Universidade Estadual de Campinas
Instituto de Matemática, Estatística e Computação Científica
Departamento de Matemática Aplicada

ALGORITMOS DE ORDENAÇÃO NA OTIMIZAÇÃO DO VALOR ORDENADO

André Luis Trevisan[1]
Mestrado em Matemática Aplicada

Orientador: Prof. Dr. José Mario Martínez

Co-orientadora: Prof.a Dr.a Sandra Augusta Santos

Campinas, 25 de fevereiro de 2008

[1]Este trabalho teve o suporte financeiro da Fapesp, no período entre março/2006 e fevereiro/2007 (Processo 05/57841-0).

ALGORITMOS DE ORDENAÇÃO NA OTIMIZAÇÃO DO VALOR ORDENADO

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por André Luis Trevisan e aprovada pela comissão julgadora.

Campinas, 25 de fevereiro de 2008.

Prof. Dr. José Mario Martínez

Orientador

Prof.a Dr.a. Sandra Augusta Santos

Co-Orientadora

Banca Examinadora

1. Prof. Dr. José Mario Martínez
2. Prof. Dr. Ricardo Biloti
3. Prof. Dr. Paulo José Silva e Silva

Dissertação apresentada ao Instituto de Matemática, Estatística e Computação Científica, UNICAMP, como requisito parcial para obtenção do Título de MESTRE em Matemática Aplicada.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues

Trevisan, André Luis
T729a Algoritmos de ordenação na otimização do valor ordenado / André Luis Trevisan -- Campinas, [S.P. :s.n.], 2008.

Orientadores : José Mario Martínez ; Sandra Augusta Santos
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1. Otimização matemática. 2. Algoritmos de computador. 3. Programação não-linear. . I. Martínez, José Mario. II. Santos, Sandra Augusta. . III. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. IV. Título.

Título em inglês: Sorting algorithms in order value optimization.

Palavras-chave em inglês (Keywords): 1. Mathematical optimization. 2. Computer algorithms. 3. Nonlinear programming.

Área de concentração: Otimização

Titulação: Mestre em Matemática Aplicada

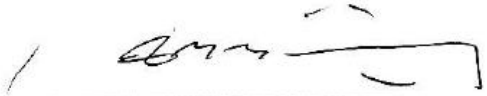
Banca examinadora: Prof. Dr. José Mario Martínez (IMECC-UNICAMP)
Prof. Dr. Ricardo Biloti (IMECC-UNICAMP)
Prof. Dr. Paulo José Silva e Silva (IME-USP)

Data da defesa: 25/02/2008

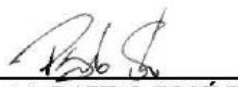
Programa de pós-graduação: Mestrado em Matemática Aplicada

Dissertação de Mestrado defendida em 25 de fevereiro de 2008 e aprovada

Pela Banca Examinadora composta pelos Profs. Drs.



Prof. (a). Dr (a). JOSÉ MARIO MARTÍNEZ PEREZ



Prof. (a). Dr (a). PAULO JOSÉ DA SILVA E SILVA

Prof. (a). Dr (a). RICARDO CAETANO AZEVEDO BILOTI

À minha família

Agradecimentos

Agradeço, antes de mais nada, a Deus, por permitir que pudesse alcançar mais essa conquista.

Agradeço a todos os professores que passaram por minha vida, alguns deles hoje colegas de trabalho, por serem responsáveis pelo conhecimento que pude aqui demonstrar. Ao professor Martínez, pela proposta de trabalho e pela orientação segura. À professora Sandra, com o qual trabalho junto desde o primeiro ano de graduação. Uma pessoa amiga e totalmente profissional, que ao longo destes anos me ajudou em tudo o que eu precisei.

Agradeço aos meus pais e à minha família, que acreditaram no meu trabalho e sempre me incentivaram e apoiaram a continuar meus estudos. À minha querida Ana Carolina, por estar sempre ao meu lado e por compreender todas as vezes que não pude estar presente ou não pude lhe dar muita atenção.

Agradeço a todos meus amigos, tantos os feitos em Campinas ao longo desses seis anos, como os de Rolândia, pela ajuda que recebi em várias ocasiões. Em especial aos amigos da "República Paula Tejano", com quem convivi grande parte desse tempo.

À Fapesp, pelo apoio financeiro.

Por fim, agradeço a todos que direta ou indiretamente contribuiram para a realização deste trabalho.

Resumo

Este trabalho aborda o problema de Otimização do Valor Ordenado e tem por objetivo investigar o papel desempenhado pelas estratégias de ordenação, com vistas a aplicações na área de avaliação de risco. São apresentados alguns métodos clássicos de ordenação, bem como algoritmos e adaptações para determinar o elemento que ocupa uma dada posição num vetor. Por meio de experimentos numéricos, foi possível comparar o desempenho desses métodos e verificar qual mostrou-se mais eficiente para o problema de determinação de uma dada estatística de ordem.

Palavras-chave: *Otimização do valor ordenado; estratégias de ordenação; estatísticas de ordem; experimentos numéricos.*

Abstract

This paper deals with the Order-Value Optimization problem and it aims to investigate the role played for the sorting strategies, with sights the applications in the area of risk evaluation. Some classic methods for sorting are presented, as well as algorithms and adaptations to determine the element that occupies a given position in a vector. Throughout numerical experiments, it was possible to compare the performance of these methods and to verify which of them revealed more efficient towards the problem of determination of a given order statistics.

Keywords: *Order-Value Optimization; sorting strategies; order statistics; numerical experiments.*

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Risco pode se definido como uma medida da incerteza associada aos retornos esperados de investimentos. O risco está presente na rotina de qualquer investimento (financeiro ou não). Risco não é um conceito novo. Em finanças, a Teoria Moderna das Carteiras, que se originou do trabalho pioneiro de Markowitz, já existe por mais de quatro décadas. Esta teoria está baseada nos conceitos de retorno e risco. Risco assumiu sua justa posição de destaque somente recentemente, seguindo-se a acontecimentos tais como colapsos, socorros emergenciais, disputas judiciais, entre outros [16].

Se uma carteira, cujo valor de mercado é R$ 100.000,00, apresentasse, por exemplo, uma probabilidade de ocorrência de retornos abaixo de -2% igual a 5%, então poderia se fazer a seguinte afirmação acerca do seu risco: *a carteira tem 5% de probabilidade de gerar uma perda financeira maior ou igual a R$ 2.000,00.* Os modelos de análise de risco são, a grosso modo, um conjunto de técnicas que tem por objetivo gerar este tipo de informação.

A sentença probabilísta enunciada no parágrafo anterior é a essência do conceito desenvolvido pelo banco JP Morgan, o *Value-at-Risk (VaR).* O valor de R$ 2.000 é o VaR daquela carteira de ativos a 5% de probabilidade ou, em linguagem estatística, a perda será maior ou igual a R$ 2.000 ao nível de significância de 5%. Assim, pode-se dizer que o VaR é uma medida que expressa, de forma probabilísta, as variações adversas esperadas de uma carteira de ativos [1, 8, 13, 16, 20, 21, 24].

A noção de *risco de um portfólio* está associada ao fato de seu retorno em um dado período de tempo não ser conhecido de antemão. Ao contrário, existe um conjunto de retornos possíveis. Quando tomamos um universo de ativos, podemos considerar um conjunto de cenários possíveis para os preços futuros desses ativos. Para cada decisão tomada, dentro de um espaço de decisões possíveis, existe um risco de perda ligado a esta decisão, associado a um determinado cenário.

Assim, dadas as perdas associadas a um conjunto de decisões, propõe-se o problema de minimizar a

perda que ocupa a $k-$ésima posição em uma sequência ordenada, dentro de um certo nível de significância. Esse tipo de problema, conhecido como Problema de Otimização de Valor Ordenado [2, 3, 4] envolve a determinação do elemento que ocupa a $k-$ésima posição em uma sequência ordenada, ou, sem outras palavras, o cálculo da $k-$ésima estatística de ordem de um conjunto de perdas possíveis.

A busca de estratégias eficazes para a determinação da $k-$ésima estatística de ordem motivou o o estudo teórico dos métodos de ordenação clássicos [7, 11, 12, 14, 23, 29], eventuais adaptações destes métodos [9, 15, 17, 22, 25, 26], bem como algoritmos que apareceram mais recentemente [18, 19, 5]. Paralelamente ao estudo desses algoritmos, trabalhamos com a sua implementação em linguagem `Fortran` [10].

No Capítulo 2 apresentamos o problema a ser tratado, bem como alguns conceitos e terminologias envolvidos. Enquanto no Capítulo 3 discutimos alguns dos métodos clássicos de ordenação, o Capítulo 4 foca o problema de determinação de estatística de ordem e os algoritmos adaptados para esta finalidade. Por fim, no Capítulo 5 são apresentados resultados de experimentos numéricos, que visam determinar maneiras eficazes de se calcular a $k-$ésima estatística de ordem de um conjunto de números representativos do problema em questão. Algumas considerações acerca dos resultados obtidos, bem sugestões para trabalhos futuros, são apresentados no Capítulo 6.

Capítulo 2

O problema

Inicialmente apresentaremos alguns conceitos e também alguns termos que serão utilizados durante a formulação do problema a ser tratado.

2.1 Alguns conceitos em Economia Matemática

Chamamos de *ativos reais* a todos os componentes da cadeia produtiva de uma economia, como, por exemplo, terras, maquinarias, prédios, fábricas, produtos agrícolas, corporações, entre outros. Já os papéis negociáveis são um exemplo de *ativos financeiros*. Apesar de não representarem as riquezas de uma sociedade, eles participam de maneira indireta do processo produtivo, pois tornam possível operacionalizar os processos de transferência de recursos entre participantes da economia.

O *mercado financeiro* é o ambiente composto por todos os ativos finaceiros, onde estes são negociados e, por conseguinte, ganham valor. Podemos identificar três tipos de mercado financeiro:

- *Mercado Direto*: onde os compradores e vendedores se procuram uns aos outros, sem intermediários;
- *Mercado Intermediado*: onde existem agentes intermediadores que encarregam-se de juntar compradores com vendedores;
- *Mercado de Leilão*: é o mercado mais integrado, onde todos os agentes se juntam e procuram e/ou oferecem seus ativos financeiros. Devido a esta convergência, os preços são estabelecidos em função da oferta e da demanda desses ativos. O melhor exemplo é a Bolsa de Valores.

Um *portfólio de aplicações financeiras* é uma coleção de investimentos mantida por uma instituição ou indivíduo. Manter um portfólio de aplicações faz parte de uma estratégia de diversificação, com o

intuito de diminuir riscos. Assim, dado um ativo qualquer, podemos, entre outros, utilizar o histórico de preços e/ou retornos desse ativo para gerar *cenários de preços futuros*.

O desenvolvimento de modelos de otimização de portfólio tem origem na área econômico-financeira; tais modelos são utilizados para auxiliar na determinação da carteira de ativos financeiros que apresente a melhor relação risco $\times$ retorno sob o ponto de vista de um investidor. A principal motivação para o desenvolvimento destes modelos se relaciona à redução do risco a que o investidor está exposto, através da diversificação ou balanceamento da carteira.

2.2 Os conceitos de VaR e CVaR

Formalmente, o *Value-at-Risk* de um portfólio trata da maior perda, esperada para uma determinada carteira, com um determinado nível de significância, dentro de um horizonte de tempo fixado. Mollica [16] apresenta várias técnicas de cálculo do VaR. Dentre elas, provavelmente o *método de Simulação Histórica* é a abordagem mais direta e intuitiva para o cálculo do *Value-at-Risk* de uma carteira de ativos. Tal método será descrito na seção 2.3.

Embora o VaR seja uma medida de risco largamente aceita e utilizada, seu uso tem sofrido críticas por parte da comunidade acadêmica pelo fato de ser uma medida de risco que não fornece nenhuma informação a respeito das perdas que o excedem, as quais podem ser significativamente grandes. Rockafellar & Uryasev [21] sugerem uma medida de risco denominada *Conditional Value-at-Risk* (CVaR). Tomando-se uma sequência ordenada de perdas possíveis, e escolhido um certo nível de confiança, o CVaR é definido como a média dos valores superiores ao VaR.

Adotar o CVaR como medida de risco de um portfólio se caracteriza como uma estratégia de gerenciamento de riscos mais conservadora do que o VaR. Isto porque o CVaR de um portfólio a um nível de confiança nunca será menor do que o respectivo VaR.

2.3 O método de Simulação Histórica

Segundo Chaia *et al.* [6], " A metodologia de 'Simulação Histórica' consiste na aplicação de cada uma das variação passadas (...) sobre seu valor atual", ou seja, aplicar os pesos atuais de cada ativo na carteira às respectivas séries históricas dos retornos destes ativos.

Este método tem uma série de atrativos. Em primeiro lugar, trata-se de uma técnica não paramétrica, o que implica dizer que não é necessária a estimação de nenhum parâmetro, como volatilidades e correlações, para obtenção do VaR. Com isso, evitam-se problemas de modelagem e erros de estimações destes parâmetros. Outra vantagem do método é que ele dispensa qualquer hipótese *a priori* sobre a distribuição dos retornos dos ativos. Por último, os recursos computacionais atualmente disponíveis tornam

extremamente simples sua implementação.

Infelizmente, o método apresenta também algumas limitações. A hipótese que está implícita nesta técnica é que a distribuição empírica dos retornos é capaz de refletir a verdadeira distribuição dos retornos. Desta forma, o VaR estimado é extremamente sensível à janela utilizada, inclusão ou não de períodos onde ocorreram grandes *outliers* ou de longos períodos de pouca oscilação nos retornos - todos esses fenômenos produzem grandes diferenças nos resultados. Este efeito é mais pronunciado quando o objetivo é estimar o VaR para níveis de significância muito baixos (menores que 1%).

Outro problema é a falta de adaptabilidade das estimativas, ou seja, incapacidade de perceber rapidamente mudanças estruturais no ambiente financeiro, como mudanças de regimes cambiais ou aumento de instabilidade decorrente da desregulamentação de um determinado mercado. Isto acontece porque o método aplica pesos idênticos a todas as observações da série histórica. Assim, eventos ocorridos num passado distante continuam tendo o mesmo peso nas estimativas que os eventos mais recentes.

