Universidade Federal do Espírito Santo

Eliezer de Souza da Silva

Estudo Comparativo de Algoritmos de Ordenação

SÃO MATEUS

2010

Universidade Federal do Espírito Santo

Eliezer de Souza da Silva

Estudo Comparativo de Algoritmos de Ordenação

Trabalho da disciplina de Tópicos Especiais de Programação II (C++) submetido ao Departamento de Engenharias e Computação da Universidade Federal do Espírito Santo como requisito parcial para passar na matéria.

Orientador:
Prof. Renato Moraes, D.Sc.

SÃO MATEUS

2010

Projeto Final de Curso

Eliezer de Souza da Silva

Trabalho da disciplina de Tópicos Especiais de Programação II (C++) submetido ao Departamento de Engenharias e Computação da Universidade Federal do Espírito Santo como requisito parcial para passar na matéria.

Aprovada por:

Prof. Renato Moraes, D.Sc. / DECOM-UFES (Orientador)

Prof. Rodolfo Villaça, M.Sc. /DECOM-UFES

Prof. Roney Pignaton, D.Sc. / DECOM-UFES

São Mateus, 21 de Dezembro de 2010.

Resumo

Os algoritmos de ordenação desempenham um papel fundamental tanto no estudo teórico de algoritmos quanto nas aplicações práticas. Existem uma variadade de algoritmos de ordenação propostos na literatura atual, de modo que estudar o comportamento destes algoritmos, condições de melhor desempenho e suas complexidades computacionais se torna necessário na determinação de algoritmos mais eficientes ou adequados para determinados problemas. Este trabalho apresenta um estudo comparativo de seis algoritmos de ordenação, visando a verificação experimental da complexidade computacional destes.
Palavras-chave: complexidade computacional, análise de algoritmos, algoritmos de ordenação.

Abreviações

QK1 - Quicksort com particionamento com pivô no elemento médio

QK2 - Quicksort com particionamento com pivô aleatório

MS - Mergesort

HS - Heapsort

IS - InsertionSort

BB - BubbleSort

SH1 - ShellSort com sequência de incrementos com bons resultados experimentais

SH2 - ShellSort com sequência de incrementos de Incerpi-Sedgewick

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os algoritmos de ordenação ou classificação consituem uma classe de algoritmos extremamente estudada e muito popular. Seu estudo é parte vital de praticamente todo curso na área da Computação e tem aplicações práticas no dia-a-dia. Em diversas aplicações a ordenação é uma das etapas, dentre muitas outras a serem efetuadas, de modo que selecionar o melhor algoritmo de ordenação é extremamente importante nestes casos. Em busca em índices previamente ordenados, por exemplo, é a maneira mais eficiente de busca por um elemento qualquer de uma coleção. Uma sequência previamente ordenada tem aplicação prática na resolução dos seguintes problemas: busca, par mais próximo (dado uma sequência de números), unicidade de elementos, distribuição de frequência, convex hull, etc. [4]

Estudar a complexidade de um algoritmo é determinar o custo computacional (tempo ou espaço) para a execução deste algoritmo. Basicamente podemos fazer este estudo por duas vias: através da teoria da Complexidade Computacional, que envolve o estudo de classes inteiras de algoritmos, usando uma metodologia mais genérica, ou através da Análise de Algoritmos. A análise de algoritmos é a área de ciência da computação que se preocupa em determinar a complexidade (custos de tempo da cpu ou de memória) de um determinado algoritmo e uma determinada implementação. Este tipo de análise foi popularizado por Donald E. Knuth nos seus livros da séries "The Art Of Computer Programming"[2].

Diversos algoritmos de ordenação já foram propostos ao longo da história e e muitos resultados já foram encontrados, estabelecendo, por exemplo, limites inferiores para a complexidade de tempo de um algoritmo de ordenação. Neste trabalho foca-se no estudo do BubbleSort (ordenação por bolha), InsertionSort (Inserção Ordenada), MergeSort (In-

tercalação), QuickSort, HeapSort e ShellSort.

1.1 Formulação do Problema

De uma maneira formal podemos dizer que dado uma sequência $<v_0, v_2, ..., v_{n-1}>$ de elementos pertencentes a um certo domínio D, um conjunto K de chaves, dotado de uma relação de ordem, uma função bijetiva $key : D \longrightarrow K$ que associa cada elemento a uma chave, o problema da ordenação consiste em achar uma permutação da sequência inicial tal que para cada i, $key(v_i) \leqslant key(v_{i+1})$.
Em diversas situações práticas o domínio K das chaves coincide com o domínio D dos elementos.

1.2 Metodologia

A metodologia adotada consiste, inicialmente, no estudo teórico da complexidade e comportamento de cada algoritmo, em seguida a implementação destes na linguagem C++ e experimentação e medição de tempo no resolução de instâncias variadas do problema. As instâncias serão divididas em instâncias de melhor caso, pior caso e caso médio (aleatória). As medições de tempo serão apresentadas em um gráficos de tempo x tamanho da instância.
Todas os exemplos e medições foram feitas em um computador com um processador Intel Pentium Dual-Core T3200 (2GHz), 1GB de memória e rodando uma distribuição Ubuntu Linux (9.0.4).

Capítulo 2

Algoritmos de Ordenação

Existem diversas estratégias utilizadas na resolução do problemas computacionais. Os algoritmos aborados neste trabalho abrangem várias técnicas: dividir-para-conquistar, força bruta, uso de estruturas de dados eficientes, seleção, trocas, etc. As estratégias utilizadas por cada algoritmo determinam também a complexidade computacional de tempo e espaço do algoritmo. Todos os algoritmos de ordenação considerados aqui podem ser classificados como algoritmos comparativos, ou seja, o mecanismo usado para determinar a ordem dos elementos é uma função ou operador de comparação entre dois elementos. Na considerações da complexidade dos algoritmos iremos em geral basear esta análise no número de comparações que o algoritmo realiza e o número de trocas/atribuições.

2.1 BubbleSort

A ordenação bubblesort é um dos métodos mais simples de ordenação e utiliza a estratégia de trocas entre elementos sucessivos da sequência. O algoritmo faz diversas varreduras na sequência e sempre que dois elementos sucessivos estão na ordem errada, ela faz uma troca de posição. Ele pára quando não é feito nenhuma troca ao longo de uma varredura. O nome ordenação por 'bolha' se deve a dinâmica das trocas de elementos. Um elemento não vai ser colocado diretamente na sua posição certa ou definitiva, mas através das trocas sucessivas com os elementos vizinhos, 'borbulhar' até a sua posição certa.
O bubblesort tem complexidade de caso médio e pior caso $O(n^2)$, a sua complexidade de melhor caso é $O(n)$. Dentre todos os algoritmos de ordenação elementar, é considerado o algoritmo com pior performance e exemplo de projeto fraco de algoritmos, certos livros de algoritmos nem sequer apresentam ele, e preferem logo apresentar o InsertionSort.

