algoritmo é a seguinte: o laço externo executa (n-1) vezes, onde *n* é o tamanho do vetor. As atribuições seguintes possuem complexidade O(1). O laço seguinte (do enquanto) pode executar no pior caso *n* operações, assumindo por exemplo um vetor ordenado de maneira decrescente, em que na última iteração são necessárias (n-1) inversões para que o elemento seja corretamente posicionado na primeira posição. Esse laço coloca o elemento da posição *j* na posição *j-1* e decrementa o valor de *j* (operações O(1)). Essa operação é repetida até que ele esteja corretamente posicionado, então o valor é armazenado e a próxima iteração do laço externo é executada. A complexidade do algoritmo é portanto O(n-1) x O(n) = $O(n^2)$.

Uma característica interessante da ordenação por inserção é que, se o vetor já estiver ordenado, são executadas apenas as operações do laço externo, requerendo apenas o tempo O(n) para retornar o vetor ordenado. Ele também é útil em casos no qual o vetor está quase ordenado, pois são requeridas poucas inversões (execuções do laço interno).

```
insertionSort (vetor[] a)
        para (i de 2 até a.tamanho)
                j = i;
                valor = a[i];
                enquanto (j > 0 E a[j - 1] > valor)
                        a[j] = a[j - 1];
```

```
                    j--;

                a[j] = valor;
            retorna;
```

Figura 2.3: Pseudocódigo do método de inserção (*InsertionSort*)

2.4 Shellsort

O método de ordenação Shell Sort foi proposto por Donald Shell, e é considerado um melhoramento da seleção por inserção [Knuth 98]. A idéia é que ao invés de fazer os elementos desordenados andarem de uma em uma posição até alcançar o seu local correto, a cada passo o elemento anda um número maior de posições até o destino. Como exemplo a figura 2.4.1 (de [Knuth 98]) exibe uma ordenação, onde na primeira passada os elementos são deslocados de oito em oito posições até alcançar o lugar adequado, na próxima passada ocorrem deslocamentos de quatro posições, depois de duas posições, até a última passada em que os elementos são deslocados em uma posição (como na seleção por inserção).

Esse procedimento de deslocar os elementos um número variável de vezes, de acordo com uma função decidida previamente, faz com que o número de inversões dos elementos tenda a ser menor do que na seleção por inserção.

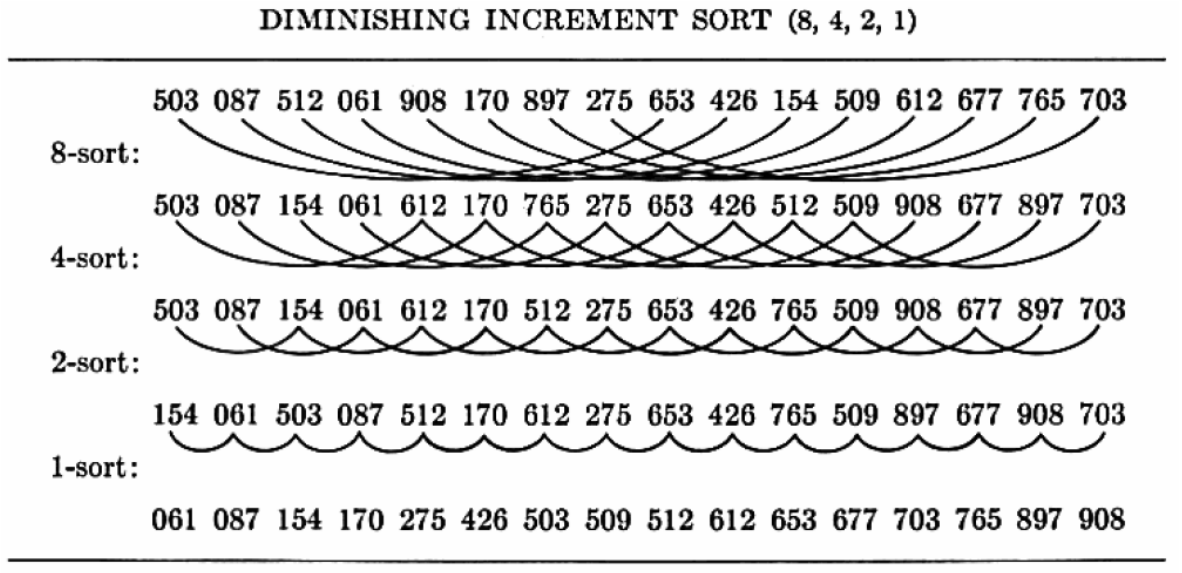

Figura 2.4.1: Exemplo do funcionamento do *ShellSort* com função $h = 2^s$

O algoritmo do *shellSort* está ilustrado no pseudocódigo da Figura 2.4.2. A complexidade do algoritmo não é tão fácil de calcular, pois depende da função escolhida para determinar o salto (ou *gap*). Em [Knuth 98] é discutida uma função que diminui a complexidade de pior caso de $O(n^2)$ da seleção por inserção para $O(n^{1,667})$, e também a demonstração que para a função $h = 2^s - 1$, onde *s* representa a passada, a complexidade de pior caso é $O(n^{3/2})$.