Por último, o método de simulação histórica é bastante eficaz para estimação da distribuição dos retornos num intervalo concentrado ao redor da média, região na qual a distribuição empírica tende a ser bastante densa. No entanto, devido à característica discreta dos retornos extremos, o método produz uma estimativa bastante pobre das caudas, região de interesse para o cálculo do VaR, o que tende a gerar alta variabilidade nas estimativas do VaR.

2.4 Nosso problema

Consideremos inicialmente um universo de w ativos e uma decisão x a ser tomada, $x \in \Omega$, onde Ω é o espaço de possíveis decisões. Suponhamos que existam n cenários, cada um deles correspondendo a uma $w-$upla de preços futuros. Sem perda de generalidade, suponhamos que todos os cenários são igualmente prováveis.

Seja $f_i(x)$ a perda provocada pela decisão x, quando ocorre o cenário i. Para cada decisão x, podemos *ordenar* as perdas da menor para a maior:

$$f_{i_{1(x)}}(x) \leq f_{i_{2(x)}}(x) \leq \cdots \leq f_{i_{n(x)}}(x). \tag{2.4.1}$$

Tomando-se um valor $k \in \{1, 2, \cdots, n\}$, temos um nível de confiança dado por $\alpha = \frac{k}{n}$; podemos então definir os conceitos de VaR(x) e CVaR(x), em função dessas escolhas, do seguinte modo:

$$\text{VaR}_\alpha(x) = f_{i_{k(x)}}(x) \tag{2.4.2}$$

$$\text{CVaR}_\alpha(x) = \frac{1}{n-k}\left(\sum_{j=k+1}^{n} f_{i_{j(x)}}(x)\right). \tag{2.4.3}$$

Em outras palavras, podemos caracterizar o $\text{VaR}_\alpha(x)$ como o valor de $f_i(x)$ que ocupa o $k-$ésimo lugar na ordenação (2.4.1), enquanto $\text{CVaR}_\alpha(x)$ é definida como a média dos $(n-k)$ valores superiores ao $\text{VaR}_\alpha(x)$.

A determinação do elemento que ocupa a $k-$ésima posição em uma sequência ordenada é parte de um problema de otimização mais geral, apresentado por Andreani, Dunder e Martínez [2], e conhecido como *Order-Value Optimization (OVO)*, ou *Problema de Otimização do Valor Ordenado*. Trata-se de uma generalização do problema clássico *Minimax*; entretanto, ao invés do máximo de um conjunto de funções, estamos interessados em minimizar o valor da função que ocupa a $k-$ésima posição em uma sequência ordenada.

Assim, dadas n funções $f_1, \cdots, f_n$, definidas em um domínio $\Omega \subset \Re^n$, e um inteiro $k \in \{1, \cdots, n\}$, definimos a $k-$ésima função f de valor ordenado do seguinte modo:

$$f(x) = f_{i_{k(x)}}(x),$$

para todo $x \in \Omega$, onde

$$f_{i_{1(x)}}(x) \leq f_{i_{2(x)}}(x) \leq \cdots \leq f_{i_{k(x)}}(x) \leq \cdots \leq f_{i_{n(x)}}(x).$$

Se $k = 1$, $f(x) = \min\{f_1(x), \cdots, f_n(x)\}$. Por outro lado, se $k = n$, temos então $f(x) = \max\{f_1(x), \cdots, f_n(x)\}$. Em qualquer outro caso, $f(x)$ é a $k-$ésima *estatística de ordem* do conjunto $\{f_1(x), \cdots, f_n(x)\}$.

O problema OVO consiste na minimização da $k-$ésima função de valor ordenado, dentro do domínio em questão:

$$\begin{array}{ll} \min & f(x) \\ s.a. & x \in \Omega. \end{array} \tag{2.4.4}$$

Andreani, Dunder e Martínez [2] demonstram que f é uma função contínua em Ω, e desenvolvem um algoritmo que generaliza o método de máxima descida, para resolver o problema (2.4.4). Em [3] são apresentadas as condições de otimalidade, bem como uma reformulação não-linear para este problema.

Em linhas gerais, supõe-se que $f_{i_{k(x)}}(x)$ é efetivamente calculada em cada iteração dos algoritmos empregados, por meio da ordenação apresentada em (2.4.1) . Se o tamanho n é moderado, este trabalho é admissível. Entretanto, à medida que n cresce, o tempo computacional dos algoritmos dedicados a resolver o Problema de Otimização do Valor Ordenado é quase que exclusivamente dedicado a essas ordenações de números.

Sabe-se que, em alguns problemas de risco, é fundamental gerar, mediante simulações apropriadas, centenas de milhares de cenários. Neste caso, "abreviar"o tempo computacional das ordenações é essencial para otimizar com relativa eficiência. Nosso trabalho foca então o estudo dos métodos mais eficientes de ordenação, em especial a determinação do elemento que ocupa a $k-$ésima posição no vetor ordenado.

Capítulo 3

Ordenação

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. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade se acessar seus dados de modo mais eficiente.

Os *métodos de ordenação* são classificados em dois grandes grupos. Se o arquivo a ser ordenado cabe todo na memória principal, então o método de ordenação é chamado de *ordenação interna*. Neste caso, o número de registros a ser ordenado é pequeno o bastante para caber em um vetor. Se o arquivo a ser ordenado não cabe na memória principal, e por isso, tem de ser armazenado em disco, então o método de ordenação é chamado de *ordenação externa*. A principal diferença entre eles é que, em um método de ordenação interna, qualquer registro pode ser imediatamente acessado, enquanto em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos. Trataremos neste trabalho apenas de algoritmos de ordenação interna.

3.1 A escolha do algoritmo

A análise de algoritmos envolve dois tipos de problemas distintos:

- *Análise de um algoritmo particular:* Qual é o custo de usar um dado algoritmo para resolver um problema específico? Características do algoritmo devem ser investigadas, geralmente uma análise do número de vezes que cada parte do algoritmo deve ser executada, seguida do estudo da quantidade de memória necessária.
- *Análise de uma classe de algoritmos:* Qual é o algoritmo de menor custo possível para resolver um problema particular? Toda uma família de algoritmos para resolver um problema específico é

investigada com o objetivo de identificar um que seja o melhor possível. Isto significa colocar limites para a complexidade computacional dos algoritmos pertencentes à classe.

3.1.1 Custo de utilização de um algoritmo

O custo de utilização de um algoritmo pode ser medido de várias maneiras, uma delas sendo a própria medição direta do tempo de execução. Esta, entretanto, dependerá, entre outros fatores, do compilador utilizado. Uma forma mais adequada de se medir o custo de utilização de um algoritmo é por meio do uso de um modelo matemático, baseado em um computador idealizado. O conjunto de operações a serem executadas deve ser especificado, bem como o custo associado com a execução de cada operação, considerando-se as mais significativas.

Para medir o custo de execução de um algoritmo, é comum definir uma função de custo ou *função de complexidade f*, onde $f(n)$ é a medida do tempo necessário para executar um algoritmo para uma instância de tamanho n. Neste caso, não se representa o tempo diretamente, mas o número de vezes que determinada operação, considerada relevante, é executada.

Costumam-se distinguir três cenários possíveis:

- *Melhor caso:* menor tempo de execução sobre todas as possíveis entradas de tamanho n;
- *Pior caso:* maior tempo de execução sobre todas as possíveis entradas de tamanho n;
- *Caso médio (ou esperado)*: Supõe-se uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n. É comum supor uma distribuição de probabilidades onde todas as entradas possíveis são igualmente prováveis.

3.2 Funções de complexidade

Avaliando os problemas por meio de suas *funções de complexidade*[1], e negligenciando as constantes de proporcionalidade, podemos classificar os problemas nas seguintes *classes*:

- *Complexidade constante:* $f(n) = O(1)$, ou seja, é independente do tamanho n.
- *Complexidade linear:* $f(n) = O(n)$. Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída.

[1] Uma função $g(n)$ é dita $O(f(n))$ se existem constantes $c \in \Re$ e $m \in N$ tais $g(n) \leq cf(n)$, para todo $n \geq m$.

- *Complexidade logarítmica:* $f(n) = O(\log n)$ ou $f(n) = O(n \log n)$. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores, resolvendo cada um deles independentemente, e depois juntando as soluções.
- *Complexidade quadrática:* $f(n) = O(n^2)$. Algoritmos dessa ordem aparecem quando os itens de entrada são processados aos pares, muitas vezes, em um laço dentro de outro.
- *Complexidade cúbica:* $f(n) = O(n^3)$. São úteis apenas para resolver problemas pequenos, como por exemplos as fatorações.
- *Complexidade exponencial:* $f(n) = O(2^n)$ e *complexidade fatorial:* $f(n) = O(n!)$. Em geral, não são úteis sob o ponto de vista prático. Eles ocorrem na resolução de alguns problemas quando se usa a força bruta.

A Tabela 3.1, extraída de [29], compara o tempo de execução de algoritmos com diferentes funções de custo, para diferentes valores do número n de dados de entrada.

Função custo	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s	0.00006 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s	0.0036 s
n^3	0.01 s	0.008 s	0.027 s	0.64 s	0.125 s	0.316 s
n^5	0.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13 min
2^n	0.001 s	1 s	17.9 min	12.7 dias	35.7 anos	366 séc.
3^n	0.059 s	58 min	6.5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Tabela 3.1: Comparação de várias funções de complexidade.

3.3 Algoritmos Clássicos de Ordenação

Conforme citado anteriormente os *algoritmos de ordenação* colocam os elementos de uma dada sequência em uma certa ordem. Entre os mais importantes, podemos citar os algoritmos elementares (*Ordenação por Seleção*, *Ordenação por Inserção* e `BubbleSort`), o `QuickSort` e o `HeapSort`, cujas caracterizações são apresentadas ao longo dessa seção. Assumiremos um vetor A com n elementos a ser ordenado.

3.3.1 Algoritmos Elementares

Em nossos primeiros "passos"na área de algoritmos de ordenação, estudamos alguns algoritmos elementares, apropriados para arquivos pequenos ou com algum tipo especial de estrutura. De um modo

geral, eles nos permitiram compreender melhor a terminologia e os mecanismos utilizados em algoritmos de ordenação, bem como na linguagem de programação utilizada . Além disso, em diversas aplicações, esses algoritmos elementares mostram-se mais eficientes que algoritmos mais sofisticados. Por fim, muitos deles podem ser usados como subrotinas no melhoramento de algoritmos de ordenação mais poderosos.

BubbleSort

O `BubbleSort` ou *Método de Ordenação da Bolha* é um algoritmo de ordenação popular. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem. Os elementos são como bolhas em um tanque de água: cada uma procura o seu próprio nível. A forma mais simples da ordenação bolha é mostrada na Figura 3.1.

```
BubbleSort(A)
01.          for i = n downto 2
02.               for j = 2 to i
03.                    if A[j-1] < A[j] then
04.                         A[j-1] ⇌ A[j]
05.               end
06.          end
```

Figura 3.1: Implementação de `BubbleSort`.

Neste processo, o maior elemento do vetor é encontrado no primeiro passo, sendo trocado com cada um dos elementos à direita, até encontrar sua posição final. Num segundo passo, o segundo maior elemento é colocado em sua posição final e assim sucessivamente.

Ordenação por Seleção

Um dos algoritmos mais simples de ordenação dos n elementos de um vetor A baseia-se no seguinte princípio:

1. Selecione o menor elemento do conjunto;
2. Troque este elemento com o primeiro elemento $A[1]$.

A seguir, repita as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$ elementos, até que reste apenas um elemento. A implementação deste processo é apresentada na Figura 3.2.

```
SelectionSort(A)
01.          for i = 1 to n − 1
02.               min ↩ i
03.               for j = i + 1 to n
04.                    if A[j] < A[min] then
05.                         min ↩ j
06.                    A[min] ⇌ A[i]
07.               end
08.          end
```

Figura 3.2: Implementação de `SelectionSort`.

Neste algoritmo, o ponteiro i percorre o vetor da esquerda para a direita, de modo que os elementos à esquerda do ponteiro estão em suas posições finais; assim, o vetor está totalmente ordenado quando o ponteiro encontra a extremidade direita do vetor.

Apesar do evidente processo de "força bruta" utilizado, este algoritmo de ordenação tem uma importante aplicação: como cada elemento do vetor é movido não mais que uma vez, ordenação por seleção é indicado para ordenar arquivos com registros longos e pequenas chaves.

Ordenação por Inserção

É também um algoritmo eficiente para ordenar um número pequeno de elementos. Funciona da maneira como as pessoas ordenam as cartas em um jogo de pôquer, por exemplo. Iniciamos com a mão esquerda vazia e as cartas viradas com a face para baixo da mesa. Em seguida, removemos uma carta de cada vez da mesa, inserindo-a na posição correta na mão esquerda, comparando-a com cada uma das cartas que já estão na mão, da direita para a esquerda. A implementação deste processo é mostrada na Figura 3.3.

Como acontece no algoritmo de *Ordenação por Seleção*, os elementos à esquerda do ponteiro i já estão ordenados durante o processo; entretanto, não estão em suas posições finais, já que precisam ser movidos para a direita, dando assim lugar aos elementos com valor menor. Aqui também o vetor está totalmente ordenado quando o ponteiro atinge sua extremidade direita.

Caracteríticas dos algoritmos elementares

Apresentamos a seguir alguns resultados relativos aos algoritmos elementares de ordenação, apresentados em detalhes por Sedgewick [23]:

```
InsertionSort(A)
01.         for i = 2 to n
02.               v = A[i]; j = i
03.               while A[j − 1] > v do
04.                     A[j] ↩ A[j − 1]
05.                     j = j − 1
06.               end
07.               A[j] ↩ v
08.         end
```

Figura 3.3: Implementação de `InsertionSort`.