Algoritmo 2.1 Algoritmo de Ordenação usando o método de Bolhas

Procedure $bubblesort(v[])$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$

```
1: trocou ← verdadeiro;
2: while trocou do
3:    trocou ← falso ;
4:    for i = 0, 1, .., n − 2 do
5:       if v[i + 1] < v[i] then
6:          troca(v[i], v[i + 1]) ;
7:          trocou ← verdadeiro ;
8:       end if
9:    end for
10: end while
```

2.2 InsertionSort

O Insertionsort é um método elementar de ordenação, que usa a inserção ordenada de elementos como estratégia de ordenação. A inserção ordenada é a maneira que normalmente usamos para ordenar um baralho de cartas, mantendo uma pilha de cartas já ordenadas enquanto inserimos novas cartas nesta pilha ordenada, nas posições certas. Ao ordenarmos um vetor podemos usar a mesma estratégias, no entanto temos que 'empurrar' alguns elementos para colocar um novo elemento na posição 'certa'. Este tipo de ordenação é muito eficiente em listas encadeadas, principalmente se estas já estiverem parcialmente ordenadas. Ele apresenta boa performance para arquivos pequenos, fato este que será explorado no Shellsort. O loop externo do algoritmo vai rodar $n - 1$ vezes, para cada

Algoritmo 2.2 Algoritmo de ordenação Insertionsort

Procedure $Insertionsort(v[], n)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiro n indicando o fim da sequencia

```
1: for j ← 1 to n − 1 do
2:    x ← v[j];
3:    i ← j − 1;
4:    while  i ≥ 0 e v[i] > x do
5:       v[i + 1] ← v[i];
6:       i ← i − 1;
7:    end while
8:    v[i + 1] ← x;
9: end for
```

vez que ele rodar, no pior caso, o loop interno irá rodar $1, 2, ..., n - 1$ vezes, e portanto a complexidade do algioritmo estaria na ordem da soma dos $n - 1$ primeiros naturais, ou seja, $O(n^2)$. Por outro lado, o melhor caso é $O(n)$, uma vez que para um vetor orde-

nado, ele não vai percorrer o subconjunto ordenado todo para achar a posição certa de um elemento. O caso médio a complexidade é $O(n^2)$[3].

2.3 ShellSort

O Shellsort é um algoritmo de ordenação proposto em 1959 por Donald Shell, que pode ser entendido como uma generalização do insertionsort e que tira proveito do fato que o Insertionsort funciona bem para sequências pequenas e quase ordenadas. A ideia basica é ver o vetor dos números a ser ordenados como várias listas interlaçadas, com uma distância h entre os elementos vizinhos de uma dada sequência. O algoritmo faz várias varreduras na sequência, sempre ordenando os elementos de todas sublistas a uma distância h, começando com sublistas de 2 elementos, depois 4, ..., repetindo o processo que diminui o número de sublistas e aumenta o tamanho de cada sublista que sempre estará quase ordenada no início de cada varredura.
Este sequência de números h é a principal determinante da performance do Shellsort, sendo que existem sequências bem estudadas que apresentam resultados muito próximos aos dos algoritmos com complexidade $O(nlogn)$. Dependendendo da sequência utilizada o shellsort pode ter uma complexidade média de $O(n^{3/2})$, $O(n^{4/3})$, chegando a apresentar complexidade $O(n(logn)^2)$, usando a sequência gerada pelo método de Pratt [3].
Na implementação prática usaremos sequência proposta por Sedgewick [3].

Algoritmo 2.3 Algoritmo de ordenação Shellsort

Procedure $Shellsort(v[], n)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiro n indicando o fim da sequencia e um método para gerar a sequencia de números h ou a própria sequência

1: gerar a sequência $h[1, ..., k]$;
2: **for** h da sequência de incrementos **do**
3: Ordena cada subsequência $v[0, h, ..]$, $v[1, h+1, 2h+1, ..]$, ... , $v[h-1, 2h-1, ..]$ usando insertionsort
4: **end for**

2.4 MergeSort

Os algoritmos considerados anteriormente, Shellsort e Insertionsort, usam a estratégia da inserção para alcançar os objetivos de ordenação. Nesta seção examinaremos um algoritmo que usa uma estratégia diferente: *merging*, ou intercalação ordenada.
O processo de intercalação ordenada consiste no formação de uma nova sequência orde-

nada usando duas sequências previamente ordenadas. Este processo é linear no tamanho das duas sequências previamente ordenadas, mas exige o uso de espaço de memória linear com o tamanho da entrada.

O algoritmo de *mergesort* utiliza o paradigma de dividir-para-conquistar para ordenar a sequência dada, ele divide a sequência em duas duas sequencias menores com metade do tamanho da sequência original, e recursivamente ordena esta sequencias menores, resolvendo os subproblemas menores ele combina as duas soluções usando o procedimento de intercalação.

A complexidade do mergesort não depende da sequência de entrada, uma vez que a sequência é sempre subdivida no "meio". Vejamos, analisando o pseudo-codigo do algoritmo 2.4 podemos escrever a seguinte equação para descrever a complexidade do mergesort.

$$T(n) = 2 \cdot T(\frac{n}{2}) + b \cdot n = 2 \cdot (2 \cdot T(\frac{n}{4}) + b \cdot \frac{n}{2}) + b \cdot n \tag{2.1}$$

$$= 4 \cdot T(\frac{n}{4}) + 2 \cdot b \cdot n = 4 \cdot (2 \cdot T(\frac{n}{8}) + b \cdot \frac{n}{4}) + 2 \cdot b \cdot n \tag{2.2}$$

$$= 8 \cdot T(\frac{n}{8}) + 3 \cdot b \cdot n = 2^i \cdot T(\frac{n}{2^i}) + i \cdot b \cdot n, 2^i \leq n \tag{2.3}$$

Se dissermos que $T(1) = c$ e fizermos $2^i = n \Rightarrow i = log_2 n$, obteremos a expressão $T(n) = n \cdot c + b \cdot n \cdot log_2 n$ para a complexidade do mergesort. Deste modo, independente da entrada a complexidade do mergesort será $O(nlogn)$.

O Mergesort apresenta a vantagem de ter uma garantia de performance independente da entrada, sua complexidade é sempre $O(nlogn)$, somando a este fato, ele é um algoritmo estável (se a intercalação subjacente for estável [3]), no entanto requer espaço extra na ordem de $O(n)$.

Algoritmo 2.4 Algoritmo de Ordenação MergeSort

Procedure $mergesort(v[], p, q)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiros p e q indicando o inicio e fim da sequencia

1: **if** $p < q$ **then**
2: $m \leftarrow \lfloor \frac{p+q}{2} \rfloor$;
3: $mergesort(v, p, m)$;
4: $mergesort(v, m + 1, q)$;
5: $intercalaOrdenado(v, p, m, q)$;
6: **end if**

2.5 QuickSort

Algoritmo introduzido em 1960 por C. A. R. Hoare [1], baseado em uma ideia simples, mas que na prática apresenta-se muito eficiente. Este fato, alinhado à relativa facilidade de implementação e ao baixo consumo de recursos computacionais (cpu e memória), contribuiu para a popularização deste algoritmo, que pode ser encontrado em diversas bibliotecas de programação como o algoritmo padrão de ordenação [3].