A implementação do pseudocódigo usa a função h = 3 * h + 1, que retorna a seqüência de valores: 1, 4, 13, 40, 121, 364, etc. Em [Sedwick 88] é comentado que ninguém foi capaz de analisar o algoritmo com o objetivo de conhecer o tempo de execução em função de *h*, mas propõe que "Shellsort nunca realiza mais que $n^{3/2}$ comparações" para a função aqui proposta.

```
shellSort (vetor[] a)
        h = 1;
        enquanto ((h * 3 + 1) < a.tamanho)
                h = 3 * h + 1;

        enquanto (h > 0)
                para (i de (h – 1) até a.tamanho)
                        temp = a[i];
                        j = i;
                        enquanto (j >= h) E (a[j - h] > temp)
                                a[j] = a[j - h];
                                j-= h;
                        a[j] = temp;
                h = h / 3;

        retorna;
```

Figura 2.4.2: Pseudocódigo do algoritmo ShellSort

2.5 MergeSort

O *MergeSort* ou ordenação por intercalação é um algoritmo de ordenação que utiliza a abordagem de dividir e conquistas, ou seja, desmembram o problema em vários subproblemas que são semelhantes ao problema original, podendo ser resolvidos recursivamente [Cormen 04].

O algoritmo ilustrado na Figura 2.5 opera da seguinte forma: primeiramente o vetor é dividido em dois vetores com a metade do tamanho (*n*/2). Cada um desses vetores é ordenado recursivamente, utilizando o próprio algoritmo do *mergesort*. Em seguida é feito o procedimento de *merge* ou intercalação dos dois vetores ordenados, de modo a produzir o vetor resultante. A primeira chamada ao *mergesort* deve passar os parâmetros lo = 1 e hi = tamanho do vetor.

O algoritmo implementado utiliza um vetor temporário para uso na intercalação. A função mergeSort é chamada recursivamente até que o vetor a ser ordenado tenha tamanho 1, que já é naturalmente ordenado, e retorna. A operação de intercalação do vetor de tamanho *n* possui complexidade O(n), pois o laço executa de *lo* até *hi* vezes, com complexidade hi-lo que é igual a n-1 para o vetor original. As operações desse laço (comparação e atribuição) são de complexidade O(1).

O *mergesort* obedece, portanto, a seguinte recursão: T(n) = O(1), se n = 1 e T(n) = 2T(n/2) + O(n), se n > 1. Ao construir a árvore de recursão para essa recorrência pode-se notar que a árvore completamente expandida tem lg n + 1 níveis (isto é, tem altura lg n) [Cormen 04]. Como cada nível contribuí com o custo total de O(n), então a complexidade do mergesort é da ordem de O(n . log(n)).

```
mergeSort(vetor[] a, int lo, int hi, vetor temp[])
        se (lo >= hi)
                retorna; // a[lo] já ordenado

        mid = (lo + hi) / 2;
        mergeSort(a, lo, mid, temp); // Ordena sublista a[lo..mid]
        mergeSort(a, mid + 1, hi, temp); // Ordena sublista a[mid+1..hi]

        t_lo = lo, t_hi = mid + 1;       // Intercalação
        para (k de lo até hi)
                se ((t_lo <= mid) E ((t_hi > hi) OU (a[t_lo] < a[t_hi])))
                        temp[k] = a[t_lo++];
                senão
                        temp[k] = a[t_hi++];

        para (para k de lo até hi)
                a[k] = temp[k]; // Copia de volta para a

        retorna;
```

Figura 2.5: Pseudocódigo do método de intercalação (*MergeSort*)

2.6 HeapSort

Um *heap* é um objeto que pode ser visto como uma árvore praticamente completa, onde cada nó da árvore corresponde a um elemento do vetor que armazena o valor do nó [Cormen 04]. O *heap* possui a propriedade que cada filho deve conter um elemento menor que o seu pai, considerando uma ordenação crescente (*heap* máximo, onde o maior valor está na raiz).

O *HeapSort* é um método de ordenação que utiliza a estrutura de *heaps*. A idéia do algoritmo é bem parecida com a ordenação por seleção, mas ao invés de buscar o menor elemento no tempo O(n), a estrutura *heap* permite encontrar o menor elemento em um tempo O(log(n)), onde *n* é o tamanho do vetor. O *heap* é representado por um vetor da seguinte forma: o primeiro elemento a[1] é a raiz. Os seus filhos são (2*i) e (2*i)+1, ou seja, a[2] e a[3]. Os filhos destes dois elementos são, respectivamente, a[4],a[5] e a[6][7]. E assim por diante, até alcançar o final do vetor (composto por nós “folhas”, sem filhos).

As Figuras 2.5.1 e 2.5.2 ilustram o algoritmo do *HeapSort* e a chamada *DownHeap*, respectivamente. O funcionamento do algoritmo é o seguinte: o vetor de entrada deve ser

transformado em um *heap*. Para isso basta começar o processamento do nó da posição (n/2), pois este é o último nó da árvore que possui algum filho. O motivo de se fazer isso é que para manter a propriedade de *heap* é necessário que o valor de um dado nó seja maior do que o valor de seus filhos. O algoritmo *DownHeap* é responsável por trocar a posição do nó pai pelo maior dos seus nós filhos. A complexidade de pior caso do DownHeap é de O(log(n)), pois pelo *heap* se tratar de uma árvore binária quase completa, a cada deslocamento que a sub-rotina executa para uma sub-árvore, obtemos aproximadamente a metade dos elementos.