- *Ordenação por Seleção* trabalha com cerca de $\frac{n^2}{2}$ comparações e n movimentações de elementos. É fácil ver que, para cada i variando de 1 até $n-1$, existem, no máximo, uma movimentação e $n-i$ comparações, de modo que há um total de $n-1$ movimentações de elementos e $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$ comparações;
- *Ordenação por Inserção* trabalha com cerca de $\frac{n^2}{4}$ comparações e $\frac{n^2}{8}$ movimentações de elementos;
- `BubbleSort` trabalha com cerca de $\frac{n^2}{2}$ comparações e $\frac{n^2}{2}$ movimentações de elementos.

3.3.2 QuickSort

`QuickSort` (C.A.R. Hore, 1960), ou algoritmo de ordenação rápida, é um algoritmo de ordenação cujo tempo de execução do pior caso é $O(n^2)$ sobre um vetor de entrada de n números. Apesar disto, é com frequência a melhor opção prática de ordenação, devido à sua notável eficiência na média: seu tempo de execução esperado é $O(n \log n)$. `QuickSort` se baseia no paradigma de dividir e conquistar[2]: ele divide o vetor em duas partes, que são ordenadas independentemente. A posição exata da partição também faz parte do procedimento. A estrutura recursiva do algoritmo é apresentada na Figura 3.4.

Os parâmetros l e r delimitam o sub-arquivo que será ordenado; a chamada `QuickSort`(A,1,n) ordena o vetor A todo. O ponto crucial do algoritmo `QuickSort` é o procedimento `Partition`, que rearranja o vetor de modo que:

- o elemento $A[i]$ está em sua posição final, para algum i;

[2]Algoritmos que baseiam-se na abordagem de *dividir e conquistar* desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os problemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

```
QuickSort(A,l,r)
01.          if r > l then
02.                i =Partition(A,l,r)
03.                QuickSort(A,l,i − 1)
04.                QuickSort(A,i + 1,r)
05.          end
```

Figura 3.4: Implementação de `QuickSort`.

- todos os elementos em $A[l], \cdots, A[i-1]$ são menores ou iguais a $A[i]$;
- todos os elementos em $A[i+1], \cdots, A[r]$ são maiores ou iguais a $A[i]$;

Inicialmente, escolhemos arbitrariamente algum elemento $A[i]$, nosso *pivô*, que estará em sua posição final. Uma escolha possível, é tomar $A[r]$ como pivô. A seguir, percorremos o vetor a partir da extremidade esquerda, até que um elemento maior que $A[i]$ seja encontrado, e percorremos o vetor a partir da extremidade direita, até encontrar um elemento menor que $A[i]$. Como ambos os elementos estão fora de suas posições finais, eles são trocados de lugar. Continuamos o procedimento até garantir que todos os elementos a esquerda do ponteiro esquerdo são menores que $A[i]$, e todos a direita do ponteiro direito são maiores que $A[i]$. Quando os ponteiros então se cruzam, o processo de particionamento está quase completo: basta apenas trocar o pivô com o elemento indicado pelo apontador esquerdo. O procedimento `Partition`[3], que reorganiza o subvetor $A[l, ..., r]$ localmente, é mostrado na Figura 3.5.

```
Partition(A,l,r)
01.          if r > l then
02.                v = A[r], i = l − 1, j = r
03.                repeat
04.                      repeat i = i + 1 until A[i] >= v
05.                      repeat j = j − 1 until A[j] <= v
06.                      A[i] ⇌ A[l]
07.                until j <= i
08.             return i
09.          end
```

Figura 3.5: Implementação do procedimento `Partition`.

[3]Tomamos aqui o elemento $A[r]$ como pivô.

QuickSort aleatório

O conhecimento de uma distribuição sobre as entradas pode nos ajudar a analisar o comportamento de um algoritmo no caso médio. Muitas vezes, entretanto, não temos tal conhecimento, e nenhuma análise de cado médio é possível. Muitos algoritmos aleatórios tornam a entrada aleatória permutando o vetor de entrada dado.

Podemos usar uma técnica de aleatoridade com o `QuickSort`: ao invés de sempre usar $A[r]$ como pivô, usaremos um elemento de $A[l, \ldots, r]$ escolhido ao acaso:

Essa modificação, em que fazemos a amostragem aleatória do intervalo $l, \ldots, r$ assegura que o elemento pivô $x = A[r]$ tem a mesma probabilidade de ser qualquer um dos $r - l + 1$ elementos do subvetor. Como o elemento pivô é escolhido ao acaso, esperamos que a divisão do vetor de entrada seja razoavelmente bem equilibrada na média.

As mudanças em `Partition` e `QuickSort` são pequenas: no novo procedimento de partição, simplesmente implementamos a troca do particionante, e o novo algoritmo `QuickSort` chama `RandPartition`, implementado no quadro da Figura 3.6, ao invés de `Partition`.

```
RandPartition(A,l,r)
01.        i =Random(l,r)
02.        A[l] ⇌ A[i]
03.        return Partition(A,l,r)
```

Figura 3.6: Implementação do procedimento `RandPartition`.

3.3.3 HeapSort

O algoritmo `HeapSort` introduz uma nova técnica de projeto de algoritmos: o uso de uma estrutura de dados, nesse caso uma estrutura que chamamos *heap* (ou "monte") para gerenciar informações durante a execução do algoritmo.

Um *heap* é um objeto que pode ser visto como uma árvore binária[4] praticamente completa. Cada nó da árvore corresponde a um elemento do vetor que armazena o valor no nó. A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda até certo ponto.

[4]Em ciência da computação, a árvore de busca binária ou árvore de pesquisa binária é uma árvore binária onde todos os nós são valores, todo nó à esquerda contêm uma sub-árvore com os valores menores ao nó raiz da sub-árvore e todos os nós da sub-árvore à direita contêm somente valores maiores ao nó raiz.

Um vetor A que representa um *heap* é um objeto com dois atributos: $comprimento[A]$, que é o número de elementos no vetor, e $tamanho-do-heap[A]$, o número de elementos no *heap* armazenado dentro do vetor. A raiz da árvore é A[1] e, dado o índice i de um nó, os índices de seu pai `Parent(i)`, do filho da esquerda `Left(i)` e do filho da direita `Right(i)` podem ser calculados de modo simples, conforme é ilustrado nos quadros da Figura 3.7.

```
Parent(i)
        return ⌊i/2⌋
```

```
Left(i)
        return 2i
```

```
Right(i)
        return 2i + 1
```

Figura 3.7: Implementação dos procedimentos `Parent`, `Left` e `Right`.

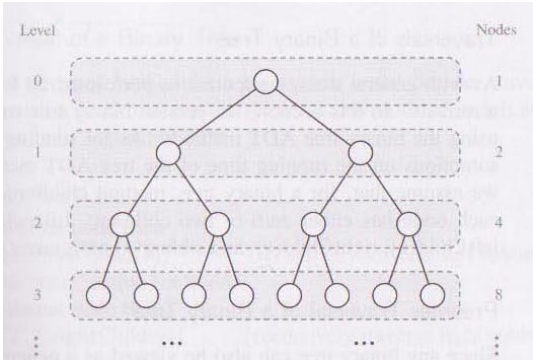


Figura 3.8: Número máximo de nós em cada nível de uma árvore binária. Figura extraída de [11].

Existem dois tipos de *heaps* binários: *heaps* máximos e *heaps* mínimos. Para o algoritmo `Heapsort`, usaremos *heaps* máximos. A propridade de *heap* máximo é que, para todo i diferente da raiz, $A[\texttt{Parent(i)}] \geq A[i]$, isto é, o valor de um nó é no máximo o valor do seu pai. Desse modo, o maior elemento em um *heap* máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó.

Visualizando um heap como uma árvore (Figura 3.8), definimos a altura de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e definimos a altura do heap como a altura h de sua raiz. Em geral, o nível d de uma árvore binária tem não mais

que 2^d nós, que modo que o número total de nós de uma árvore binária é pelo menos $2h + 1$ e não mais que $2^{h+1} - 1$. A Figura 3.8 mostra o número máximo de nós em cada nível.

Manutenção da propriedade de *Heap*

O procedimento `MaxHeapify`, executado no tempo $O(\log n)$, é a chave para manter a propridade de *heap* máximo. Suas entradas são um vetor A e um índice i. Quando `MaxHeapify` é chamado, supomos que as árvores binárias com raízes em `Left(i)` e `Right(i)` são *heaps* máximos, mas que $A[i]$ pode ser menor que seus filhos, violando a propriedade de *heap máximo*.

A função de `MaxHeapify`, ilustrado no quadro da Figura 3.9 é deixar que o valor em $A[i]$ "flutue para baixo"no *heap* máximo, de tal forma que a subárvore com raiz no índice i se torne um *heap* máximo.

```
MaxHeapify(A,i)
01.          l ↩ Left(i)
02.          r ↩ Right(i)
03.          if l ≤ tamanho-do-heap[A] and A[l] > A[i] then
04.                maior ↩ l
05.          else maior ↩ i
06.          if r ≤ tamanho-do-heap[A] and A[r] > A[maior] then
07.                maior ↩ r
08.          if maior ≠ i then
09.                A[i] ⇌ A[maior]
10.                MaxHeapify(A,maior)
11.          end
```

Figura 3.9: Implementação do procedimento `MaxHeapify`.

Em cada passo, o maior entre os elementos $A[i]$, A[`Left(i)`] e A[`Right(i)`] é determinado, e seu índice é armazenado em *maior*. Se $A[i]$ é maior, então a subárvore com raiz no nó i é um *heap* máximo e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e $A[i]$ é trocado por $A[maior]$, o que faz o nó i e seus filhos satisfazerem a propriedade de *heap* máximo. Porém, agora o nó indexado por *maior* tem o valor original $A[i]$ e, desse modo, a subárvore com raiz em *maior* pode violar a propriedade de *heap* máximo. Em consequência disso, `MaxHeapify` deve ser chamado recursivamente nessa subárvore.

A construção de um *Heap*

Podemos usar o procedimento `MaxHeapify` de baixo para cima, a fim de converter um vetor $A[1,\ldots,n]$ em um *heap* máximo. Como os elementos no subvetor $A[\lfloor n/2\rfloor+1,\ldots,n]$ são todos folhas da árvore, então cada um deles é um *heap* de 1 elemento com o qual podemos começar. O procedimento `BuildMaxHeap` percorre os nós restantes da árvore e executa `MaxHeapify` sobre cada um, como no quadro da Figura 3.10.

```
BuildMaxHeap(A, n)
01.          for i = ⌊n/2⌋ downto 1
02.               do MaxHeapify(A,i)
```

Figura 3.10: Implementação do procedimento `BuildMaxHeap`.

O algoritmo HeapSort

O algoritmo `HeapSort` (ou ordenação por monte) começa usando `BuildMaxHeap` para construir um *heap* no vetor de entrada $A[1,\ldots,n]$. Tendo em vista que o elemento máximo do vetor está armazenado na raiz $A[1]$, ele pode ser colocado em sua posição final correta, trocando-se esse elemento por $A[n]$. Se agora "descartamos"o nó n do *heap* (diminuindo *tamanho-do-heap*$[A]$), observaremos que $A[1,\ldots,n-1]$ pode ser facilmente transformado em um *heap* máximo.

Os filhos da raiz continuam sendo *heaps* máximos, mas o novo elemento raiz pode violar a propriedade de *heap* máximo. Porém, tudo o que é necessário para restabelecer a propriedade de *heap* máximo é uma chamada `MaxHeapify(A,1)`, que deixa um *heap* máximo em $A[1,\ldots,n-1]$.

Então, o algoritmo `HeapSort` repete esse processo para o *heap* de tamanho $n-1$, descendo até um *heap* de tamanho 2. O algoritmo é apresentado no quadro da Figura 3.11.

```
HeapSort(A)
01.          BuildMaxHeap(A)
02.          for i = n downto 2 do
03.               A[1] ⇌ A[i]
04.               tamanho-do-heap[A] ↩ tamanho-do-heap[A] − 1
05.               MaxHeapify(A,1)
```

Figura 3.11: Implementação de `HeapSort`.

Algumas características do HeapSort

Cormen *et al.* [7] apresentam algumas características acerca do desempenho do algoritmo `HeapSort`, das quais destacamos:

- O procedimento `MaxHeapify` é executado no tempo $O(\log n)$, e é a chave para manter a propriedade de *heap* máximo;
- O procedimento `BuildMaxHeap`, executado em tempo linear, produz um *heap* a partir de um vetor de entrada não ordenado;
- O procedimento `HeapSort`, executado no tempo $O(n \log n)$, ordena um vetor localmente.

3.3.4 Características dos métodos de Ordenação por Comparação

Apresentamos até agora dois algoritmos que podem ordenar n números no tempo $O(n \log n)$. `HeapSort` alcança esse limite superior no pior caso, enquanto `QuickSort` o alcança na média.

Esses algoritmos compartilham uma propriedade interessante: a sequência ordenada que eles determinam se baseia apenas em *comparações* entre os elementos de entrada. Chamamos esses algoritmos de *ordenações por comparação*. Todos os algoritmos de ordenação apresentados até agora são de ordenações por comparação.

Limites inferiores para ordenação

Em uma ordenação por comparação, usamos apenas comparações entre elementos para obter informações de ordem sobre uma sequência de entrada $A[1 \cdots n]$. Ou seja, dados dois elementos $A[i]$ e $A[j]$, executamos um dos testes $A[i] < A[j]$, $A[i] \leq A[j]$, $A[i] = A[j]$, $A[i] \geq A[j]$ ou $A[i] > A[j]$, para determinar sua ordem relativa. Cormen *et al.* [7] utilizam o modelo de árvore de decisão para estabelecer um limite inferior sobre o tempo de execução de qualquer algoritmo de ordenação por comparação, apresentando o seguinte resultado[5], na forma de Teorema[6]:

> *Qualquer algoritmo de ordenação por comparação exige* $\Omega(n \log n)$ *comparações no pior caso.*

[5] Da mesma maneira que a notação O fornece um limite assintótico superior sobre uma função, a notação Ω fornece um limite assintótico inferior.

[6] Teorema 8.1, de Cormen *et al.* [7].