Tal como o Mergesort, o Quicksort é um algoritmo de dividir-para-conquistar. Ele aproveita da estrutura recursiva da ordenação, e utiliza a ideia do particionamento da sequência em duas partes, ordenando então cada parte separadamente.

Algoritmo 2.5 Algoritmo de Ordenação QuickSort

Procedure $quicksort(v[], p, q)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiros p e q indicando o inicio e fim da sequencia

1: **if** $p < q$ **then**
2: $m \leftarrow particiona(v, p, q)$;
3: $quicksort(v, p, m-1)$;
4: $quicksort(v, m+1, q)$;
5: **end if**

Algoritmo 2.6 Algoritmo de Particionamento em duas subsequencias

Procedure $particiona(v[], p, q)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiros p e q indicando o inicio e fim da sequencia

1: $x \leftarrow v[p]$;
2: $i \leftarrow p - 1$;
3: $j \leftarrow r + 1$;
4: **while** $0 = 0$ **do**
5: **while** $v[j] \leq x$ **do**
6: $j \leftarrow j - 1$;
7: **end while**
8: **while** $v[i] \geq x$ **do**
9: $i \leftarrow i + 1$;
10: **end while**
11: **if** $i < j$ **then**
12: $troca(v[i], v[j])$;
13: **else**
14: **return** j
15: **end if**
16: **end while**

Neste ponto, vale uma explicação do significado do particionamento utilizado. Dado um certo elemento $v[m]$ (o pivô), o particionamento da sequência v deve ser tal que a

seguinte condição seja satisfeita $v[i] \leqslant v[m], \forall i < m$ e $v[k] > v[m], \forall i > m$. Este processo de separação é linear com o tamanho do vetor de entrada no número de comparações feitas e de trocas e no pior caso vai trocar todos elementos de lugar e percorrer o vetor inteiro.

A performance do quicksort, depende então da qualidade deste particionamento, ou seja quanto mais balanceado for o particionamento mais rápido será o quicksort, uma vez que o particionamento é feito h, onde h é a profundida da recursão alcançada pelo algoritmo. Esta profundida é $O(logn)$ no melhor caso (altura de uma árvore binária balanceada) e $O(n)$ no pior caso (árvore binária degenerada).

A qualidade deste particionamento por sua vez depende da sequência de entrada e da escolha do pivô. Existem diferentes propostas na literatura para fazer este particionamente menos dependente da sequência de entrada [2, 1].

Deste modo a complexidade de pior caso do quicksort é $O(n^2)$ e a complexidade de melhor caso é $O(nlogn)$. Uma vez que o pior caso ocorre somente para alguns casos específicos ela não domina o cálculo do caso médio, em média o quicksort apresenta complexidade $O(nlogn)$ [3, 1].

2.6 HeapSort

Este algoritmo utiliza um paradigma de projeto diferente de todos os outros considerados anteriormente, mas ao mesmo tempo combinando técnicas mais elementares de ordenação. O SelectionSort é um algoritmo de ordenação que mantem um subconjunto ordenado e outro desordenado da sequência original, e a cada iteração retira o *menor elemento* do conjunto não-ordenado e adiciona ao conjunto ordenado. A extração do menor elemento de uma sequência não ordenada tem complexidade $O(n)$, pois sempre seria necessário percorrer a sequência inteira para verificar que de fato um certo elemento é menor da sequência. Desta forma, como o selectionsort repete a iteração n vezes para ordenar o vetor inteiro, a sua complexidade é $O(n^2)$.

No entanto fica a questão, poderíamos organizar os elementos não-ordenados de maneira tal que a operação de extrair o mínimo se torne menos custosa? A resposta é sim, e a estrutura de dados usada para tal propósito é o *heap*. A operação de extração de mínimo (ou máximo, dependendo do tipo de heap) é $O(1)$ e a reconstrução do *heap* depois da extração é $O(logn)$. Deste modo a ordenação *heapsort* tem complexidade $O(nlogn)$.

2.6.1 Heaps

É uma estrutura de dados simples e elegante que oferece suporte eficiente às operações *insert* e *extract-min* de uma fila de prioridade. Esta eficiência se deve ao fato do heap manter uma ordem parcial entre os pais e filhos da árvore, que apesar de ser mais fraco que uma ordem total, é melhor que a ordem randômica [4].
Assim, o *heap binário* é uma árvore binária *quase cheia* e com uma relação de ordem parcial entre os elementos adjacentes (nós pais e filhos). Devido ao fato de ser uma árvore quase cheia, ele pode ser armazenado de maneira compacta e eficiente em um vetor. Para este fim, padroniza-se que o nó filho à esquerda do elemento $v[k]$ do vetor é o $v[2k]$ e o nó filho à direita é o $v[2k+1]$, e para um dado elemento $v[n]$ com $n > 1$, o pai deste elemento se encontra na posição $\lfloor \frac{n}{2} \rfloor$.
Iremos considerar a partir deste ponto que estamos tratando de um *max-heap*, que é um heap onde a relação de ordem é tal que um nó pai é sempre maior ou igual aos seus nós filhos.
Um dos algoritmos mais importantes para esta estrutura de dados é o *Corrige-descendo*, conhecido também como *heapify* ou *fixdown*. Este algoritmo resolve o seguinte problema associado à inserções e deleções: dado um vetor $v[1..n]$ e um índice i tal que as subárvores com raízes nos elementos $v[2i]$ e $v[2i+1]$ são max-heaps, rearranjar este vetor de modo que o elemento $v[i]$ seja a o nó raíz de um max-heap.
A ideia do algoritmo é a seguinte: se $v[i]$ é maior ou igual a seus filhos o algoritmo pára, caso contrário troque A[i] com o maior dos filhos e repita o processo para a subárvore cuja raíz é o filho envolvido na troca.

No pior caso o algoritmo vai percorrer os níveis do nó i até os nós folhas, ou seja, a altura do nó i até o nó folha que é $O(log(n/i)$.
A construção de um heap, por sua vez, pode ser feito em tempo linear, através de chamadas sucessivas à função *CorrigeDescendo*. Deste modo o algoritmo de construção do Heap tem complexidade de tempo $O(n)$.

2.6.2 O Algoritmo Heapsort

Uma vez que comprendemos a estrutura de dados heap, o algoritmo de ordenação heapsort fica fácil de ser analisado e compreendido. Anteriormente comentou-se que o heapsort é um algoritmo de seleção, ou seja a organização básica do algoritmo é o mesmo de algoritmo de ordenação por seleção e é um algoritmo de ordenação *in place*, por não precisar manter nenhuma estrutura auxiliar além do próprio vetor original, no processo de ordenação.