Após transformar o vetor de entrada do *HeapSort* em um *heap*, temos que o maior elemento do vetor está armazenado na raiz (a[1]). Então é feita uma troca entre esse elemento e o último do vetor (que será o último nó-folha), operação com custo O(1), e esse elemento é removido (diminuindo o tamanho do vetor em 1 unidade). É provável que após executar essa operação o vetor deixe de ser um *heap*, requerendo, portanto, uma chamada do procedimento *downheap* para que a nova raiz seja colocada em um nó adequado. Esse procedimento é repetido N vezes, para cada elemento do *heap*, até que ele tenha tamanho 1, quando o menor valor estará armazenado na posição a[1], e os demais valores até a[n] estarão ordenados. A complexidade do laço principal é de O(n) x (O(1) + O(log(n))), o que resulta em O(n.log(n)).

```
heapSort(vetor a[])
        N = a.tamanho;
        para (k de N / 2 até 1 decrescente)
                downheap(a, k, N);
        faça
                temp = a[1];
                a[1] = a[N];
                a[N] = temp;
                N = N - 1;
                downheap(a, 1, N);
        enquanto (N > 1);
        retorna;
```

Figura 2.5.1: Pseudocódigo do algoritmo *HeapSort*

```
downheap(vetor a[], int k, int N)
        T = a[k - 1];
        enquanto (k <= N / 2)
                j = k + k;
                se ((j < N) E (a[j - 1] < a[j]))
                        j++;

                se (T >= a[j - 1])
                        break;
                senão
                        a[k - 1] = a[j - 1];
                        k = j;
```

```
        a[k - 1] = T;
        retorna;
```

Figura 2.5.2: Pseudocódigo do algoritmo *DownHeap*, utilizado pelo *HeapSort*

2.7 QuickSort

O *quicksort* é um algoritmo de ordenação cujo pior caso possui complexidade O(n2), porém o seu tempo de execução médio é de O(n lg n), sendo muito utilizado na prática pela sua notável eficiência [Cormen 04].

O seu funcionamento também utiliza o método de dividir e conquistar, da seguinte forma: o vetor a[p..r] é particionado em dois subvetores a[p..q-1] e a[q+1..r], onde *q* é um índice calculado como parte do procedimento de particionamento [Cormen 04, pp 120]. Os dois subvetores são ordenados por chamadas recursivas ao *quicksort*, e ao contrário do *mergesort*, não é necessário combinar os vetores, pois eles são ordenados localmente, sem necessidade de intercalação.

O pseudocódigo do *Quicksort* é mostrado na Figura 2.7. O seu funcionamento é o seguinte: para o subvetor de a[] que começa em lo0 e termina em hi0, se a lista tiver um elemento ele retorna o próprio elemento, e se contiver dois elementos eles são ordenados (se necessário) e o sub-vetor retornado (complexidade O(1)). Caso contrário é necessário escolher um pivô, que corresponderá ao elemento do sub-vetor correspondente à metade [(lo+hi) /2], ao redor do qual será feito o particionamento do sub-vetor. Esse particionamento será responsável por efetuar trocas dos elementos ao redor do pivô que não estão ordenados, de forma que ao final do procedimento os valores pertencerão a um dos seguintes conjuntos: elementos menores que o pivô, elementos maiores que o pivô e o próprio pivô, que será colocado no centro da lista. Até esse ponto do algoritmo a complexidade é de O(n), onde *n* será igual a (hi-lo+1), por causa do laço mais externo (enquanto lo < hi). Finalmente são feitas chamadas recursivas do próprio *quickSort* para ordenar os elementos à esquerda do pivô, e para ordenar os elementos à direita do pivô.

A análise de complexidade do *quickSort* como um todo depende do particionamento que é feito na primeira parte do algoritmo. Se o particionamento for desbalanceado, ou seja, se for gerado um sub-problema de tamanho (n-1) elementos e outro com 0 elementos [Cormen 04], a recorrência resultante é T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n), que tem a solução $T(n) = O(n^2)$, pelo método de substituição. Para o caso em que o balanceamento é o mais uniforme possível, a recorrência gerada é T(n) = 2 T(n/2) + O(n) que produz uma complexidade O(n.log(n)). Em [Cormen 04] é intuída a complexidade de caso médio, que também é da ordem de O(n.log(n)).

```
quickSort(vetor[] a, int lo0, int hi0)
        lo = lo0;
        hi = hi0;

        se (lo >= hi)
                retorna;
```

```
        senão se (lo == hi - 1) // Ordena uma lista de 2 elementos
                se (a[lo] > a[hi])
                        int T = a[lo];
                        a[lo] = a[hi];
                        a[hi] = T;
                retorna;

        // Escolhe um pivot
        pivot = a[(lo + hi) / 2];
        a[(lo + hi) / 2] = a[hi];
        a[hi] = pivot;

        enquanto (lo < hi) // busca de a[lo] em diante por um elemento > pivot
                enquanto (a[lo] <= pivot E lo < hi)
                        lo++;

                // busca de a[hi] para trás até encontrar um elemento < pivot
                enquanto (pivot <= a[hi] E lo < hi)
                        hi--;

                // Troca elementos a[lo] e a[hi]
                se (lo < hi)
                        int T = a[lo];
                        a[lo] = a[hi];
                        a[hi] = T;

        // Coloca a mediana no "centro" da lista
        a[hi0] = a[hi];
        a[hi] = pivot;

        // Chamadas recurvias, elementos a[lo0] até a[lo-1] são menores ou iguais ao
pivot, e elementos a[hi+1] até a[hi0] são maiores que o pivot.
        quickSort(a, lo0, lo - 1);
        quickSort(a, hi + 1, hi0);
        retorna;
```

Figura 2.7: Pseudocódigo do algoritmo *QuickSort*

2.8 Método das caixas

A ordenação pelo método das caixas é um algoritmo de ordenação com complexidade O(n+b) onde *b* é o valor máximo dentre os valores do conjunto. Portanto esse algoritmo só funciona para conjuntos de dados enumeráveis, como números inteiros ou números naturais.