3.3.5 Ordenação em tempo linear

Alguns algoritmos de ordenação são executados em tempo linear. Esses algoritmos utilizam operações diferentes de comparações para determinar a sequência ordenada. Em consequência disso, o limite inferior $\Omega(n \log n)$ não se aplica a eles.

Resultados envolvendo tempo linear também podem ser obtidos quando desejamos ordenar apenas parte de um vetor, ou determinar o elemento que ocupa uma dada posição no vetor ordenado, a chamada estatística de ordem. Esse assunto será abordado no próximo capítulo.

3.4 FlashSort

`FlashSort`(K. D. Neubert, 1998)[19] é um algoritmo que não utiliza comparações entre todos os elementos do vetor para determinar a sequência ordenada, e, portanto, o limite inferior $\Omega(n \log n)$ não se aplica a ele, de acordo com o que foi mencionado na seção 3.3.5.

Segundo Neubert[19], `FlashSort` ordena n números no tempo $O(n)$ sob a hipótese de um distribuição uniforme, já que é possível determinar a posição final "aproximada"de cada número, sem que haja a necessidade de comparações. Vale mencionar que apenas uma justificativa superficial para o fato do tempo de execução ser $O(n)$ é mencionada pelo autor. Não nos ateremos aqui a este fato, já que, conforme mencionado anteriormente, estamos interessados na determinação da $k-$ésima estatística de ordem, e não na ordenação completa do vetor. Além disso, a eficiência do algoritmo está fortemente apoiada no fato dos números apresentarem distribuição uniforme, diferente do tipo de distribuição que trabalharemos em parte dos experimentos numéricos.

O ponto central de `FlashSort` está na divisão do vetor A de n elementos em m classes, do modo que, cada classe contenha aproximadamente $\frac{n}{m}$ elementos. O algoritmo está estruturado em três blocos lógicos, a saber: *classificação*, *permutação* e *inserção*.

A implementação do algoritmo `FlashSort` é apresentada na Figura 3.12. Supomos, neste caso, que A_{min} e A_{max}, os valores mínimo e máximo do vetor, respectivamente, sejam conhecidos, e determinação é feita de acordo com o algoritmo mostrado na Figura 3.13.

```
FlashSort(A,L,n,m)
01.         C = (M − 1)/(A_max − A_min)
02.         for i = 1 to n
03.               K ↩ 1 + int(C × (A(i) − A_min))
04.               L(K) ↩ L(K) + 1
05.         end
06.         for i = 1 to m
07.               L(K) ↩ L(K) + L(K − 1)
08.         end
09.         A_max ⇌ A(1)
10.         move ↩ 0; j ↩ 0;K ↩ m
11.         while (move < n − 1) do
12.               while (j > L(K)) do
13.                     j ↩ j+1; K ↩ 1+int(C×(A(i)−A_min))
14.               end
15.               flash ↩ a(j)
16.               while (j ≠ L(K) + 1)
17.                     K ↩ 1 + int(C × (flash − A_min))
18.                     A(L(K)) ⇌ hold
19.                     L(K) ↩ L(K) − 1; move ↩ move + 1
20.               end
21.         end
22.         for i = 1 to n
23.               if A(i + 1) > A(i) then
24.                     hold ↩ A(i); j ↩ i
25.                     while (A(j + 1) > hold) do
26.                           A(j) ↩ A(j + 1); j ↩ j + 1
27.                     end
28.                     A(j) ↩ hold
29.               end
30.         end
```

Figura 3.12: Implementação de `FlashSort`.

```
01.     A_min ↩ A(1)
02.     A_max ↩ A(1)
03.     for i = 1 to n
04.         if A(i) < A_min then
05.             A_min ↩ A(i)
06.         if A(i) > A_max then
07.             A_max ↩ A(i)
08.     end
```

Figura 3.13: Determinação de A_{min} e A_{max}.

Na etapa inicial, caracterizada como *classificação*, percorremos o vetor para determinar a qual classe pertence cada elemento $A(i)$ do vetor, computando-se o valor:

$$K(A(i)) = 1 + int\left((m-1)\frac{A(i) - A_{min}}{A_{max} - A_{min}}\right). \tag{3.4.1}$$

Aqui, m é o número de classes que se deseja formar e *int* representa a parte inteira do valor obtido. O resultado obtido será um número entre 1 e m, chamado *classe* de $A(i)$, e, em média, cada classe terá aproximadamente $\frac{n}{m}$ elementos.

Percorrendo-se o vetor, é possível saber a qual classe pertence cada um dos elementos e, portanto, o número de elementos que cada uma das classes contém. Um vetor auxiliar L, de dimensão m, inicialmente contendo zero em todas as posições, armazena esta informação. O vetor L é em seguida atualizado de modo que cada $L(i)$ seja igual ao que chamaremos de quantidade acumulada de elementos, em cada uma das classes de 1 até i. Assim, $L(1)$ é simplesmente o número de elementos na classe 1, $L(2)$ é o total nas classes 1 e 2, e assim sucessivamente, de modo que $L(m)$ é igual a n, o total de elementos no vetor. Analisando com mais atenção a expressão (3.4.1), vemos que o número de elementos de $L(m-1)$ será sempre igual a $n-1$, independente da distribuição dos elementos nas classes.

Feito isso, é possível saber onde cada classe aparecerá no vetor ordenado. Na etapa seguinte, chamada *permutação*, percorremos o vetor movendo cada elemento para sua classe correta. Dado um elemento qualquer $A(i)$, computamos a classe a qual ele pertence, por meio da expressão 3.4.1. A seguir, verificamos no vetor L quantos elementos contém aquela classe, movendo o elemento para esta posição. Um elemento que pertence a uma classe que contém um total acumulado de j elementos será movido para a posição $A(j)$. O número de elementos acumulado naquela classe é então reduzido em uma unidade, e o mesmo procedimento é repetido para o elemento que antes ocupava a posição $A(j)$. Esse processo é repetido até que uma das classes esteja complemente cheia, ou seja, o total acumulado naquela classe é zero (ou seja, não existem mais "vagas"naquela classe). O contador j é então incrementado até encontrar um novo

item que satisfaça a condição $j < L(K(A(j)))$, e o processo continua até que todos os elementos do vetor tenham sido deslocados para suas classes corretas. Ao fim do processo, temos o vetor ordenado segundo suas classes.

Por fim, na etapa de *inserção*, cada uma das classes é ordenada, determinando assim a posição final dos elementos no vetor. Isso é feito de maneira análoga ao procedimento de *Ordenação por Inserção*, descrito na seção 3.3.1: o vetor é percorrido da direita para a esquerda, e à medida que avança vai deixando os elementos mais à direita ordenados.

3.4.1 Um exemplo

O exemplo a seguir permite visualizar cada uma das etapas de `FlashSort`. Tomemos um vetor arbitrário composto por 20 elementos, e suponha que seja dividido em quatro classes.

45	56	7	10	34	89	33	3	13	67	53	23	39	50	47	49	78	65	46	12

Inicialmente, percorre-se o vetor para obtenção dos valores máximo e mínimo: $A_{min} = 3$ e $A_{max} = 89$. Por meio da expressão (3.4.1), determina-se a qual classe K pertence cada um dos elementos:

	45	56	7	10	34	89	33	3	13	67	53	23	39	50	47	49	78	65	46	12
K	2	2	1	1	2	4	2	1	1	3	2	1	2	2	2	2	3	3	2	1

Temos então o total de elementos em cada classe:

classe	1	2	3	4
n.o. elementos	6	10	3	1

e também o total acumulado em cada uma delas.

classe	1	2	3	4
n.o. elementos (acumulado)	6	16	19	20

Esta etapa é finalizada efetuando-se uma troca de posições entre $A(1)$ e A_{max}, de modo que o maior elemento é trazido para a primeira posição do vetor; no caso, os elementos 45 e 89 são trocados entre:

	89	56	7	10	34	45	33	3	13	67	53	23	39	50	47	49	78	65	46	12
K	4	2	1	1	2	2	2	1	1	3	2	1	2	2	2	2	3	3	2	1

Na próxima etapa, permutam-se então os elementos de modo que ocupem as classes corretas. Um contador j é incrementado sempre que um elemento é deslocado, e cada elemento é deslocado uma única vez nessa etapa. Iniciando com $j = 1$, temos que $A(j) = 89$, pertence à classe 4. Como o número de elementos acumulados nessa classe é 20, o elemento 89 é deslocado para a 20.a. posição, e o valor

é atualizado para 19 (na verdade, isso indica que existem ainda 19 "vagas"a serem preenchidas). O elemento que antes ocupava aquela posição, no caso o 12, pertence à classe 1, que contém 6 elementos. Ele é então deslocado para a posição 6, e o total acumulado naquela classe é atualizado para o valor 5.

Ao final do processo de permutação, o vetor apresenta-se do seguinte modo:

	3	7	10	13	23	12	33	56	46	53	34	39	50	47	49	45	65	78	67	89
K	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	4

Finalmente, organizam-se os elementos, utilizando-se o processo de inserção em cada uma das classes, obtendo desse modo o vetor ordenado, representado a seguir:

	3	7	10	12	13	23	33	34	39	45	46	47	49	50	53	56	65	67	78	89
K	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	4

A eficiência desse algoritmo depende do ajuste do número de classes a serem utilizadas. Assim, diminuindo-se o número de classes, o tempo na etapa de classificação diminui, mas na etapa de inserção aumenta. Uma discussão acerca desse fato será apresentada na seção 5.1, por meio de resultados obtidos em experimentos numéricos.

Capítulo 4

Estatísticas de Ordem

Conforme apresentado anteriormente, o cálculo do VaR (2.4.2) consiste basicamente na determinação do elemento que ocupa o $k-$ésimo lugar na ordenação (2.4.1), com $1 \leq k \leq m$. Quando se trata da CVaR (2.4.3), conhecido o valor que ocupa a $k-$ésima posição, basta apenas saber quais elementos do vetor são maiores que este, não sendo necessário que estejam ordenados. Em diversas outras aplicações, estamos interessados em determinar apenas o elemento que ocupa uma dada posição em uma sequência ordenada. Um procedimento seria ordenar todos os elementos do vetor e tomar aquele que ocupa a posição desejada. Entretanto, é possível desenvolver procedimentos menos "dispendiosos", adaptando versões de algoritmos de particionamento, como o `QuickSort`, por exemplo.

As operações de encontrar o máximo, o mínimo e a mediana de um vetor são casos especiais do problema que chamamos de *seleção:* determinar o $k-$ésimo menor elemento de um conjunto de n números. O uso da mediana e de outras *estatísticas de ordem* para dividir um vetor é comum em muitas aplicações que envolvem processamento de dados experimentais. Frequentemente, apenas uma parte de grandes arquivos é processada, ao invés do arquivo todo.

Podemos pensar inicialmente em um algoritmo simples para determinação de uma dada *estatística de ordem*, adaptada do método de *Ordenação por Seleção.* Se k é pequeno, este método mostra-se bastante eficiente, e requer um tempo proporcional a nk: primeiro encontramos o menor elemento, depois o segundo menor elemento, e assim sucessivamente. Entretanto, à medida que k cresce, torna-se necessário o uso de outros métodos mais eficientes.

4.1 Seleção em tempo linear

Na obtenção do mínimo (ou do máximo) de um vetor, basta examinar cada elemento isoladamente e manter o controle do menor (ou maior) elemento visto até então. Podemos estabelecer facilmente um limite superior de $n-1$ comparações neste procedimento, de modo que um algoritmo que alcança tempo de execução $O(n)$ para este problema pode ser desenvolvido.

O problema de seleção geral, ou seja, a determinação do elemento que ocupa o $k-$ésimo lugar, parece mais difícil que o problema de se achar um mínimo, embora o tempo de execução assintótico para ambos os problemas seja o mesmo: $\Theta(n)$[1].

Apresentamos a seguir um algoritmo de dividir e conquistar para o problema de seleção, modelado sobre o algoritmo `QuickSort` da seção 3.3.2, e que denominaremos `QuickSelect`[7]. Como no `QuickSort`, a idéia é particionar o vetor de entrada recursivamente. Porém, diferente deste, que processa ambos os lados da partição, `QuickSelect` só funciona sobre um lado da partição. Essa diferença fica evidente na análise: enquanto `QuickSort` tem um tempo esperado $O(n \log n)$, o tempo esperado de `QuickSelect` é $O(n)$.

Neste caso, `QuickSelect` utiliza o procedimento `RandPartition` introduzido na seção 3.3.2. Desse modo, como o algoritmo `QuickSort aleatório`, seu comportamento é determinado pela saída de um gerador de números aleatórios. O código apresentado na Figura 4.1 retorna o $k-$ésimo elemento do vetor $A[l \ldots r]$.

Após `RandPartition` ser executado na linha 3, o vetor $A[l \cdots r]$ é particionado em dois subvetores $A[l \cdots q-1]$ e $A[q+1 \cdots r]$ tais que cada elemento de $A[l \cdots q-1]$ é menor ou igual ao pivô $A[q]$ que, por sua vez, é menor que cada elemento de $A[q+1 \cdots r]$. A linha 4 calcula o número i de elementos no subvetor $A[l \cdots q]$, ou seja, o número de elementos na parte esquerda da partição, mais uma unidade para o pivô. A linha 5 verifica se $A[q]$ é o $k-$ésimo elemento. Se for, então $A[q]$ é retornado. Senão, o algoritmo determina em qual dos dois subvetores $A[l \cdots q-1]$ e $A[q+1 \cdots r]$ o $k-$ésimo elemento se encontra. Se $k < i$, então o elemento desejado está na parte esquerda da partição, e é recursivamente selecionado do subvetor na linha 8. Porém, se $k > i$, o elemento desejado reside na parte direita da partição. Como já conhecemos i valores que são menores que o $k-$ésimo elemento de $A[l \cdots r]$, o elemento desejado é o $(k-i)$-ésimo elemento de $A[q+1 \cdots r]$, encontrado recursivamente na linha 10.