Algoritmo 2.7 Algoritmo de correção da estrutura de max-heap

Procedure $CorrigeDescendo(v[], i, n)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiros i e n indicando o ponto a começar a correção e fim da sequencia

```
1: j ← i ;
2: while 2j ≤ n do
3:    if 2j < n and v[2j] > v[2j + 1] then
4:       f ← 2j;
5:    else
6:       f ← 2j + 1;
7:    end if
8:    if v[j] ≥ v[f] then
9:       j ← n;
10:   else
11:      troca(v[j], v[f]);
12:      j ← f;
13:   end if
14: end while
```

Algoritmo 2.8 Algoritmo de construção de um max-heap

Procedure $ConstroiMaxHeap(v[], n)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiro n indicando o fim da sequencia

```
1: j ← i ;
2: for i ← ⌊n/2⌋ downto 1 do
3:    CorrigeDescendo(v, i, n);
4: end for
```

O primeiro passo do algoritmo é o construção do heap máximo, chamando o procedimento $ConstroiMaxHeap$, processo com complexidade $O(n)$, em seguida retira-se o primeiro elemento (o maior elemento do heap), trocando com um elemento na posição final, diminuindo em seguida o tamanho do heap e corrigindo a estrutura deste.

Algoritmo 2.9 Algoritmo de ordenação Heapsort

Procedure $Heapsort(v[], n)$

Require: Sequência de elementos $<v_0, v_2, ..., v_{n-1}>$, inteiro n indicando o fim da sequencia

1: $ConstroiMaxHeap(v, n)$;
2: **for** $i \leftarrow n$ **downto** 2 **do**
3: $troca(v[1], v[n])$;
4: $CorrigeDescendo(v, n-1, 1)$;
5: **end for**

O loop principal do procedimento, faz $n-2$ chamadas ao $CorrigeDescendo$ e como o $CorrigeDescendo$ tem complexidade $O(n)$ e o $ConstroiMaxHeap$ tem complexidade $O(n)$, a complexidade do Heapsort é $O(nlogn)$. Esta complexidade vale para o melhor caso, pior caso e caso médio, uma vez que o loop principal não depende de nenhuma propriedade do vetor de entrada, e o processo de retirada de um elemento e reconstrução do heap sempre levar um tempo $O(n)$.

Capítulo 3

Resultados e Discussões

As instâncias de teste foram organizadas em dois grupos, entradas grandes (n =100000 a 3000000) e entradas pequenas (até 100000), sendo as primeiras para os algoritmos mais rápidos (quicksort, shellsort, heapsort e mergesort) e as menores para os algoritmos com tempos quadráticos (bubblesort, insertionsort e pior caso do quicksort não aleatorizado). Cada instância tinha 4 tipos de sequências: estritamente crescente, estritamente decrescente, todos elementos repetidos e aleatória. A medição do tempo das sequências aleatórias foi feito tirando a média do tempo para cada sequência do mesmo tamanho (foram geradas quatro do mesmo tamanho).

Foram implementadas duas versões do quicksort, uma com o pivô no elemento médio, e outra com pivô aleatório. O pior caso do quicksort foi rodado junto com os testes das instâncias menores.

Testou-se também duas versões do shellsort, com sequências de incrementos distintas, a sequência de Sedgewick-Incerpi e outra que experimental apresenta bons resultados.

Dentre todos, o quicksort com elemento médio como pivô foi o que apresentou melhor desempenho (caso médio, sequência crescente e decrescente), depois dele aparece o quicksort aleatorizado, que na verdade fez aproximar o desempenho de todos outros casos para o desempenho da sequência aleatória, em seguida temos o Shellsort com a sequência experimental, depois com a sequência de Sedgewick-Incerpi, e depois o heapsort e o mergesort, o insertionsort e o bubblesort.

O heapsort e o mergesort apresentam desempenhos de pior e melhor caso muito parecidos, ficando perto de 2s para instancia de tamanho maior que 10^6, mas o caso médio do mergesort é praticamente igual ao melhor e pior caso, enquanto para o heapsort o caso médio apresenta tempo maior que os outros.

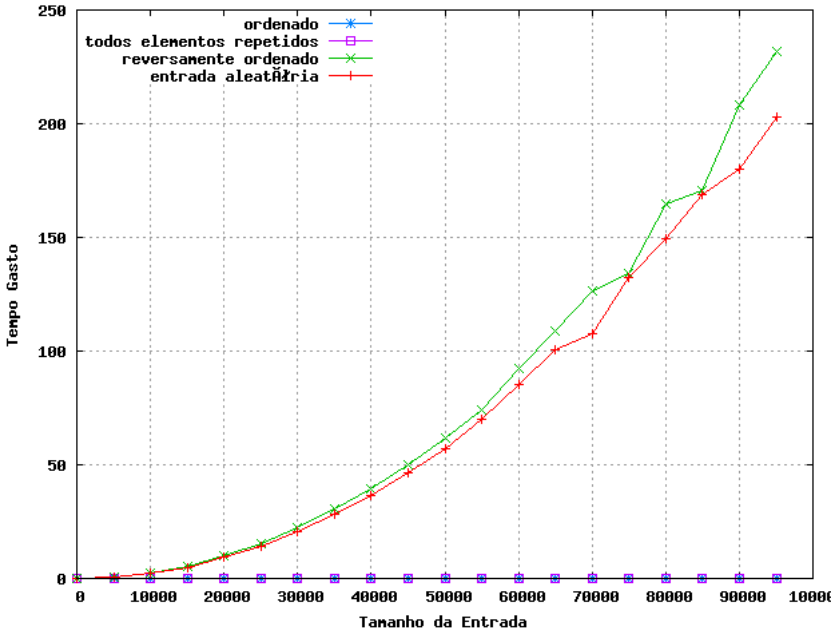

Figura 3.1: Gráficos do tempo por tamanho de entrada do algoritmo Bubblesort: entrada crescente, decrescente, igual e randômica

3.1 Bubblesort

Apresentou os piores tempos de caso médio e pior caso, chegando a gastar cerca de 300s para ordenar um vetor de aproxidamente 100000 entradas. É interessante notar que há pouca diferença entre o seu desempenho de pior caso e de caso médio, ambos $O(n^2)$, como podemos verificar na figura 3.1. Por outro lado, seu melhor caso é muito rápido devido ao teste para ver se o vetor já está ordenado, teste este que em uma única passagem, e portanto $n-1$ comparações, percebe se o vetor já está ordenado.
Podemos verificar assim que o bubblesort não seria recomendado para a ordenação de sequências com muitos elementos, uma vez que rapidamente ele atinge tempos impraticáveis. Sua vantagem é a facilidade de implementação e pouco necessidade de espaço extra.

