

Desenvolvimento de Programas

MATERIAL DE APOIO

PREPARADO POR ANTONIO A.F. LOUREIRO, <loureiro@dcc.ufmg.br>

VERSÃO DE MARÇO DE 2001

1 O papel do engenheiro de software¹

A evolução da área de Ciência da Computação tem definido o papel do *engenheiro de software* bem como a educação e experiência necessárias para exercer essa profissão. Um engenheiro de software deve naturalmente ser

- um bom programador,
- conhecer bem as diversas estruturas de dados básicas e os algoritmos associados,
- além de conhecer bem pelo menos uma linguagem de programação.

Estes são requisitos para “programming-in-the-small” (ou programação em ponto pequeno), o que representa basicamente a capacidade de desenvolvimento de programas por uma pessoa individualmente. No entanto, um engenheiro de software também participa do desenvolvimento de “programming-in-the-large” (ou programação em ponto grande) que trata do problema de desenvolvimento de software em outro nível.

Do ponto de vista de programação em ponto grande, um engenheiro de software deve

- conhecer diferentes técnicas de projeto,
- ser capaz de expressar requerimentos vagos e imprecisos em especificações precisas e,
- conversar com o usuário de um sistema em termos da aplicação e não em “computês.”

Isto irá demandar um conhecimento da área de aplicação e é, em geral, um dos motivos para tantos fracassos no desenvolvimento de software. Ou seja, a não comunicação e entendimento entre o usuário e o engenheiro de software.

Como foi dito acima, um engenheiro de software é responsável por muitas coisas. Na prática, muitas organizações dividem as responsabilidades entre diversos especialistas com denominações diferentes. Por exemplo, um analista é responsável por definir os requisitos de um sistema enquanto um analista de desempenho é responsável por analisar o desempenho de um sistema.

2 Comentários sobre a área de algoritmos

Os comentários abaixo aparecem na seguinte referência:

Breaking Intractability, por J.F. Traub and H. Wozniakowski, **Scientific American**, Volume 270, Number 1, January 1994.

- ... Even though scientists have computers at their disposal, problems can have so many variables that no future increase in computer speed will make it possible to solve them in a reasonable amount of time.
- ... certain scientific questions can never be answered because the necessary computing resources do not exist in the universe.
- ... An impediment even more serious than intractability may occur: a problem may be unsolvable. A problem is unsolvable if one cannot compute even an approximation at finite cost.

¹Baseado no livro “Fundamentals of Software Engineering” de Carlo Ghezzi, Mehdi Jazayeri e Dino Mandrioli, Prentice-Hall, 1991.

- ... Because these results are intrinsic to the problem, one cannot get around them by inventing other methods.
- ... One possible way to break unsolvability and intractability is through randomization.
- ... Unfortunately, the proof of this result does not tell us what the points and algorithms are, thus leaving a beautiful challenge for the future.
- ... One important achievement of mathematics over the past 60 years is the idea that mathematical problems may be undecidable, noncomputable or intractable. Kurt Gödel proved the first of these results. He established that in a rich mathematical system, such as arithmetic, there are theorems that can never be proved.

3 Um “panaché” de recomendações para o desenvolvimento de programas²

Programas devem ser feitos para serem lidos por seres humanos.

Tenha em mente que seus programas deverão ser lidos e entendidos por outras pessoas (e/ou por você mesmo), de tal forma que possam ser corrigidos, modificados ou receber manutenção.

Escreva os comentários no momento que estiver escrevendo o algoritmo.

Um programa mal documentado é um dos piores erros que um programador pode cometer, e o sinal de um amador (mesmo com 10 anos de experiência).

O melhor momento para se escrever os comentários é aquele em que o programador tem maior intimidade com o algoritmo, ou seja, durante a sua confecção.

Os comentários devem acrescentar alguma coisa de útil, não apenas frasear o código.

O código nos diz *como* um certo problema está sendo resolvido. Os comentários deverão dizer *o que* está sendo feito em cada passo.

Use comentários no prólogo.

É muito importante a colocação de comentários no prólogo de um programa ou de cada módulo, para explicar o que ele faz e fornecer instruções para seu uso.

Poderiam ser colocados comentários dos seguintes tipos:

- o que faz o programa ou módulo;
- como chamá-lo ou utilizá-lo;
- significado dos parâmetros, variáveis de entrada, de saída e variáveis mais importantes;
- arquivos utilizados;
- outros módulos utilizados;
- métodos especiais utilizados, com referências nas quais possa se encontrar mais informações;
- avaliação do tempo de processamento e memória requeridos;
- autor e data de escrita e última atualização;
- etc.

²Baseado em notas de aula preparadas pelo Professor Newton José Vieira que por sua vez se baseou no livro “Program Style, Design, Efficiency, Debugging, and Testing” de D. Van Tassel, segunda edição, Prentice-Hall, 1978.

Utilize espaços em branco para melhorar a legibilidade.