O seu funcionamento é o seguinte: para cada possível elemento enumerável do vetor a ser ordenado, é atribuído o valor inicial 0 em um vetor que representa as “caixas” ordenadas,

cada caixa representando um valor possível (0, 1, 2, ...). Depois é feita uma varredura do vetor desordenado, fazendo com que os elementos sejam "armazenados" nas caixas correspondentes. Por fim o laço principal lê todas as caixas seqüencialmente, e se encontrar um ou mais elementos dentro dela, retira e coloca na próxima posição do vetor ordenado que será a saída do algoritmo. O pseudo-código da Figura 2.8 mostra esse algoritmo.

Se assumirmos que o vetor pode conter elementos que variam de 0 até *b*, a complexidade do algoritmo é a seguinte: o primeiro laço que zera o vetor auxiliar possui complexidade O(b). O laço seguinte que varre o vetor a ser ordenado possui complexidade O(n), onde *n* é o tamanho do vetor. Em seguida o laço principal executa O(b) vezes. Em cada iteração desse laço é executada a operação que checa se a posição atual do vetor auxiliar está zerada (enquanto), com complexidade O(1). Cada vez que essa condição for verdadeira, o elemento é "retirado" do vetor auxiliar e posicionado no local adequado no vetor resultante (operações O(1)). Essas operações serão executadas exatamente *n* vezes, pois equivale ao número de vezes em que serão encontrados elementos em uma das posições do vetor auxiliar que devem ser copiados para o vetor resultante. Logo a complexidade do laço principal é O(b) + O(n), que é a complexidade do algoritmo como um todo.

```
boxSort (vetor a[], int fim)  // para números de 0 até fim
        para (j de 0 até fim)
                vetor n[j] = 0;

        para (i de 0 até a.tamanho)
                n[a[i]] = n[a[i]] + 1;

        k = 0;
        para (j de 0 até fim)
                enquanto (n[j] > 0)
                        n[j] = n[j] - 1;
                        a[k] = j;
                        k = k + 1;
        retorna;
```

Figura 2.8: Pseudo-código do método de ordenação das caixas

3. Resultados Computacionais

Os algoritmos foram implementados com o propósito de comparar o seu desempenho computacional. A linguagem escolhida para a codificação foi C, com o compilador MinGW 5.1.4 (versão minimalista do GCC para Windows), sistema operacional Windows XP. Todos os testes foram executados em um Intel Core 2 Duo CPU 4400 2.0 GHz com 2 Gbytes de RAM.

Os desempenho computacional dos algoritmos implementados é mostrado a seguir. Para computar o tempo de execução dos algoritmos foi utilizada a função *clock()* da biblioteca *time.h*, e com uma resolução de 0,1 ms. Foi considerada a média de 10 execuções para

computar o tempo de execução do algoritmo. Os gráficos foram produzidos pelo software GnuPlot versão 4.2.4.

O vetor de entrada para os algoritmos variou para cada execução. Cada vetor possui números inteiros que variam de 0 até N, onde N é o tamanho do vetor, gerados aleatoriamente com possibilidade de repetição. O mesmo vetor gerado serve como entrada para todos os algoritmos. Apenas para a análise do algoritmo de ordenação do método das caixas a grandeza dos valores do vetor variou, conforme explicado na seção apropriada.

3.1 Resultados dos algoritmos de complexidade $O(n^2)$

A Figura 3.1.1 mostra o desempenho do algoritmo *Bubble Sort*, que possui complexidade $O(n^2)$ conforme visto na Seção 2.1. O gráfico está ajustado com os pontos da curva $O(a.x^2)$. As execuções variaram o tamanho do vetor (*n*) de 10.000 até 130.000 elementos, quando o tempo de execução ultrapassou 1 minuto e meio (90.000 ms).

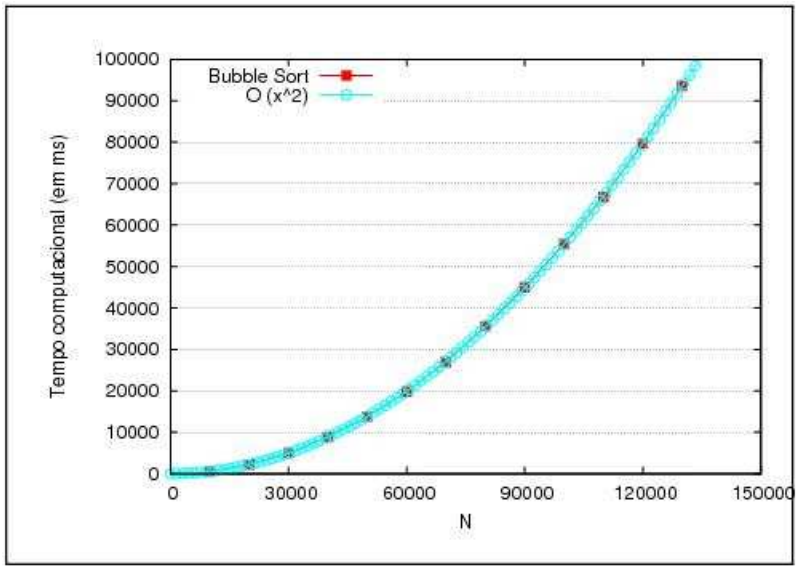

Figura 3.1.1: Tempo de execução ao algoritmo *Bubblesort* ajustado à curva $O(x^2)$

A Figura 3.2.1 mostra o gráfico de execução dos três algoritmos com complexidade $O(n^2)$: *BubbleSort*, *SelectionSort* e *InsertionSort*. Pode-se observar que embora os três algoritmos tenham a mesma complexidade de pior caso, nas execuções o método da bolha (*Bubblesort*) tem um desempenho pelo menos duas vezes pior que os outros dois. O algoritmo que obteve o melhor desempenho foi a ordenação por inserção, que conseguiu executar a ordenação em vetores de até 300.000 elementos em um tempo inferior a 90.000 ms.