O tempo de execução no pior caso para `QuickSelect` é $O(n^2)$, o mesmo para encontrar o mínimo, porque poderíamos estar sem sorte e sempre efetuar a partição em torno do maior elemento. Entretanto, o algoritmo funciona bem no caso médio e, porque ele é aleatório, nenhuma entrada específica surge do comportamento no pior caso. De acordo com Cormen *et al.* [7], o tempo exigido por `QuickSelect` em um vetor de entrada $A[l \cdots r]$ é uma variável aleatória cujo limite superior é $O(n)$, o que nos permite

[1] A notação Θ limita assintoticamente uma função acima e abaixo. Quando temos apenas um limite assintótico superior, usamos a notação O, e quando esse limite for inferior, usamos Ω.

```
QuickSelect(A,l,r,k)
01.          if l = r then
02.                return A[l]
03.          q ⇌ RandPartition(A,l,r)
04.          i ⇌ q − l + 1
05.          if k = i then
06.                return A[q]
07.          elseif k < i then
08.                return RandSelect(A,l,q − 1,k)
09.          else
10.                return RandSelect(A,q+1,r,k−i)
```

Figura 4.1: Implementação de `QuickSelect`.

concluir que qualquer estatística de ordem pode ser determinada em média no *tempo linear*.

4.2 Alterações no Método FlashSort

Visto que, em nosso problema, não é necessário ordenar todo o vetor, mas determinar apenas o elemento que ocupa a $k-$ésima posição no vetor ordenado, sugerimos uma alteração no método `FlashSort`, apresentado na seção 3.4, para tratar desse caso. Basta que se ordene apenas a classe que contém a posição do elemento procurado.

Uma simples inspeção no vetor L acumulado, ao final do processo de classificação, permite obter os índices das posições limite da classe que contém a $k-$ésima posição. O vetor, que antes era percorrido da direita para a esquerda, é agora percorrido apenas do limite inferior ao limite superior, garantido que toda a classe esteja ordenada, e, portanto, determinando o elemento desejado.

Outra possível alteração do algoritmo diz respeito à substituição do terceiro bloco, que utiliza o método da inserção, por um método de ordenação mais eficiente, como o `QuickSelect`.

4.3 SecaMenores

Apresentamos a seguir o método chamado *SecaMenores*, proposto por Andreani-Martinez-Martinez-Yano [5]. Baseado no *Método da Secante*[2], este algoritmo utiliza sucessivas construções de retas secantes, que

[2]O *Método da Secante* é um método de iteração para se obter os zeros de uma função suave f. Os pontos de cruzamento da secante de uma aproximação são usados como pontos de base inicial para uma nova secante, mais próxima do zero real.

fornecem aproximações para o elemento que ocupa a $k-$ésima posição no vetor ordenado. A implementação do algoritmo *SecaMenores* para determinação do elemento da $k-$ésima posição é apresentada na Figura 4.2.

```
SecaMenores(A,k)
01.     f_max = n − k
02.     f_min = −k
03.     der = (f_max − f_min)/(A_max − A_min)
04.     v = (A_max − A_min)/der
05.     if v < A_min + 0.1(A_max − A_min) ou v > A_max − 0.1(A_max − A_min) then
06.           v = 0.5(A_max − A_min)
07.     c ↩ 0
08.     for i = 1 to n
09.           if A(i) < v then
10.                 c=c+1
11.     end
12.     if c = n then
13.           j ↩ 1
14.           for i = 1 to n
15.                 if A(i) < v then
16.                       L(j) ↩ i ; j ↩ j + 1
17.           end
18.     if A(i) < v then
19.           A_min ↩ v; f_min ↩ c − k
20.     else
21.           A_max ↩ v; f_max ↩ c − k
```

Figura 4.2: Implementação de *SecaMenores*.

Tomemos inicialmente um vetor A composto por n elementos, tal que min e max, sejam, respectivamente, os índices do menor e o maior elementos. Sejam A_{min} e A_{max}, respectivamente, tais elementos. Pensando no plano cartesiano, onde os elementos no vetor são representados no eixo das abscissas, e suas índices no eixo das ordenadas, vamos considerar o índice do elemento que procuramos, ou seja, aquele que ocupa a $k-$ésima posição, como a origem do eixo das ordenadas. Seu valor, representado por v, será aproximado pelo ponto que está na intersecção do eixo das abscissas com o segmento de extremidades $(A_{min}, -k)$ e $(A_{max}, N-k)$, conforme podemos observar na figura 4.3.

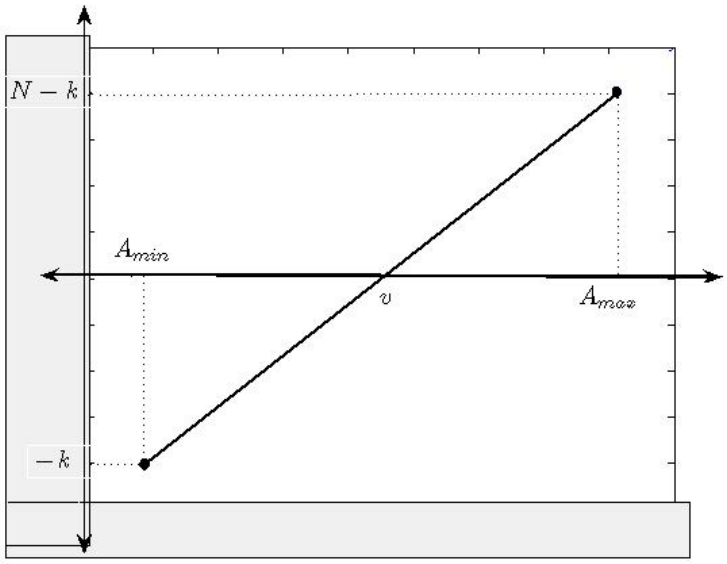


Figura 4.3: Estimativa para o elemento da $k-$ésima posição.

Temos então a seguinte relação de proporcionalidade:

$$\frac{(N-k)-(-k)}{A_{max}-A_{min}} = \frac{(N-k)-(0)}{A_{max}-v},$$

ou seja,

$$\frac{N}{A_{max}-A_{min}} = \frac{N-k}{A_{max}-v}.$$

Como k é conhecido, e A_{min} e A_{max} são facilmente determinados (Figura 3.13), obtém-se então o valor de v, uma *estimativa* para o valor do elemento procurado. O vetor é então percorrido e um contador c é utilizado para determinar quantos elementos são menores que v. A ordenada para v é então corrigida para esse valor.

Aplicamos então a idéia secante iterativamente. O processo termina quando atingimos um v tal que exatamente k elementos são menores que esse valor, e portanto o k-ésimo elemento é obtido. Um vetor auxiliar L fornece os índices dos elementos com valor menor que v. O elemento procurado então é tomado como o maior em todos os elementos cujos índices são dados em L.

Caso o valor obtido para v esteja fora do intervalo de tolerância

$$(A_{min} + 0.1(A_{max} - A_{min}), A_{max} - 0.1(A_{max} - A_{min})),$$

este é substituido pelo ponto médio do intervalo (A_{min}, A_{max}), ou seja:

$$v = \tfrac{A_{max}-A_{min}}{2}.$$

4.3.1 Um exemplo

Exemplificamos esse procedimento, tomando o mesmo vetor apresentado na seção 3.4.1, composto por 20 elementos:

45	56	7	10	34	89	33	3	13	67	53	23	39	50	47	49	78	65	46	12

Suponha que estejamos interessados em encontrar o elemento que ocupa a posição 18 no vetor ordenado, ou seja, $k = 18$.

Inicialmente, determinamos os valores $A_{min} = 3$ e $A_{max} = 89$. Temos portanto uma primeira reta secante, unindo os pontos $(3, -18)$ e $(89, 2)$, representada na Figura 4.4.

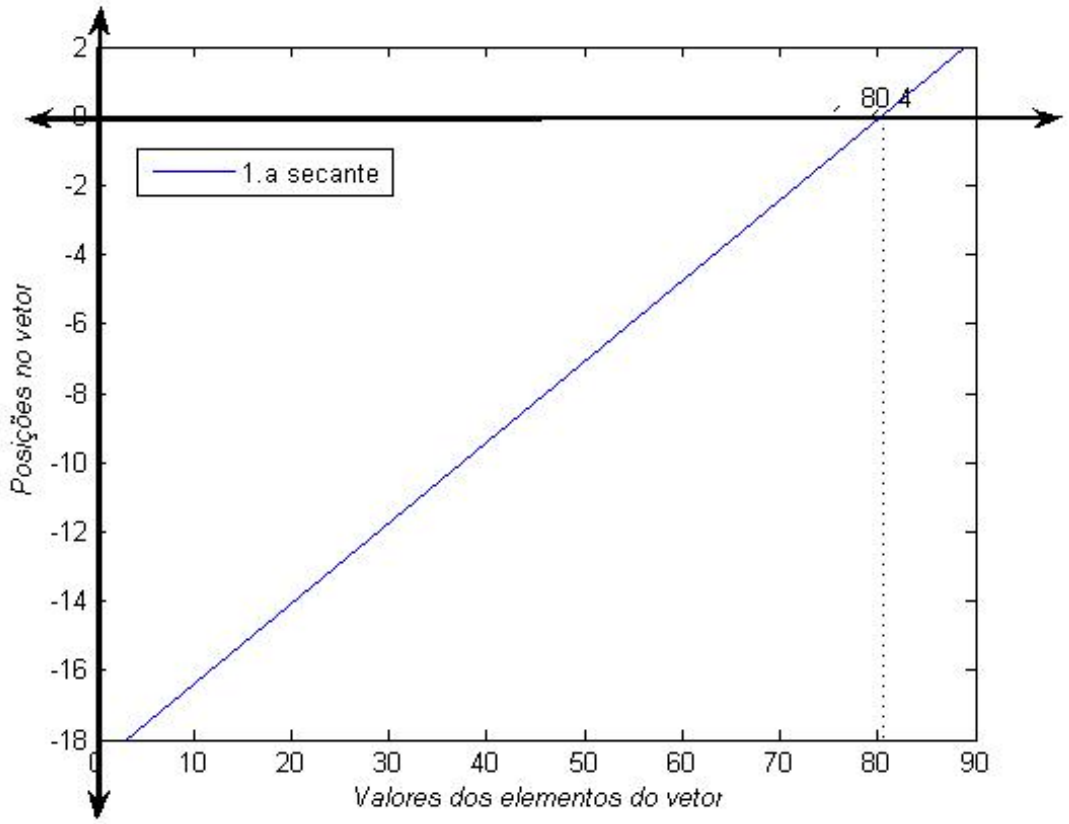


Figura 4.4: Construção da 1.a aproximação secante

Temos então a relação de proporcionalidade:

$$\frac{2-(-18)}{89-3} = \frac{2-0}{89-v},$$

para o qual obtemos $v = 80.4$. Como esse valor não pertence ao intervalo de tolerância [11.6, 80.3], aplica-se o método da bissecção no intervalo $[3, 89]$, obtendo um novo valor $v = 46$. Percorrendo o vetor, determinamos que existem 11 elementos menores que 46, e portanto sua ordenada é corrigida para o valor 11. Um novo segmento é tomado com extremo nos pontos $(46, -7)$ e $(89, 2)$, como ilustrado na Figura 4.5.

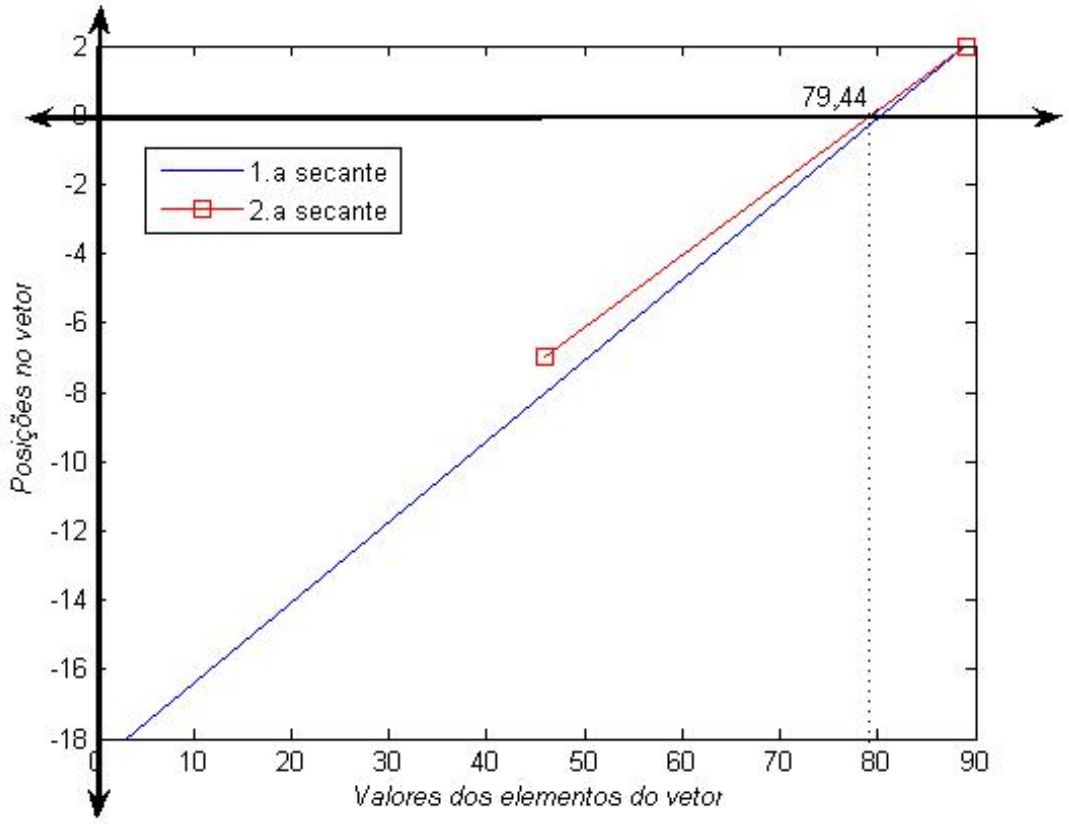


Figura 4.5: Construção da 2.a aproximação secante

Nesse caso, obtemos o valor $v = 79.44$, que pertence ao intervalo de tolerância considerado, e determinamos que existem 19 elementos menores que v. Uma outra aproximação secante, ilustrada na Figura 4.6, fornece o valor $v = 75.26$ para o elemento procurado.