3.2 InsertionSort

Apresentou o segundo pior tempo, mas no entanto relativamente melhor que o bubblesort, chegando a ter um performance média 4 vezes melhor que a do bubblesort, para o mesmo tamanho de entrada. O desempenho do caso médio, apresentou-se cerca de 2 vezes melhor que a do pior caso para praticamente qualquer tamanho de entrada. Este fato verificado

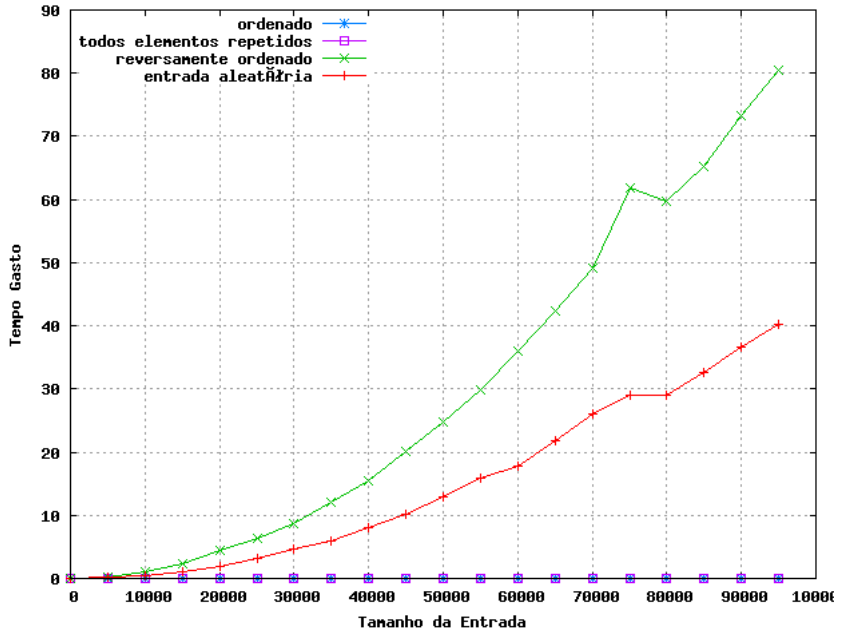

Figura 3.2: Gráficos do tempo por tamanho de entrada do algoritmo Insertionsort: entrada crescente, decrescente, igual e randômica

experimentalmente é provado por Sedgewick, anunciado como propriedade 6.2 em [3].
Tal como o bubblesort, em uma passegem e $n-1$ comparações e trocas (no loop interno ele compara e não entra, e troca o elemento por ele mesmo) o algoritmo termina, para o caso de uma sequências já ordenada.

3.3 ShellSort

O Shellsort é um algoritmo que na prática apresenta bons resultados, o grande problema é provar estes resultados. Ainda não existe uma análise matemática completa e satisfatória que apresente um limite inferior para a performance do Shellsort, alguns autores colocam os casos médios em torno de $O(n^1.25)$, $O(n^1.3)$ ou $O(n(logn)^2)$, mas não podemos dizer exatamente qual é o comportamento que melhor representa o Shellsort.
A primeira sequência usada `inc[16] = 412771, 165103, 66041, 26417, 10567, 4231, 1693, 673, 269, 107, 43, 17, 7, 3, 1` é uma sequência citada como tendo bons resultados experimentais. E a segunda, `inc[16] = 1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1` foi proposta em um artigo por Sedgewick em 1996. Iremos designar as respectivas implementações destas sequência no Shellsort de versão 1 e versão 2 do Shellsort
Ambas sequências de incrementos apresentaram bons resultados, apresentando melhores

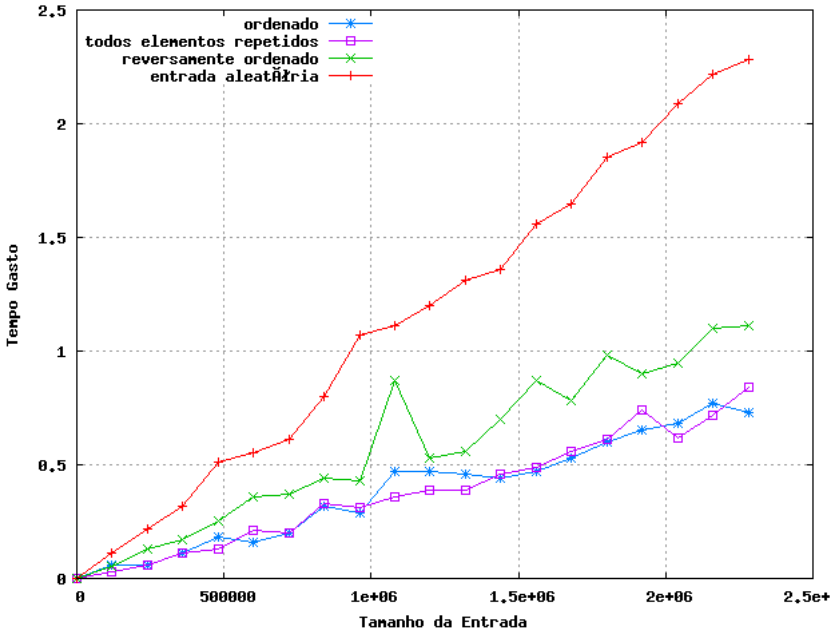

Figura 3.3: Gráficos do tempo por tamanho de entrada do algoritmo Shellsort, usando sequência de incrementos com boa performance experimental: entrada crescente, decrescente, igual e randômica

resultados, em alguns casos do quicksort (versão aleatorizada), principalmente tratando das sequência reversamente ordenada e sequência ordenada. Ambos ficaram abaixo de 1s, com uma sequência de entrade de 2000000 registros, enquanto o Quicksort com pivô aleatório fica um pouco acima de 1s, como podemos verificas nas figuras 3.3, 3.4 e 3.7. Como podemos verificar na tabela 3.1, o Shellsort (versão 1) apresentou performance em média 36% melhor que o quicksort com pivô aleatório no caso ordenado e 9% no caso reversamente ordenado, já no caso médio o quicksort mostrou-se em média 44.5% mais eficiente que o Shellsort.

A versão 2 apresentou melhor resultado que a versão 1 do Shellsort no caso reverso, em média 8,1% melhor, enquanto que no caso ordenado a diferença foi desprezível, cerca de 0,6%. No caso médio a versão 1 ficou com tempo médio 11,6% melhor que a versão 2.

3.4 MergeSort

Tal como era esperado, o comportamente do Mergesort para as instâncias, ordenadas, reversas, iguais e aleatórias foi praticamente o mesmo (figura 3.5).

Apesar de ser um algoritmo $O(nlogn)$, o seu loop interno tem mais instruções que o do quicksort, o que leva a uma performance de mesma ordem, mas ligeiramente inferior (em

	QuickSort			ShellSort			Differença (percentual)		
N	Ord	Rev	Rand	Ord	Rev	Rand	Ord	Rev	Rand
10	0.0	0.0	0.0	0.0	0.0	0.0	0.00%	0.00%	0.00%
120010	0.08	0.05	0.07	0.06	0.05	0.11	-25.00%	0.00%	57.14%
240010	0.12	0.13	0.16	0.06	0.13	0.22	-50.00%	0.00%	37.50%
360010	0.19	0.19	0.26	0.11	0.17	0.32	-42.11%	-10.53%	23.08%
480010	0.26	0.3	0.33	0.18	0.25	0.51	-30.77%	-16.67%	54.55%
600010	0.31	0.42	0.38	0.16	0.36	0.55	-48.39%	-14.29%	44.74%
720010	0.38	0.36	0.47	0.2	0.37	0.61	-47.37%	2.78%	29.79%
840010	0.37	0.4	0.58	0.32	0.44	0.8	-13.51%	10.00%	37.93%
960010	0.48	0.48	0.61	0.29	0.43	1.07	-39.58%	-10.42%	75.41%
1080010	0.58	0.52	0.71	0.47	0.87	1.11	-18.97%	67.31%	56.34%
1200010	0.66	0.68	0.79	0.47	0.53	1.2	-28.79%	-22.06%	51.90%
1320010	0.72	0.76	0.94	0.46	0.56	1.31	-36.11%	-26.32%	39.36%
1440010	0.92	0.94	0.92	0.44	0.7	1.36	-52.17%	-25.53%	47.83%
1560010	0.81	1.01	1.02	0.47	0.87	1.56	-41.98%	-13.86%	52.94%
1680010	0.89	0.96	1.16	0.53	0.78	1.65	-40.45%	-18.75%	42.24%
1800010	0.96	1.14	1.17	0.6	0.98	1.85	-37.50%	-14.04%	58.12%
1920010	1.83	1.09	1.3	0.65	0.9	1.92	-64.48%	-17.43%	47.69%
2040010	1.01	1.34	1.4	0.68	0.95	2.09	-32.67%	-29.10%	49.29%
2160010	1.12	1.34	1.47	0.77	1.1	2.22	-31.25%	-17.91%	51.02%
2280010	1.25	1.43	1.7	0.73	1.11	2.28	-41.60%	-22.38%	34.12%
3000010	1.66	1.58	2.07	0.98	1.72	3.25	-40.96%	8.86%	57.00%