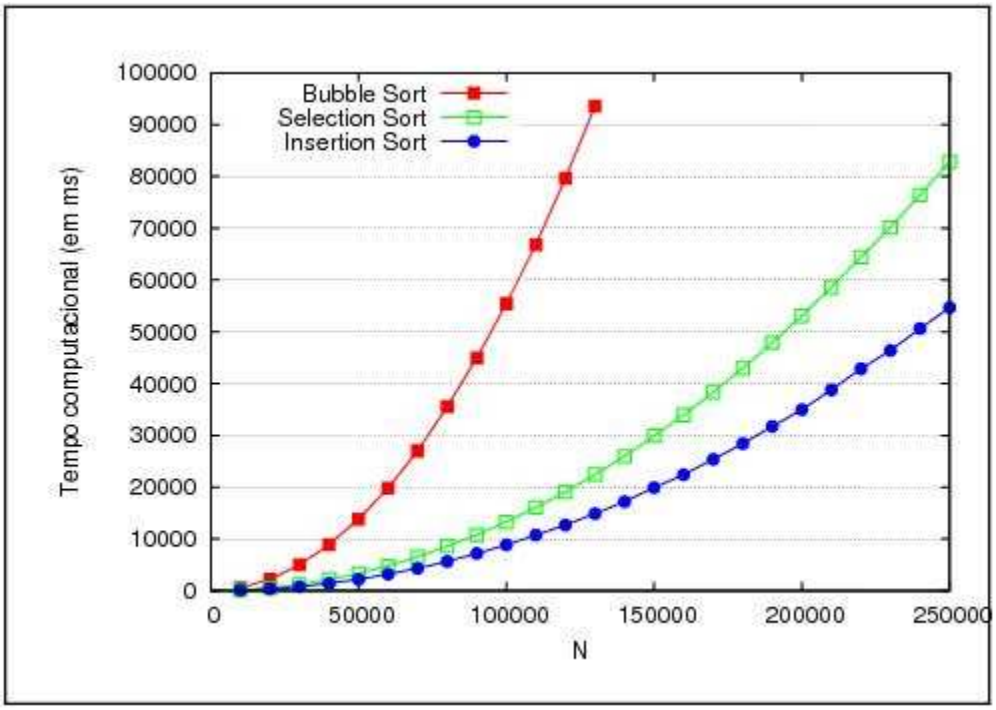

Figura 3.1.2: Comparação do desempenho dos algoritmos de complexidade $O(n^2)$

3.2 Resultados dos algoritmos de complexidade O(n.log(n))

A Figura 3.2.1 mostra o desempenho dos algoritmos de complexidade O(n.log(n)): *HeapSort, MergeSort e QuickSort.* As execuções variaram desde vetores com 10.000 elementos até o máximo alcançado de 100.000.000 (cem milhões) de elementos. O tempo de execução foi limitado até cerca de 50.000 ms. O gráfico está ajustado com os pontos da curva O(a.x.log(x)). O melhor desempenho médio foi alcançado pelo algoritmo *QuickSort*.

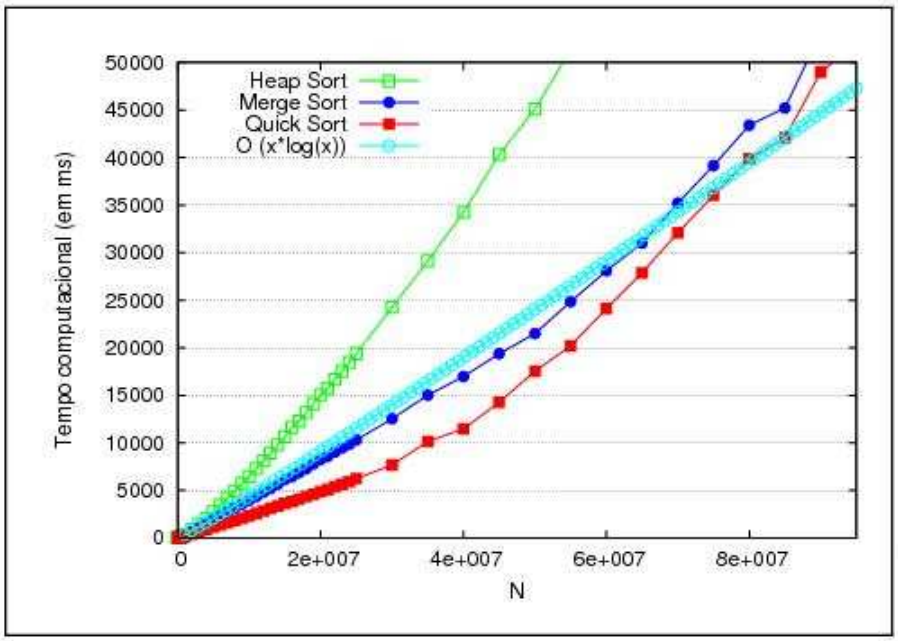

Figura 3.2.1: Comparação do desempenho dos algoritmos de complexidade O(n.log(n))

3.3 Resultados do algoritmo ShellSort

A Figura 3.3.1 mostra o desempenho do algoritmo *Shell Sort*, que possui complexidade de pior caso de no máximo $O(n^{3/2})$ conforme visto na Seção 2.4. O gráfico está ajustado com os pontos da curva $O(a.x^2)$. As execuções variaram o tamanho do vetor (n) de 10.000 até 25.000.000 elementos, com o tempo de execução próximo de 1 minuto (60.000 ms).