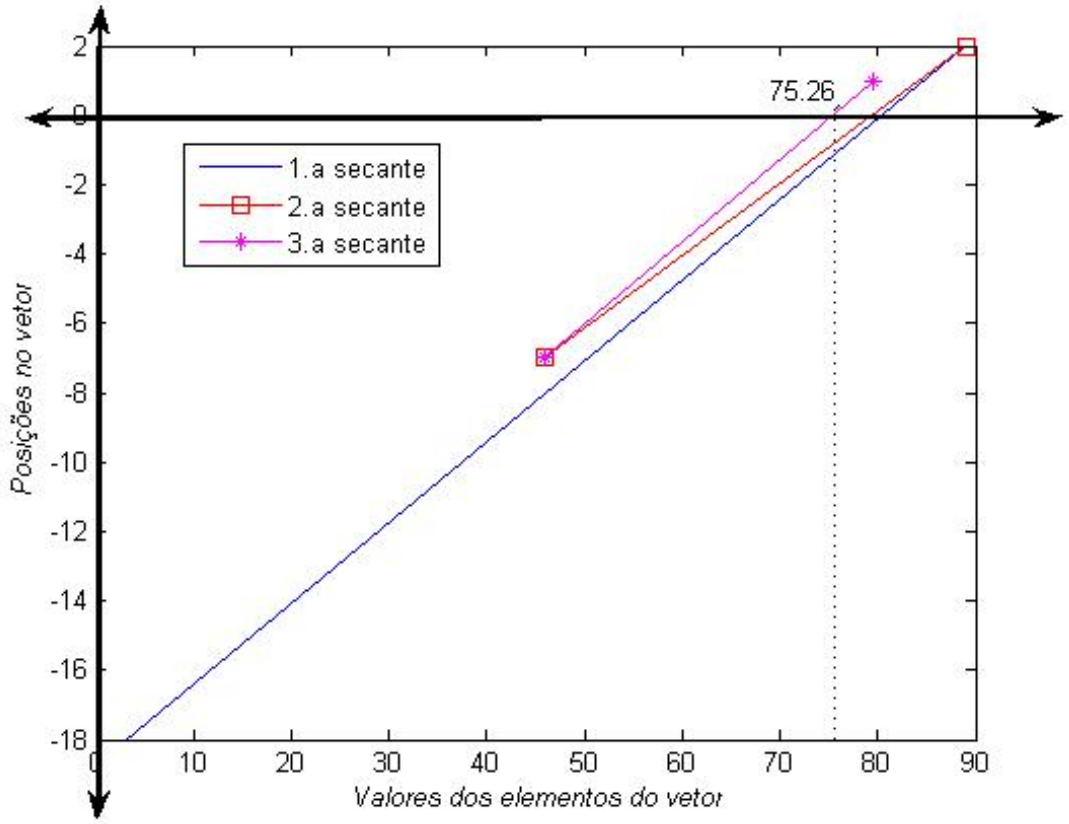


Figura 4.6: Construção da 3.a aproximação secante

Como existem exatamente 18 elementos menores que $v = 75.26$, tomamos esse valor como aproximação secante. O processo é finalizado quando percorremos o vetor armazenando os índices dos elementos que são menores que $v = 75.26$, de modo que o elemento da $k-$ésima posição será o máximo entre eles; no caso, o elemento 67.

Capítulo 5

Experimentos Numéricos

Nosso objetivo nesse capítulo é descrever, por meio de experimentos numéricos, o desempenho dos principais métodos de ordenação descritos anteriormente e verificar qual deles mostrou-se mais eficiente para o problema de determinação de uma dada estatística de ordem. O testes numéricos foram implementados em `Fortran` 77 e compilados em uma mesma máquina, processador Celeron 2.53 GHz.

5.1 A escolha do número de classes em FlashSort

Conforme mencionamos anteriormente, um dos fatores que influencia no desempenho do algoritmo `FlashSort` é a escolha do número de classes. Embora Neubert [19] apresente $m = 0.43n$ como valor ótimo para o número de classes, o valor $m = 0.1n$ também é citado em seu artigo, sendo apresentado na implementação do algoritmo.

Como forma de verificar experimentalmente valores de m que tornem o algoritmo mais eficiente, tomamos amostras uniformes de diversos tamanhos (100 mil, quinhentos mil, 1 milhão, 2 milhões e 5 milhões), e realizamos testes para valores de diversos valores de m, fixando nossa atenção para valores próximos de $0.1n$ e também $0.43n$. Os resultados encontrados foram análogos para qualquer valor de n testado. A figura 5.1 apresenta os resultados para o caso em que o vetor continha 5 milhões de elementos.

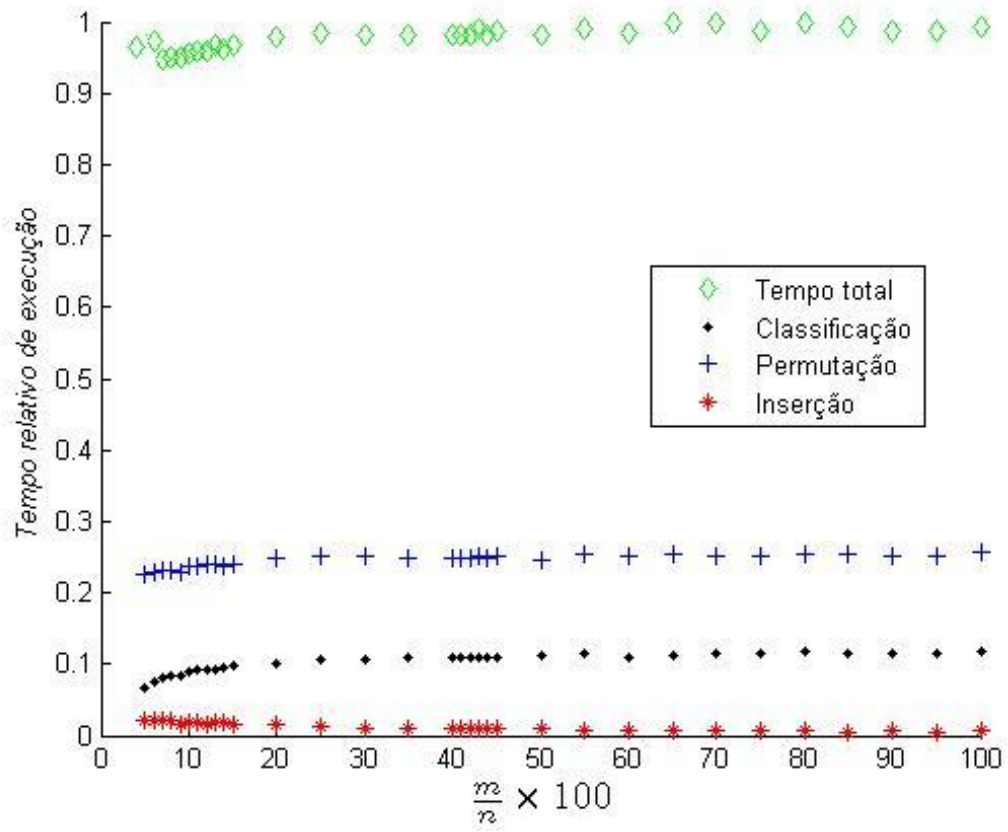


Figura 5.1: Tempo total e distribuição do tempo de das etapas do `FlashSort`, em função da razão $\frac{m}{n}$.

Nesta figura, apresentamos a distribuição do tempo de execução de cada uma das etapas, para diferentes escolhas da razão $\frac{m}{n}$. O tempo de execução relativo é computado em relação ao maior tempo total obtido. Conforme citamos anteriormente, à medida que a razão aumenta, ou seja, conforme aumenta o número de classes, o tempo destinado a etapa de classificação aumenta. Entretanto, na etapa de inserção esse tempo diminui, já que as classes tornam-se menores e, portanto, uma quantidade menor de comparações entre elementos é necessária. Além disso, a etapa mais cara do processo é a permutação, independente do número de classes.

Esses resultados são melhor visualizados no gráfico da Figura 5.2, onde o tempo relativo é agora computado com relação ao maior tempo entre as três etapas, entre todos os testes realizados.

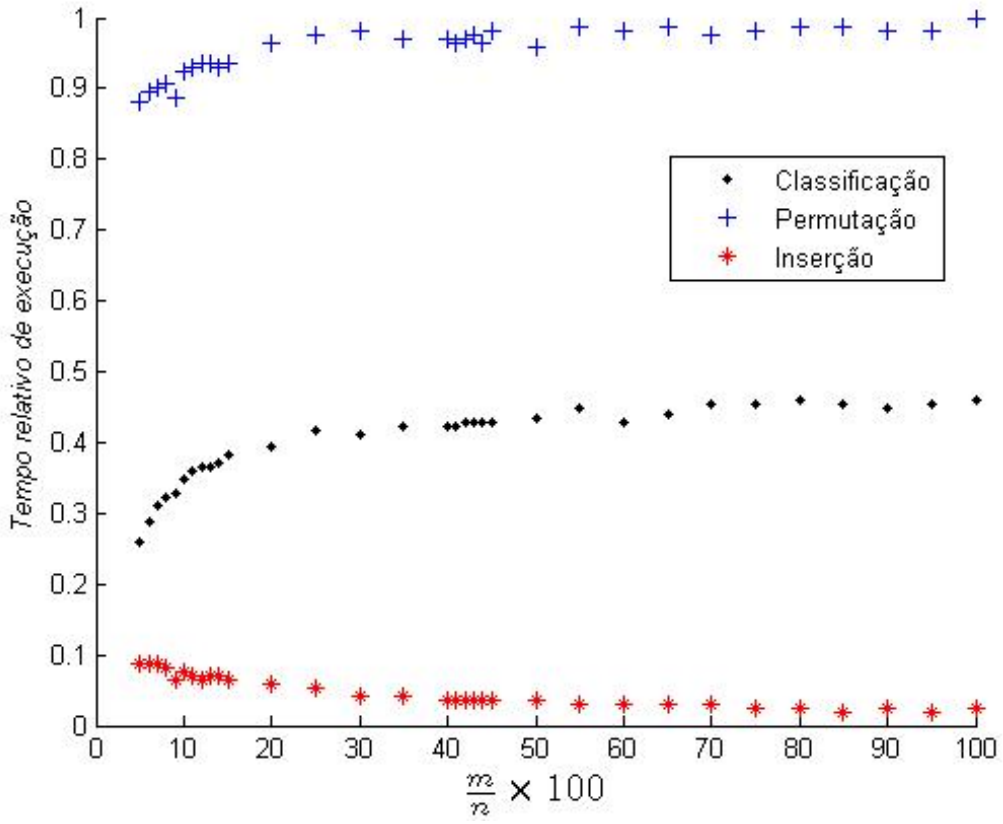


Figura 5.2: Distribuição do tempo de execução das várias etapas do `FlashSort`, em função de $\frac{m}{n}$.

No que diz respeito ao tempo total de execução do algoritmo, podemos observar no gráfico da figura 5.3 que este é bastante instável. O gráfico sugere que os melhores resultados são obtidos quando fazemos escolhas próximas do valor 0.1 para a razão $\frac{m}{n}$. O valor $m = 0.1n$ será adotado em todos os testes envolvendo o algoritmo `FlashSort`, tanto aqueles que envolvem conjuntos com elementos uniformemente distribuidos, quanto para testes com cenários gerados por simulação histórica (seção 5.2).

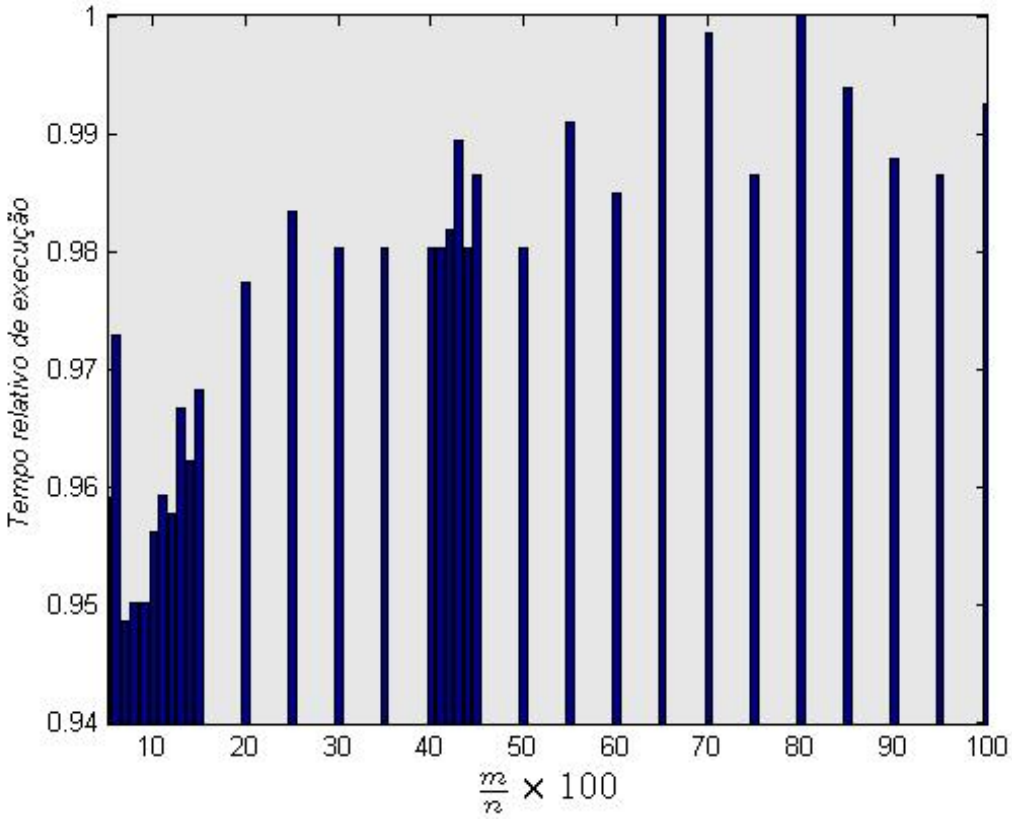


Figura 5.3: Tempo total de execução do `FlashSort` para diversas escolhas de $\frac{n}{m}$.