Espaços em branco são valiosos para melhorar a aparência de um programa. Por exemplo:

- deixar uma linha em branco entre as declarações e o corpo do programa;
- deixar uma linha em branco antes e depois dos comentários;
- separar grupos de comandos que executam funções lógicas distintas por uma ou mais linhas em branco;
- utilizar brancos para indicar precedência de operadores; ao invés de $A+B * C$ é bem mais legível a forma $A + B*C$;
- etc.

Escolha nomes representativos.

Os nomes de constantes, tipos, variáveis, procedimentos, funções, etc. devem identificar o melhor possível o que representam. Por exemplo, $X := Y + Z$ é muito menos claro que $Preco := Custo + Lucro$.

Um comando por linha é suficiente.

A utilização de vários comandos por linha é prejudicial por vários motivos, dentre eles destacam-se o fato do programa tornar-se mais ilegível e ficar mais difícil de ser depurado.

Exemplo (Pascal):

```
A:=14.2;for i:=1 to 10 do begin X[i]:=0;k:=i*k;Y[i]:=k;end;
```

O mesmo trecho poderia ser escrito assim:

```
A := 14.2;
for i:=1 to 10 do
begin
  X[i] := 0;
  k := i*k;
  Y[i] := k;
end;
```

Utilize parêntesis para aumentar a legibilidade e prevenir-se contra erros.

Exemplos:

Com poucos parêntesis	Com parêntesis extras
$A*B*C/(D*E*F)$	$(A*B*C)/(D*E*F)$
$A*B/C*D/E*F$	$(A*B*D*F)/(C*E)$
$A/B/C$	$(A/B)/C$
$X >= Y \text{ or } Q$	$(X >= Y) \text{ or } Q$
$A+B < C$	$(A + B) < C$

Utilize “identação” para mostrar a estrutura lógica do programa.

A identação não deve ser feita de forma caótica, mas seguindo padrões razoáveis. Por exemplo, ao invés de:

```
if x < y then if v < w
then j := x
else if x < v then if y < w then j := y
      else j := v
else begin if x < w
then j := w
end else j := x + w;
```

uma forma mais razoável é (existe alguma forma de tornar o seguinte trecho mais simples?):

```

if x < y
then if v < w
    then j := x
    else if x < v
        then if y < w
            then j := y
            else j := v
        else begin
            if x < w
            then j := w
            end
    else j := x + w;

```

Crie algumas regras básicas de indentação e procure segui-las ao escrever um programa.

Lembre-se: a única parte da documentação que vai estar sempre em dia é o código do programa.

O programa deve ser mantido sempre legível, mesmo depois de quaisquer correções e/ou modificações.

Não comece a desenvolver o programa antes que o problema esteja bem definido.

Uma das partes mais importantes da documentação é a especificação do problema usando alguma notação: escrita, gráfica, etc. O ato de *escrever* a especificação do problema é muito importante: ajuda na sua total compreensão e evita o esquecimento de detalhes relevantes.

É preferível realismo no início do que ter que efetuar mudanças quando o programa atinge a “puberdade” (aquele estágio da vida antes da maturidade, quando os fatos da vida tornam-se aparentes).

Após começar o desenvolvimento do programa procure não mudar a especificação do problema.

A única razão para mudar a especificação de um problema após começar o desenvolvimento do programa deve ser para corrigir algum erro de especificação. Se a especificação for mudada constantemente o programa nunca ficará pronto além de acarretar mais custos.

Obviamente, isto não significa que não se deve alterar um programa depois que ele estiver pronto. É muito importante ter em mente que a maior parte dos programas desenvolvidos sofrerá algum tipo de modificação no futuro. Por essa razão, programas devem ser desenvolvidos de tal forma a minimizar o impacto introduzido pelas modificações.

A codificação deve ser feita de forma tão simples quanto possível.

Código complexo, sofisticado ou não usual atrapalha a legibilidade, depuração e modificação.

Estabeleça objetivos realistas o mais cedo possível.

Todo projeto de programação usualmente tem alguns objetivos que devem ser estabelecidos e colocados no papel. Tais objetivos variam naturalmente de projeto para projeto. Alguns itens a serem considerados são:

- data de término;
- facilidade de uso;
- nível de confiabilidade (difícil de ser estimado);
- limites de memória e tempo de execução;
- generalidade;
- etc.

Quando se trabalha em equipe, o estabelecimento explícito dos objetivos irá garantir que todos trabalhem para se alcançar os mesmos objetivos.

Desenvolva cuidadosamente o programa usando alguma técnica de projeto como refinamento sucessivo.

Escrever diretamente qualquer código pode produzir uma barreira psicológica que poderá inibir futuros melhoramentos. Use alguma técnica de desenvolvimento de programas como a técnica de refinamento sucessivo.

Erros encontrados durante o desenvolvimento são relativamente fáceis de serem corrigidos comparados com erros encontrados durante os testes.

Um programa modesto que funciona é mais útil que um super-ambicioso que nunca funciona.

Muitas vezes é tentador fazer um programa com várias opções e melhorias. Mas nada adianta se esse programa nunca fica pronto para ser executado. Lembre-se que você não vai ter todo o tempo do mundo para escrever o seu programa. Por essa razão seja realista ao decidir o que fazer. Uma boa estratégia é ter uma versão simples que funciona e faz o que foi pedido e se houver tempo introduzir novas opções e melhorias.