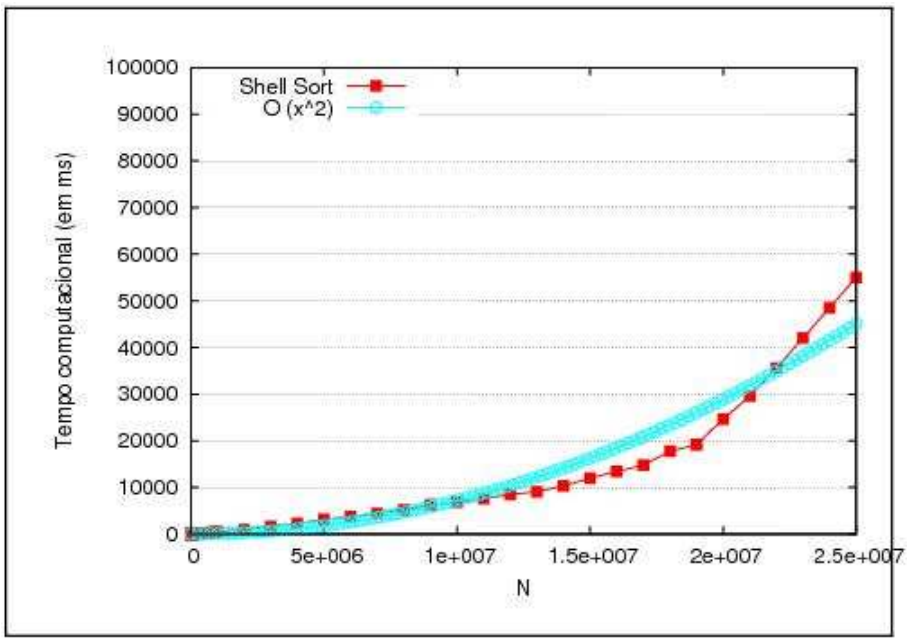

Figura 3.3.1: Tempo de execução ao algoritmo *Shellsort* ajustado à curva $O(x^2)$

O limite superior do número de elementos do vetor foi bem maior no *ShellSort* do que com os algoritmos de complexidade $O(n^2)$. A Figura 3.3.2 mostra que ele pode ser comparado com os algoritmos mais eficientes de complexidade O(n.log(n)), embora o seu desempenho ainda seja pior que os outros três algoritmos.

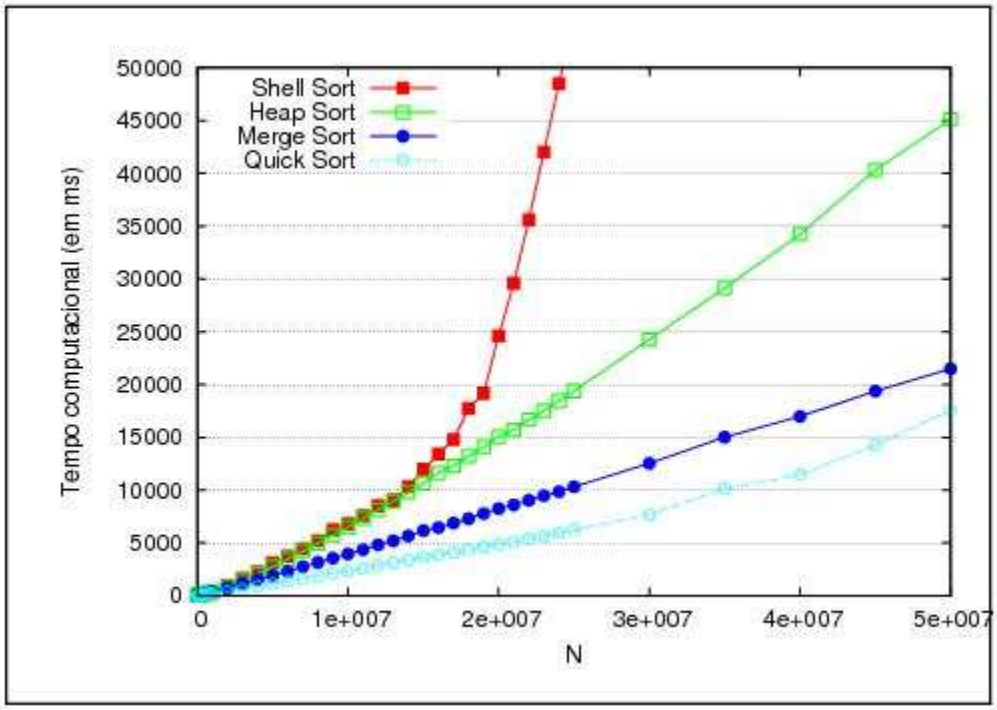

Figura 3.3.2: Comparação do desempenho do *ShellSort* com outros algoritmos

3.4 Resultados do algoritmo de ordenação das caixas

As comparações experimentais do método de ordenação das caixas consideraram não apenas o número de elementos, mas também o limite superior dos valores gerados de forma aleatória para compor o vetor, pois ao contrário dos demais algoritmos, o método das caixas é impactado por esse dado, conforme visto na Seção 2.8.