Tabela 3.1: Comparação do Desempenho do Shellsort e QuickSort

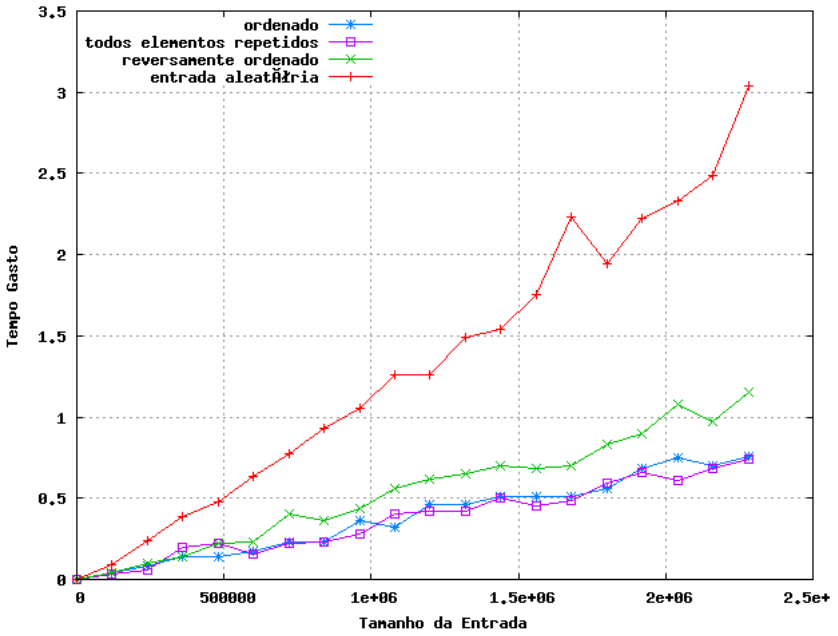

Figura 3.4: Gráficos do tempo por tamanho de entrada do algoritmo Shellsort, usando a sequência de incrementos de Incerpi-Sedgewick: entrada crescente, decrescente, igual e randômica

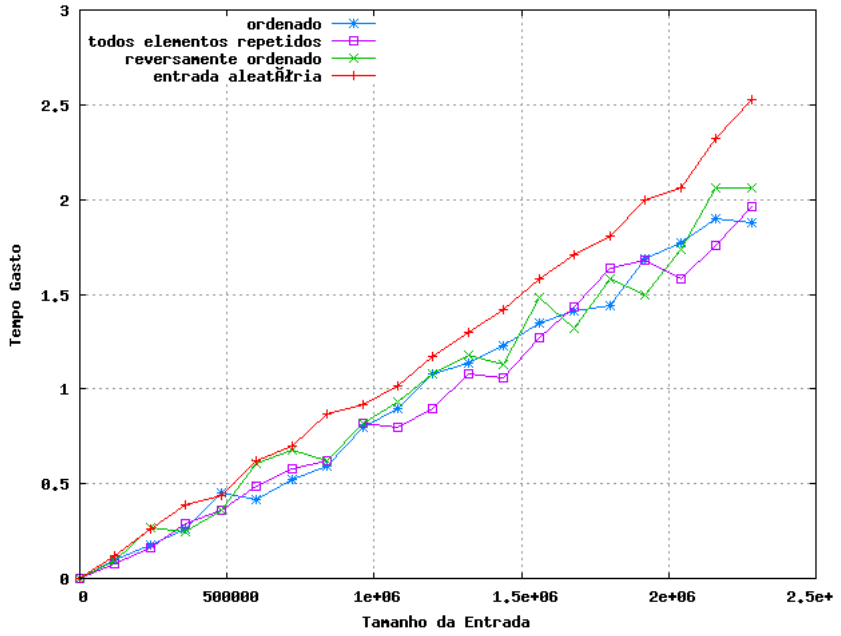

Figura 3.5: Gráficos do tempo por tamanho de entrada do algoritmo Mergesort: entrada crescente, decrescente, igual e randômica

média 50% menos rápido que o Quicksort para as instâncias testadas). Em relação ao heapsort, o mergesort perde em média por 6% no caso reverso, apresentando praticamente a mesma performance para o caso ordenada (0.5% pior), mas obtendo um tempo para o caso médio cerca de 25% melhor que o heapsort. Em relação ao shellsort (versão 1), ele é menos eficiente em todos casos, por cerca de 140%, 68% e 3,6% respectivamente para os casos ordenado, reverso e aleatório.

3.5 QuickSort

O quicksort fez jus ao seu nome e foi o algoritmo com menor tempo de processamento em praticamente todos as instâncias (exceto no caso patológico das instâncias com todos elementos repetidos). Mesmo nos casos que o algoritmo apresentou complexidade quadrática, apresentou tempos menores que o bubblesort e o insertionsort. Tudo isto se deve em parte a relativa simplicidade de projeto do loop interno do quicksort (no algoritmo de particionamento), que utiliza somente comando de incremento de inteiros e comparação com um valor fixo.

Duas versões do quicksort foram implementadas, cada uma usando uma estratégia diferente para a escolha do pivô no algoritmo de separação. A versão 1 utiliza o elemento médio da sequência como pivô, enquanto a versão 2 utiliza um elemento aleatório. Para a