5.2 Geração de cenários para testes computacionais

A geração de cenários para os testes computacionais apresentados nas seções a seguir foi realizada utilizando-se o método de simulações históricas, descrito na seção 2.3. Sem perda de generalidade, consideramos em nossos testes a geração de cenários para um único ativo.

A idéia de aplicar cada uma das variação passadas sobre seu valor atual é bastante ampla, e diversas inpretações podem ser tomadas nesse sentido. As variações passadas dos preços foram tomadas como a razão $r = \frac{P(i+1)}{P(i)}, 1 \leq i \leq D-1$, onde $P(i)$ e $P(i+1)$ são respectivamente os preços do ativo nos dias i e $i+1$, em uma série histórica de preços de D dias. Para obter um dos cenários de preço do ativo correspodente ao dia $D+T$, contado a partir do início da série histórica, escolhemos aleatoriamente

valores de $r(j)$ e construímos a relação:

$$P(j+1) = r(j) \cdot P(j),$$

com $D \leq j \leq T-1$.

Repetimos esse procedimento tantas vezes quanto for o número de cenários que desejamos obter.

Como forma de exemplificar o procedimento utilizado, apresentamos os dados históricos referentes ao preço de um ativo durante 5 dias:

Dia	Preço
1.o.	10.59
2.o.	10.56
3.o.	10.98
4.o.	10.34
5.o.	10.67

Temos então um conjunto R que representa as variações passadas desse ativo, dadas pela razão entre os preços entre dois dias consutivos:

$$R = \{\tfrac{10.56}{10.59} = 0.997, \tfrac{10.98}{10.56} = 1.039, \tfrac{10.34}{10.98} = 0.942, \tfrac{10.67}{10.34} = 1.032\}.$$

Suponha que desejamos construir um cenário para o preço desse ativo daqui a 7 dias, ou seja, 2 dias depois do fim da série histórica. Tomamos aleatoriamente um valor em R, e multiplicamos pelo preço no 5.o. dia. Escolhendo, por exemplo, o valor 0.942, e efetuando 10.67×0.942, obtemos o valor 10.05. Escolhemos aleatoriamente outro valor e repetimos o procedimento: $10.05 \times 1.039 = 10.44$. Ou seja, 10.44 representa o cenário para o preço do nosso ativo no 7.o. dia da série histórica.

Para nossos experimentos numéricos, utilizamos um conjunto de dados históricos referentes ao preço de um certo ativo num período de 1000 dias consecutivos. A simulação dos cenários futuros será feita o dia 1300 a partir do início da série histórica. Como os preços dos ativos são valores no conjunto dos racionais positivos, trabalhamos com valores truncados. Foram gerados conjunto com quantidades de elementos variando entre 5.000 e 20.000.000.

5.3 Comparação entre métodos de ordenação total

Conjuntos de números uniformes

Como forma de analisar o desempenho do `FlashSort`, realizamos testes computacionais comparando-o com outros dois algoritmos clássicos de ordenação total: o `Quicksort` e o `Heapsort`. Os testes foram

inicialmente realizados com conjuntos de números uniformemente distribuídos, para diversas escolhas de n. No caso do `FlashSort`, tomamos sempre $m = 0.1n$. Os resultados experimentais obtidos são ilustrados graficamente na figura 5.4.

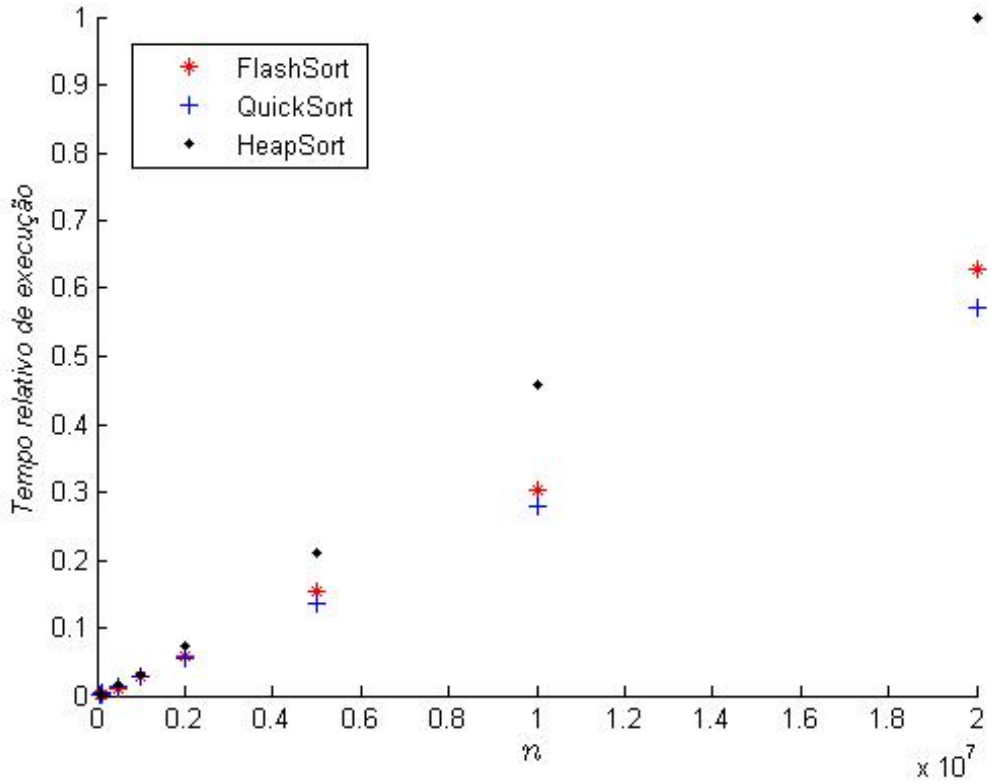


Figura 5.4: Desempenho de alguns algoritmos na ordenação de conjuntos uniformes de números, em função de n.

Os tempos relativos foram computados em relação ao maior tempo de execução obtido. Podemos observar que tanto o `FlashSort` quanto o `QuickSort` são mais rápidos que o `Heapsort` para as amostras analisadas. Estes dois apresentam desempenho parecido para pequenos valores de n. Entretanto, `QuickSort` mostra-se mais eficiente que o `FlashSort` à medida que o número de elementos no vetor aumenta. Vale mencionar aqui que `Heapsort` é indicado para problemas que envolvem a determinação dos maiores ou menores elementos de um vetor.

Da forma como foram plotados esses dados, fica difícil analisar o comportamento dos algoritmos `FlashSort` e `QuickSort` para valores de n muito pequenos. Para evitar trabalhar com valores de tempo

muito pequenos, adotamos então o seguinte procedimento: repetimos a execução do algoritmo um certo número de vezes (no caso, 1000), e tomamos como tempo de execução o valor total obtido. O gráfico da figura 5.5 ilustra os resultados obtidos. Os valores representados no eixo das abscissas correspondem ao tempo relativo ao maior valor de tempo total obtido nos testes.

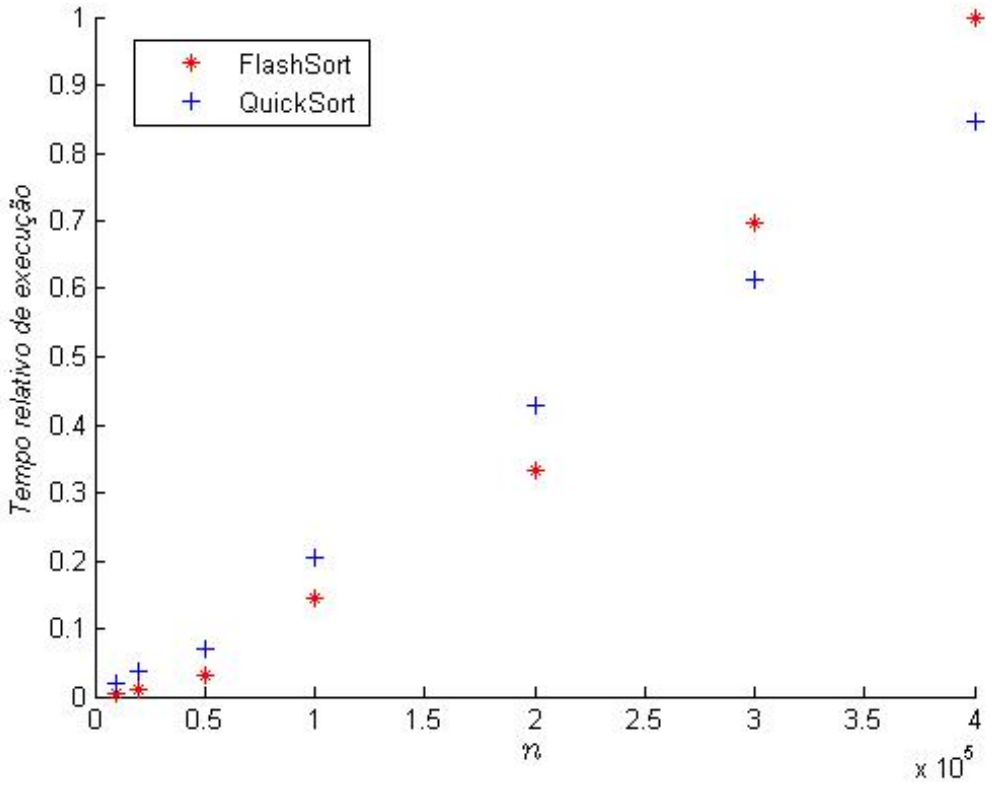


Figura 5.5: Desempenho de alguns algoritmos na ordenação de conjuntos uniformes de números, em função de n.

Vemos aqui um comportamento diferente daquele observado nos testes anteriores. `FlashSort` mostra-se mais rápido para valores de n menores que 300.000. A partir desse valor `QuickSort` mostrou-se mais eficiente. Esse tipo de análise é de extrema importância, uma vez que, em várias situações, o tamanho do vetor a ser ordenado é relativamente pequeno, mas precisa ser ordenado muitas vezes ao longo do programa.

Uma explicação plausível para este fato é que, à medida que o tamanho do vetor aumenta, a movimentação de elementos que encontram distantes de suas classes corretas durante a etapa de *permutação*

do `FlashSort` acaba tornando-se bastante cara. Em Ciência da Computação, esse fenômeno é conhecido como "falha de cache".

De acordo com [28], a *memória cache* reside entre a CPU e a memória principal: é uma pequena porção de memória muito rápida. diferente da memória convencional, os *bytes* que aparecem dentro de uma cache não tem endereços fixos. Em vez disso, a memória cache pode redeterminar o endereço de um dado. Isto permite que o sistema mantenha os valores acessados recentemente na cache. Endereços que a CPU nunca acessou ou não acessou em mais recentemente ficam na memória principal (lenta). Já que a maior parte dos acessos a memóra são para variáveis acessadas recentemente (ou para posições próximas de uma posição acessada recentemente), o dado geralmente aparece na memória cache.

A memória cache não é perfeita. Embora um programa possa gastar considerável tempo executando código em um local, eventualmente ele chamará um procedimento ou desviará para alguma seção distante de código fora da memória cache. Nestes casos a CPU tem que ir a memória principal para buscar os dados. Como a memória principal é lenta, isto requerirá a inserção de estados de espera.

Um *acerto da cache* ("cache hit") ocorre sempre que a CPU acessa a memória e encontra o dado na cache. Em tal caso a CPU pode realmente acessar o dado com zero estados de espera. Uma *falha na cache* ("cache miss") ocorre se a CPU acessa a memória e o dado não está presente na cache. Então a CPU tem que ler o dado da memória principal, causando uma perda de performance. Para tirar vantagem de localidade de referência, a CPU copia dados para dentro da cache sempre que ela acessa um endereço não presente na cache. Como é provável que o sistema acessará aquela mesma posição pouco tempo depois, o sistema economizará estados de espera tendo aquele dado na cache.

Armazenar posições da memória quando você as acessa não agilizará o programa se você constantemente acessar posições distantes entre si. Isso é exatamente o que ocorre na etapa de *permutação* do `FlashSort`, o que justifica o melhor desempenho do *QuickSort* à medida que o tamanho do vetor torna-se muito grande. Na verdade, a existência da memória cache acaba sendo um ponto positivo para o *QuickSort*, uma vez que este algoritmo trabalha com elementos do vetor próximos entre si.

Conjuntos de números gerados por simulação histórica

Em seguida, repetimos os testes utilizando conjunto de gerados por simulação histórica, sendo que, no `FlashSort`, foi usada a inserção na última etapa, como proposto no algoritmo original . Os resultados absolutos de tempo de execução, medidos em segundos, são apresentados na Tabela 5.1.

Nesse caso, percebemos que `FlashSort` mostra-se bastante ineficiente para o conjunto de números considerado, se comparado aos outros dois algoritmos; isso fica evidente mesmo para valores de n relativamente pequenos.

N	**QuickSort** (s)	**FlashSort**(s)	**HeapSort** (s)
5.000	0.0313	0.0781	0.0468
10.000	0.0469	0.1563	0.0625
20.000	0.0625	0.6250	0.0781
50.000	0.1250	3.4844	0.1250
80.000	0.1719	9.0781	0.2187
100.000	0.2344	14.281	0.3750

Tabela 5.1: Desempenho de alguns algoritmos para ordenação de conjuntos de números gerados por simulação histórica.

Vale mencionar que `FlashSort` foi implementado considerando-se uma distribuição uniforme de números. Entretanto, os conjuntos gerados por simulação histórica não seguem esse tipo de distribuição. Isso fica evidente ao observarmos o histograma da Figura 5.6 , construído tomando-se um conjunto com 100 de números gerados por simulação histórica, e distribuidos em 20 classes. Vemos que grande parte dos elementos fica na 1.a classe, de modo que falha a hipótese sobre o qual se apóia o `FlashSort`, ou seja, que cada classe tenha aproximadamente $\frac{n}{m}$ elementos.