A Figura 3.4.1 mostra a execução normal do *Quicksort* comparada com o método das caixas, onde cada elemento do vetor pode variar de 0 até *n*, onde *n* é o tamanho do vetor – da mesma forma como foram conduzidos os testes anteriores. O parâmetro *b* indica o maior valor que um elemento pode assumir. A complexidade do algoritmo para $b = n$ é de O(b+n) = O(2n) = O(n), e é possível observar que para essa configuração o tempo de execução é inferior ao *Quicksort*, com vetores variando em tamanho de 10.000 até 100.000.000.

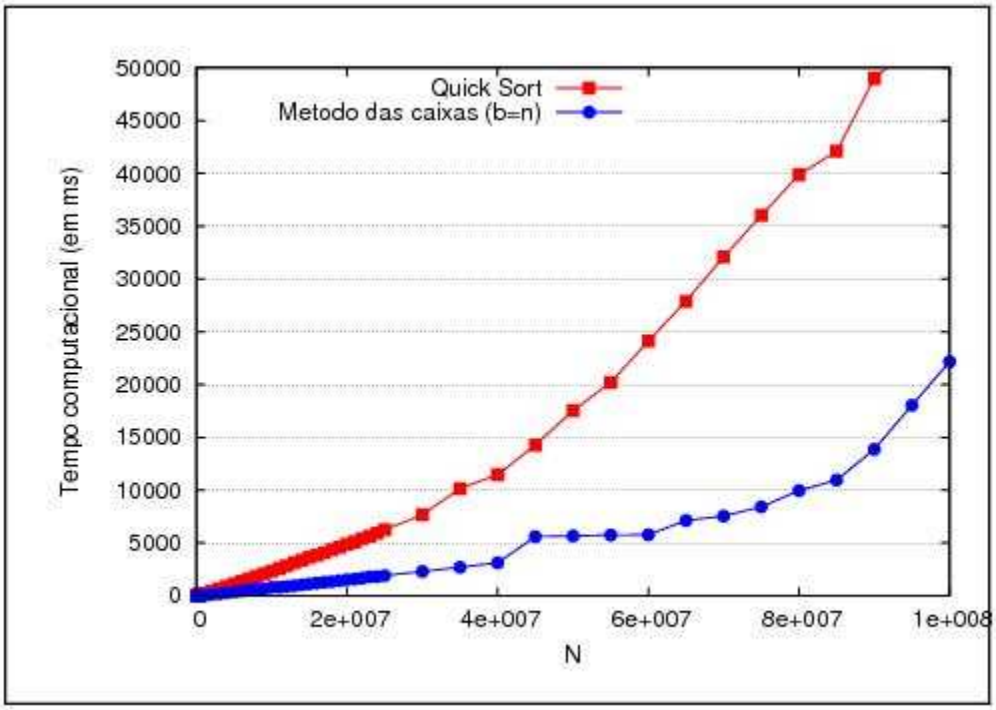

Figura 3.4.1: Desempenho do Método das caixas e do *QuickSort* variando *n*.

A Figura 3.4.2 faz a mesma comparação do *QuickSort* com o método das caixas quando o tamanho do vetor (*n*) é mantido fixo, e é variado o valor máximo que o vetor pode conter (*b*), indicado pelo eixo horizontal. Dois conjuntos distintos de execuções são feitos: o primeiro utilizando n = 1.000.000, e a segunda com n = 10.000.000. Nessa comparação o tempo de execução do *Quicksort* é virtualmente o mesmo independente do valor de *b*, se mantendo constante na faixa de 250 ms para n = 1.000.000 e na faixa de 2.200 ms para n = 10.000.000. Já para o método de ordenação das caixas, cuja complexidade é da ordem de O(n + b), o tempo de execução aumenta em função das duas variáveis. Para n = 10.000.000 e *b* acima de 150.000.000, há uma distorção na curva possivelmente por limitações de memória que obrigou o computador a realizar *swap* de disco para gerenciar os vetores de execução do algoritmo.