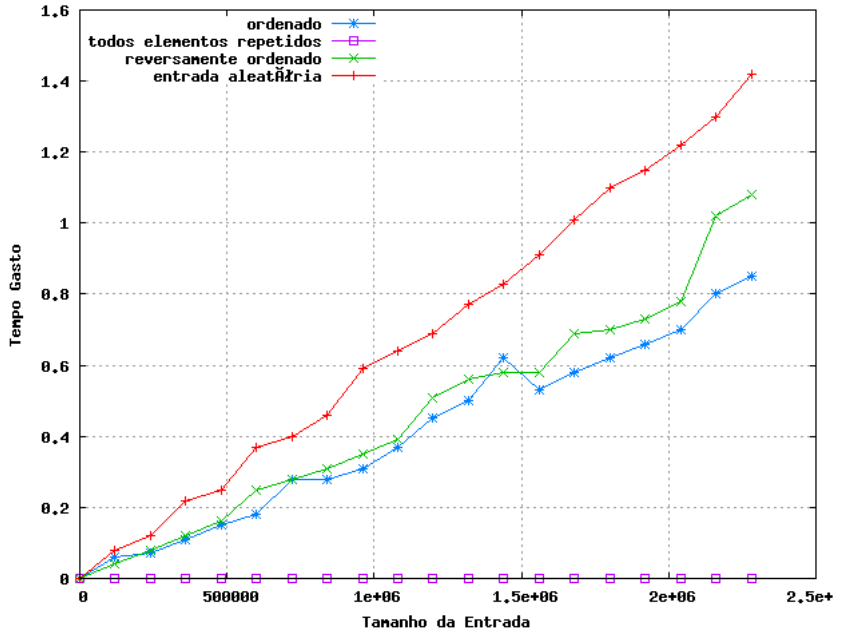

Figura 3.6: Gráficos do tempo por tamanho de entrada do algoritmo Quicksort, com pivo no elemento médio, para entradas da ordem de 1000000: entrada crescente, decrescente e randômica

versão 1, os casos ordenados e reversamente ordenados são os melhores casos, uma vez que o elemento médio corresponde à mediana do conjunto, fazendo desta forma uma divisão exata da sequência em duas partições do mesmo tamanho, resultando na complexidade de $O(nlogn)$. No entanto, a boa performance desta versão não se limitou ao casos de vetor ordenado e reverso, mas também ao caso aleatório, isto pode ser justificado em parte, porque a medida que as partições ficam menores e//ou as subsequências estejam parcialmente ordenadas, ele vai tender a ter performance ótima (uma vez que para estes pequenas subsequências tenderão a serem particionadas equalitariamente).

Em média os tempos da versão 1 foram melhores que os da versão 2, no caso ordenado a versão 2 gastou 55,5% de tempo a mais que a versão 1, no caso reverso 6,4% e no caso médio aleatório, gastou 3,5% de tempo a mais que a versão 1. Podemos verificar nas figuras 3.6 e 3.7 que o efeito da escolha aleatória do pivô é aproximar os casos ordenados e reverso do caso aleatório. Em relação ao SH1 o QK1 apresenta no caso médio, desempenho 37% melhor, e em relação ao MS, o QK1 apresenta tempo médio 39,6% melhor.

No pior caso, o QK1 e QK2 apresentam comportamentos muito parecidos, apresentando claramente a complexidade quadrática e gastando cerca de 35s para ordenar um vetor de cerca de 100000 elementos repetidos (figuras 3.6 e 3.7) .

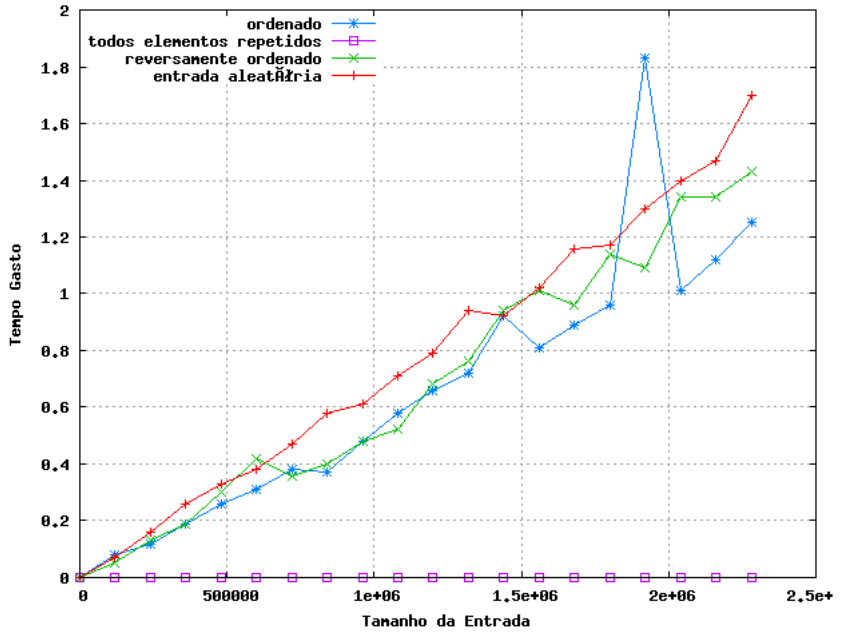

Figura 3.7: Gráficos do tempo por tamanho de entrada do algoritmo Quicksort, com pivo randômico, para entradas da ordem de 1000000: entrada crescente, decrescente e randômica

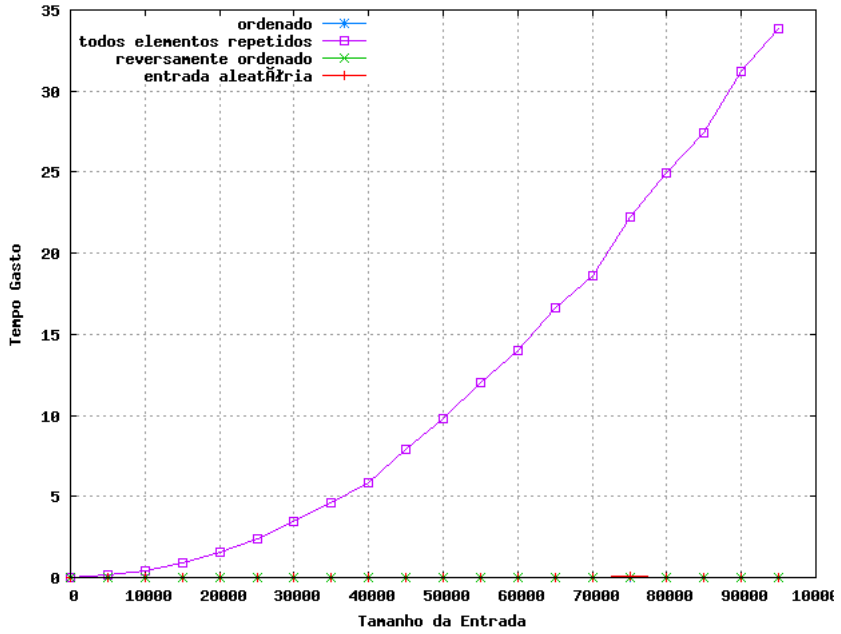

Figura 3.8: Gráficos do pior caso do Quicksort (todos elementos iguais), com pivoteamento no elemento médio