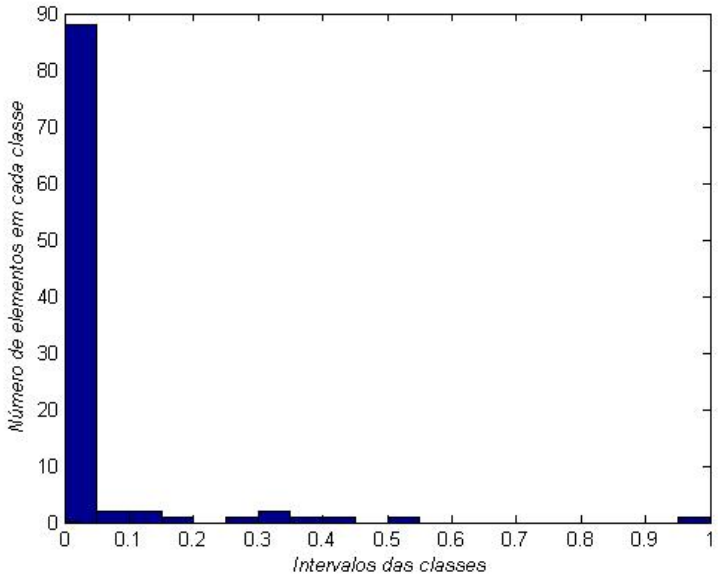


Figura 5.6: Histograma para conjunto de números gerados por Simulação Histórica.

5.4 Determinação da $k-$ésima estatística de ordem

Os experimentos apresentados nessa seção foram realizados tomando-se conjuntos de números gerados por simulação histórica. Em cada um dos casos, dado um vetor com n elementos, desejamos determinar o elemento que ocupa a posição correpondente a $0.85n$.

As alterações no método `FlashSort` foram feitas de modo que, após a etapa de permutação, apenas a classe correspondente à estatística de ordem procurada seja ordenada, conforme explicação da seção 4.2. Inicialmente, mantivemos a estrutura original do algoritmo, ordenando essa classe por meio do método de *inserção*. Comparamos então com o algoritmo `QuickSelect`, na determinação do elemento procurado. Os resultados absolutos de tempo de execução, medidos em segundos, são apresentados na Tabela 5.2. Mais uma vez, `FlashSort` mostra-se bastante ineficiente para o conjunto de números considerado, se comparado ao método `QuickSelect`; isso fica evidente mesmo para valores de n relativamente pequenos.

N	**Quick Select** (s)	**FlashSort modificado** (s)
5.000	0.0468	0.0625
10.000	0.0625	0.1875
20.000	0.0781	0.4531
50.000	0.0937	2.4531
80.000	0.1562	7.6482
100.000	0.1875	9.5937

Tabela 5.2: Desempenho de alguns algoritmos na determinação do elemento que ocupa a posição correpondente a $0.85n$.

Tendo em visto tais resultados, resolvemos alterar a terceira etapa do `FlashSort`: ao invés do método de inserção, usamos `QuickSelect` para ordenar a classe correspondente à estatística de ordem procurada.

Apresentamos no gráfico da Figura 5.7 os resultados para os testes comparativos entre esse algoritmo alterado, que denominamos `FlashSort + Quick Select`, e os algoritmos `QuickSelect` e *SecaMenores*. Os tempos relativos foram computados em relação ao maior tempo de execução obtido.

O algoritmo *SecaMenores* usa comparações entre elementos para contagem de quantos elementos são menores do que v, o que torna o algoritmo mais lento, à medida que o tamanho do vetor aumenta. Isso fica evidente no gráfico. Podemos notar também que as alterações propostas em `FlashSort` tornaram o algoritmo mais eficiente. Entretanto, entre os algoritmos destados, `QuickSelect` mostrou-se mais eficiente na determinação da $k-$ésima estatística de ordem, para qualquer escolha feita para n.

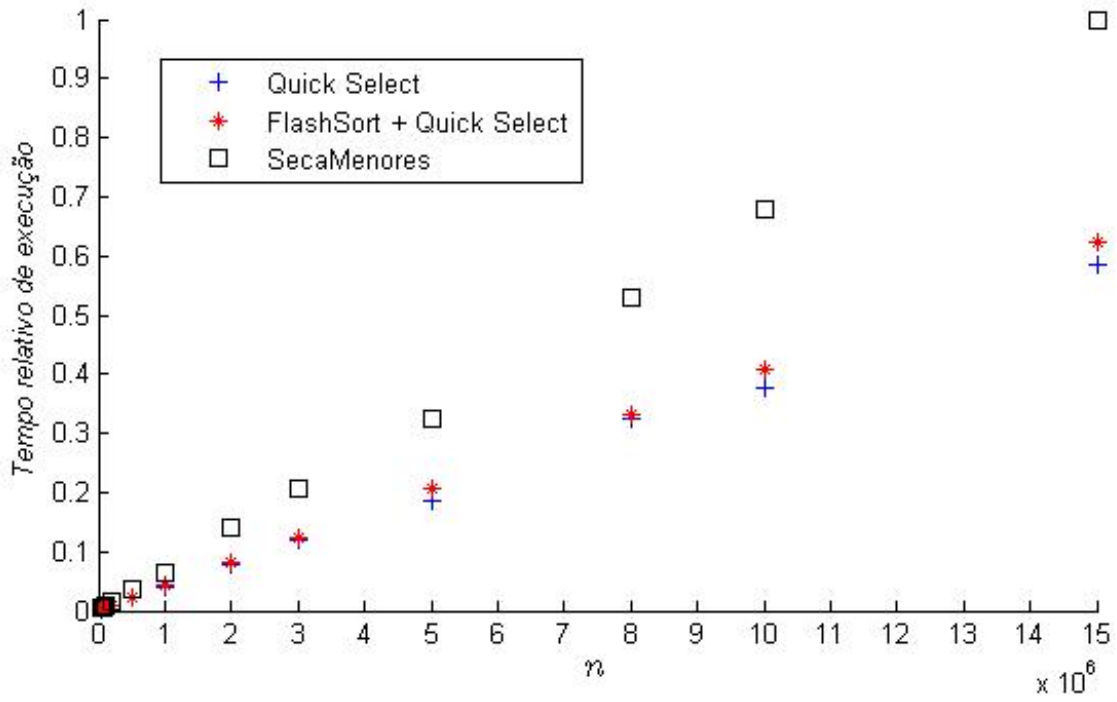


Figura 5.7: Desempenho de alguns algoritmos na determinação do elemento que ocupa a posição correpondente a $0.85n$, em conjuntos gerados por simulação histórica.

Capítulo 6

Considerações finais

Tendo em vista o grande tempo computacional dedicado à ordenação de números ao se resolver Problemas de Otimização do Valor Ordenado, propusemos, neste trabalho, analisar estratégias eficazes na determinação do elemento que ocupa a $k-$ésima posição no vetor ordenado, sem a necessidade de ordenação total.

Após um estudo preliminar dos algoritmos clássicos de ordenação, focamos nossa atenção em dois deles: `QuickSort` e `HeapSort`. Entretanto, ao analisar o problema de seleção, percebemos que `QuickSelect` mostrou-se mais interessante, uma vez que processa apenas o lado da partição que contém a estatística de ordem desejada.

Embora o algoritmo `FlashSort` proposto por Neubert [19] tenha apresentado um bom desempenho para ordenação total de vetores relativamente pequenos (com valores próximos de 300.000 elementos), `QuickSort` mostrou-se mais eficiente, à medida que o tamanho do vetor aumenta, fato este justificado principalmente pelo existência de muito mais falhas de cache durante a execução do primeiro.

No que diz respeito à determinação da $k-$ésima posição, as alterações propostas em `FlashSort` tornaram o algoritmo mais rápido, para problemas envolvendo cenários gerados por simulação histórica. Entretanto, `QuickSelect` mostrou-se o algoritmo mais eficiente, independente do tamanho do vetor. *SecaMenores* mostrou-se mais lento que os demais, à medida que o tamanho do vetor aumenta.

Dentro deste contexto, finalizamos este trabalho com algumas sugestões para trabalhos futuros. Uma primeira possibilidade para alteração em `FlashSort` diz respeito à etapa de *permutação*. Como vimos, à medida que o número de elementos no vetor aumenta, a movimentação de elementos que encontram distantes de suas classes corretas durante esta etapa torna-se bastante cara. Assim, uma vez conhecida a classe de cada um dos elementos, poderíamos trabalhar apenas com um vetor auxiliar que contivesse os elementos pertencentes a classe da $k-$ésima posição.

Uma vez que `FlashSort` está estruturado sob a hipótese de uma distribuição uniforme, caberia uma alteração na expressão (3.4.1), adaptada para o tipo de distribuição envolvendo cenários gerados por simulação histórica.

Por fim, poderia ser verificada a possibilidade de alterações nos algoritmos propostos, de modo a trabalhar com a paralelização dos mesmos. Este tipo de técnica consiste em dividir o espaço de busca da solução conforme a disponibilidade dos processadores, o que possibilitaria maior rapidez na resolução desses problemas.

Bibliografia

[1] Anderson, F.; Mausser, H.; Rosen, D. e Uryasev S. Credit risk optimization with Conditional Value-at-Risk criterion. *Mathematical Programming*, Ser. B 89, p.273 - 291 (2001).

[2] Andreani, R.; Dunder, C. e Martínez, J.M. Order-Value Optimization: formulation and solution by means of a primal Cauchy method. *Mathematical Methods of Operations Research* 58, pp. 387-399 (2003) .

[3] Andreani, R.; Dunder, C. e Martínez, J. M. Nonlinear-Programming Reformulation of the Order-Value Optimization Problem. *Mathematical Methods of Operations Research* 61, pp. 365-384 (2005).

[4] Andreani, R.; Martínez, J. M.; Salvatierra, M. and Yano, F. Quasi-Newton methods for order-value optimization and value-at-risk calculations. A aparecer em *Journal of Pacific Optimization*.

[5] Andreani, R.; Martínez, J. M.; Martínez, L. e Yano, F. S. . Low Order-Value Optimization and new applications. A aparecer em *Journal of Global Optimization*.

[6] Chaia, A. J. e Ferreira, F. Metodologias alternativas de geração de cenários na apuração do VaR de instrumentos nacionais. *IV SEMEAD* (1999).

[7] Cormen, T. H.; Leiserson, C. E. e Rivest, R. L. *Introduction to Algorithms*, McGraw-Hill (1990).

[8] Chu, L. *et al.*. Value at Risk for Investment Portfolios. *Quantitative Perspectives*, Andrew Davidson & Co., (1996).

[9] Dachraoui, T. *Fast parallel algorithms for sorting and median finding*. A thesis in the Departament of Computer Science: Concordia University, Montreal, Québec, Canada (1996).

[10] Gaelzer. R. *Apostila de Fortran 90/95*. Disponível em `http://minerva.ufpel.edu.br/~rudi/grad/ModComp/Apostila/Apostila.html`, acessado em 22/01/08.

[11] Goodrich, M.T. e Tamassia, R. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons (2002).

[12] Johnsonbaugh, R. e Schaefer, M. *Algorithms*. Pearson Education, New Jersey (2004).

[13] Jorion, P. *Value at risk: the new benchmark for managing financial risk*, 2nd edition. Mc Graw-Hill, New York (2001).

[14] Knuth, D. E. *The art of computing programming*. 2ed. vol.3 Sorting and searching. Reading (MA): Addison Wesley (1973).

[15] Martin, W. A. Sorting, *ACM Computing Surveys* 3 (4), p.147-174, Dec. 1971.

[16] Mollica, M. A. *Uma avaliação de modelos de Value-at-Risk: comparação entre métodos tradicionais e modelos de variância condicional*. Tese de mestrado, Faculdade de Economia, Administração e Contabilidade: USP, São Paulo (1999).

[17] Morris, R. Some Theorems on Sorting. *Journal on Applied Mathematics*, 17 (1), Jan. 1969.

[18] Neubert, K. D. Flash-Sort: Sorting by *in situ* permutation. Paper presented at euro FORTH'97, Sep. 1997, Oxford, England. Disponível em `http://www.neubert.net/FSOIntro.html`, acessado em 22/01/08.

[19] Neubert, K. D. The Flashsort1 Algorithm. *Dr. Dobb's Journal*, Feb. 1998. Disponível em `http://www.neubert.net/Flapaper/9802n.htm`, acessado em 05/01/08.

[20] Reyna, F. Q. *Teoria de Finanças*. Resumo das aulas. Disponível em `http://webold.impa.br/Ensino/Atividades/2004/Cursos`, acessado em 22/01/08.

[21] Rockafellar, R.T. e Uryasev, S. Conditional value-at-risk for general loss distributions. *Journal of Banking and Finance* 26, p. 1443-1471 (2002).

[22] Scowan, R. S. Algorithm 271: Quicksort. *Communications of the ACM* 8 (11), p. 669-670, Nov. 1965.

[23] Sedgewick, R. *Algorithms*, 2ed. Addison-Wesley Publishing Company, 1988.

[24] Szegö, G. Measures of risk. *Journal of Banking & Finance* 26, p.1253 - 1272 (2002).

[25] Van Emden, M. H. Increasing the efficiency of Quicksort. *Communications of the ACM* 13 (9), p. 563-567, Sept. 1970.

[26] Van Emden, M. H. Algorithm 402: Increasing the efficiency of Quicksort. *Communications of the ACM* 13 (11), p. 693, Nov. 1970.

[27] Williams, J. W. J. Algorithm 232: Heapsort. *Communications of the ACM* 7 (6), p. 347-348, June 1964.

[28] `http://www.fundao.wiki.br/articles.asp?cod=184`, acessado em 22/01/08.

[29] Ziviani, N. *Projeto de Algoritmos*, 2ed. Pioneira Thomson Learning, São Paulo (1993).