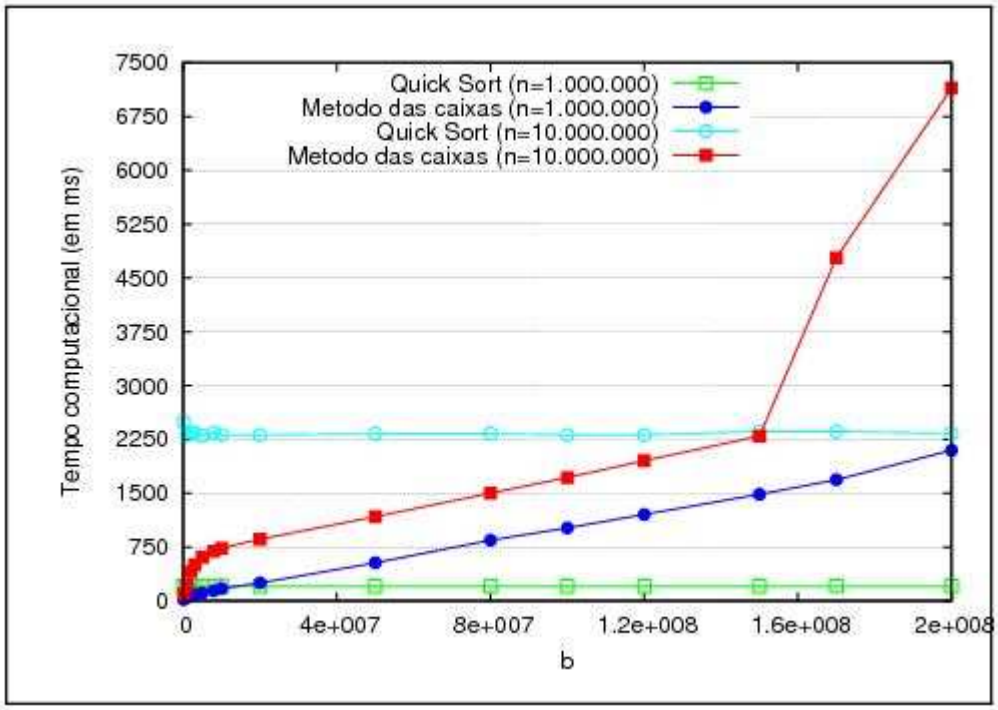

Figura 3.4.2: Desempenho do Método das caixas e do *QuickSort* variando *b*.

4. Conclusões

Neste trabalho foram analisados e implementados oito algoritmos tradicionais de ordenação para vetores inteiros de tamanho *n*. Foi realizada uma comparação experimental entre os tempos de execução dos algoritmos, comparados também com os seus tempos teóricos.

A Figura 4.1 ilustra que para vetores grandes e com a mesma configuração sugerida no trabalho, não é viável a utilização de algoritmos de ordenação com complexidade $O(n^2)$, pois o que obteve o melhor desempenho (*Insertion Sort*) não é capaz de lidar com o tamanho dos vetores ordenados pelo *Shell Sort* ou os algoritmos de complexidade O(n.log(n)).

Foi analisado também o método de ordenação das caixas, que diferente dos demais algoritmos, depende não apenas do tamanho do vetor de entrada (*n*), mas também da ordem de grandeza dos seus elementos (*b*).

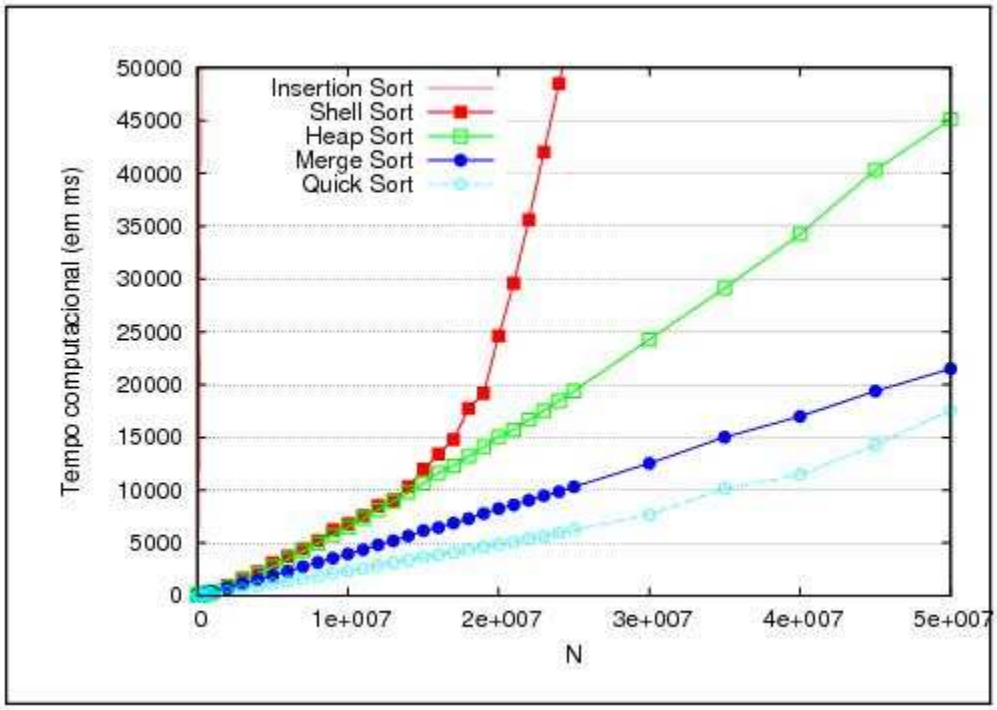

Figura 4.1:Comparação entre os métodos de ordenação.

Na prática foi constatado que o melhor algoritmo em termos de desempenho computacional, entre os analisados nesse trabalho, foi o *Quicksort*, exceto em casos em que a ordem de grandeza dos elementos não é muito grande em relação ao tamanho do vetor, onde o método de ordenação das caixas seria superior. Alguns algoritmos poderiam se beneficiar ou se prejudicar caso existam particularidades do vetor de entrada, como vetores já ordenados, parcialmente ordenados ou ordenados de maneira inversa, mas essas análises não fizeram parte do escopo desse trabalho.

Referências

[Cormen 04] T.H. Cormen, C.E. Leiserson, e R.L. Rivest. *Algoritmos: Teoria e Prática* (tradução da 2º edição de *Introduction to Algorithms*). MIT Press & McGraw-Hill, 2001.

[Knuth 98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley. ISBN 0-201-89685-0. 1998.

[Sedgewick 88] Sedgewick, R. *Algorithms*. Reading, MA: Addison-Wesley; 1988.