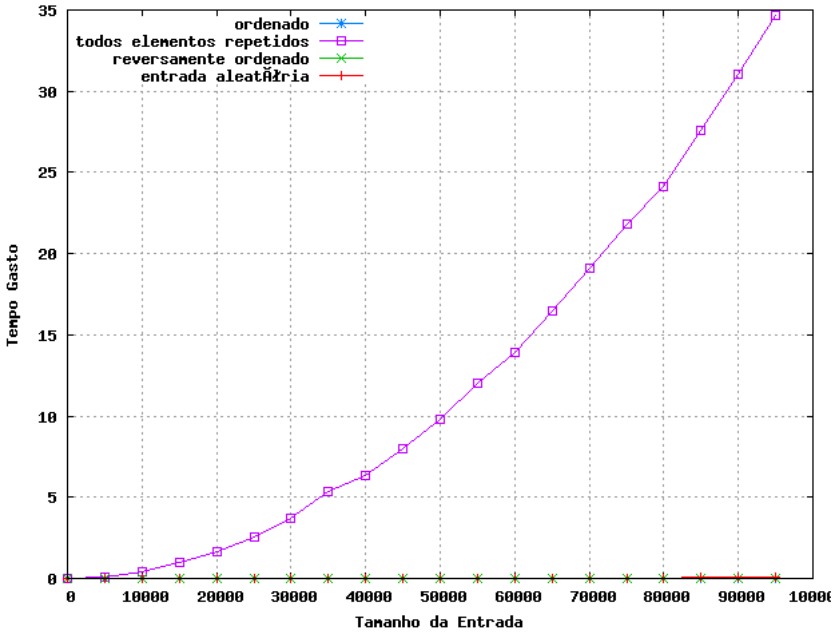

Figura 3.9: Gráficos do pior caso do Quicksort (todos elementos iguais), com pivoteamento randômico

3.6 HeapSort

O Heapsort apresentou o pior desempenho médio dentre os algoritmos com complexidade $O(nlogn)$, gastando cerca de 3,5s para ordenar uma sequência de 2000000 de registros inteiros. Nos casos do vetor ordenado e reverso ele mostrou o mesmo tempo que o MS. Em relação ao QK1 ele apresentou em médio tempos maiores em 130,5%, 106,6% e 130,2%, respectivamente para os casos ordenado, reverso e aleatório. Para um vetor com todos elementos repetidos o HS apresentou um dos melhores resultados, gastando menos de 0.5 segundos (figura 3.10).

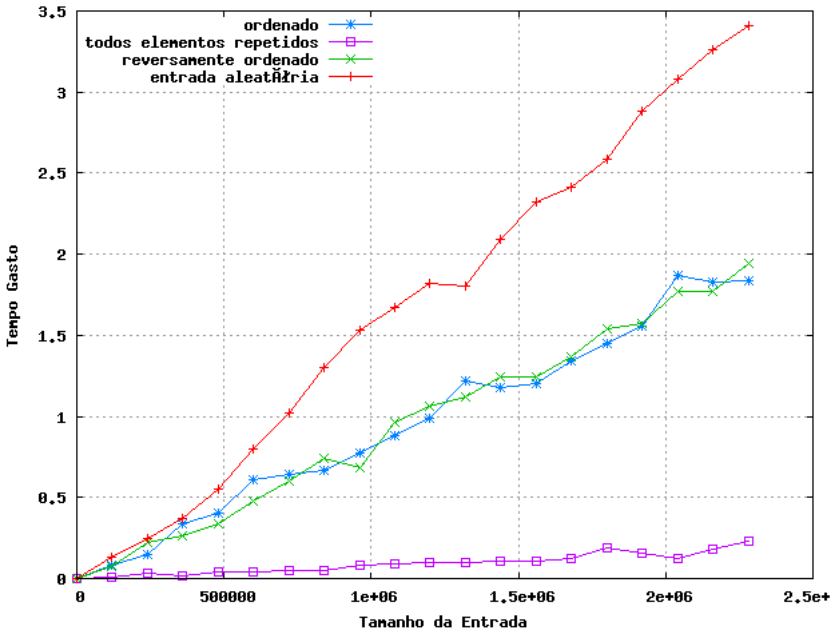

Figura 3.10: Gráficos do tempo por tamanho de entrada do algoritmo Heapsort, para entradas da ordem de 1000000: entrada crescente, decrescente, igual e randômica

Capítulo 4

Conclusões

Neste trabalho foi possível verificar experimentalmente a complexidade computacional de 6 algoritmos de ordenação: InsertionSort, Bubblesort, Mergesort, Quicksort e Heapsort. O comportamente experimental destes algoritmos foi de acordo com o previsto teoricamente. Foi possível, dentre os algoritmos da mesma classe de complexidade, verificar a eficiência relativa destes algoritmos, verificando quais são mais rápidos para que situações. O Quicksort apresentou-se como o algoritmo mais rápido no caso médio (aleatório) e reverso, e o Shellsort apresentou bons resultados também, muito próximo do quicksort nestes casos. Nos casos de instâncias ordenadas, o insertionsort e o bubblesort foram os melhores, por serem $O(n)$ para estes casos.

Além de comparações entre algoritmos diferentes, foi possível estudar o comportamento de variações de um mesmo algoritmo, o que foi feito no caso do quicksort e shellsort. No primeiro caso, comparou-se o desempenho de duas estratégias distintas de escolha de pivô na fase de particionamento do algoritmo: a primeira versão (QK1) usou o elemento médio e a segunda versão (QK2) usou um pivô aleatório. O QK1 apresentou melhores resultados que o QK2, principalmente nos casos de instãncias ordenadas e reversas. O efeito da segunda estratégia foi aproximar o desempenho dos casos extremos para o caso médio. Para o shellsort, foi feita uma comparação entre duas sequência de incrementos distintas, uma experimental com bons resultados práticos (SH1) e outra proposta por Sedgewick (SH2). No caso direto a diferença de performance foi menos de 1%, enquanto que no caso reverso o SH2 teve uma performance 8% melhor que o SH1, já no caso médio o SH2 apresentou um tempo médio 11,6% pior que o SH1. A tabela 4.1 apresenta um comparativo entre alguns dos algoritmos implementados, apresentando em médio, o percentual a mais ou a menos de tempo de um algoritmo em relação a outro.

Uma possível continuidade para este trabalho seria ampliar o leque de categorias de instâncias a serem testadas, incluindo instâncias aleatórias quase-ordenados, instâncias com

Média de tempo de	em relação a	Ord	Rev	Rand
QK2	QK1	55.5%	46.1%	13.6%
MS	QK2	47.8%	50.0%	48.9%
MS	QK1	127.7%	118.3%	69.1%
MS	HS	0.55%	6.4%	-24.4%
MS	SH1	140%	68.4%	3.58%
SH2	QK1	-4.9%	17.9%	83.1%
SH1	QK1	-3.85%	31.6%	64%
SH2	SH1	0.6%	-8.05%	11.6%
HS	QK1	130.5%	105.6%	130.2%
HS	QK2	49.7%	41.5%	102.9%
HS	SH1	143.8%	58.9%	39.7%

Tabela 4.1: Tabela comparativa entre diversos algoritmos de ordenação

distribuições gaussianas, etc. Este tipo de análise ampliaria o conhecimento do comportamento estatístico de alguns algoritmos sobre tipos específicos de sequências de entrada.

Referências

[1] Leiserson C. E. Rivest R. L. e Stein C. Cormen, T. H. *Introduction To Algorithms*. MIT Press, Cambridge, 2nd edition, 2001.

[2] R. Sedgewick. *Quicksort*. Garland Publishing, London, 1980.

[3] R. Sedgewick. *Algorithms In C++: Parts 1-4*. Addison-Wesley, Princeton, 1998.

[4] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, London, 2nd edition, 2008.