Selecione os algoritmos cuidadosamente.

Existem algumas regras que você deve procurar seguir ao fazer um programa:

1. Não re programe um programa ou procedimento ou função que está pronto. A solução, no entanto, não é utilizar o primeiro algoritmo disponível. é muito importante que se entenda o módulo escolhido, ou seja, como funciona o algoritmo utilizado, quantidade de tempo e espaço necessários à execução desse módulo, etc.

O que está sendo dito é que ninguém *inventa* todos os algoritmos que precisa utilizar em Ciência da Computação. O importante é saber e entender muito bem os algoritmos necessários à solução do seu problema.

2. Não codifique o primeiro algoritmo que lhe venha à mente. O melhor algoritmo a ser utilizado depende dentre outros fatores da aplicação onde será utilizado. Por essa razão procure conhecer os possíveis algoritmos que podem ser usados na solução do problema. Só assim você terá certeza que escolheu um bom algoritmo ou não.

A bibliografia citada na seção 7 apresenta vários livros na área de projeto de algoritmos.

Escolha a representação dos dados adequada ao problema.

Um boa representação dos dados pode eliminar páginas de código. A representação dos dados deve ser escolhida com base nas *principais operações* que serão feitas.

Veja a bibliografia citada na seção 7 para o estudo de diferentes algoritmos e estruturas de dados.

Utilize uma linguagem de programação adequada.

Se a linguagem de programação não é adequada ao problema a ser resolvido, certamente surgirão problemas na programação, na eficiência e na depuração do programa.

Evite que o programa seja dependente de dados particulares.

Um programa dependente de dados particulares, além de ser mais restrito, dificulta mudanças. Um exemplo típico de dependência de dados é na manipulação de vetores. Por exemplo,

```
for i:=1 to 25 do
  readln(entrada, A[i]);
Soma := 0;
for i:=1 to 25 do
  Soma := Soma + A[i];
```

O trecho de programa acima só funciona para um vetor com 25 elementos. No caso de se desejar manipular um número diferente de elementos, deve-se percorrer o programa modificando cada 25, o que é inconveniente. Além do mais pode-se saltar uma das constantes a ser modificada. E mais, pode-se modificar uma constante com o mesmo valor que não se deveria modificar. Uma solução melhor seria:

```

const N = 25;
. . .
for i:=1 to N do
  readln(entrada, A[i]);
Soma := 0;
for i:=1 to N do
  Soma := Soma + A[i];

```

Procure utilizar constantes, definir tipos, utilizar tipos abstratos de dados para evitar um problema similar ao apresentado acima.

Os formatos de entrada e saída devem ser bem legíveis.

Quando se projeta os formatos de entrada e saída deve-se ter sempre em mente o usuário.

É importante definir uma ordem de variáveis e formatos de dados para entrada que sejam os mais naturais para o usuário. Isto evitará erros e facilitará o uso do programa. Por exemplo, o formato :10 para leitura de quatro variáveis inteiras é bem mais razoável que o formato :7, :8, :9, :8 para as mesmas quatro variáveis inteiras.

A saída deve ser bem legível, sem que haja necessidade de se consultar o código do programa.

A facilidade de uso (entrada) e relatórios atraentes (saída) serão os itens que, em última instância, determinarão o julgamento do usuário quanto à capacidade do programador.

Se o programa não funciona, não interessa sua eficiência.

Um programa super eficiente, mas não confiável, dificilmente pode ser convertido em um programa confiável. Mas um programa confiável e bem estruturado, mesmo que ineficiente, pode ser otimizado.

A legibilidade é, usualmente, mais importante que a eficiência (obviamente que existem exceções). A legibilidade facilita o entendimento, permite possíveis modificações, inclusive a introdução de eficiência.

Antes de otimizar procure saber quão eficiente o programa precisa ser.

A eficiência depende da aplicação, sendo mais relevante em alguns casos do que em outros. Preferencialmente, o nível de eficiência que se quer de um programa deve constar na sua especificação.

Em geral, programas utilizados com muita frequência devem ser mais eficientes do que os utilizados raramente.

4 Confecção dos trabalhos práticos³

Os trabalhos práticos serão corrigidos levando-se em conta os seguintes itens e o que foi dito na seção anterior.

Documentação

A documentação do programa é como um pequeno *artigo* que explica o que o programa faz, como faz, e apresenta conclusões obtidas sobre o trabalho. A documentação é um documento à parte e não deve ser escrito no programa fonte (para uma opção alternativa veja a seção 6).

A documentação a ser entregue deve conter pelo menos:

- *Descrição sucinta sobre o desenvolvimento do trabalho.* Uma explicação sobre as decisões de implementação tomadas, uma visão geral do funcionamento do programa, comentários sobre os testes executados, etc.
- *Descrição dos módulos e sua inter-dependência.* Uma breve descrição de cada módulo bem como um diagrama, por exemplo, mostrando a relação de dependência entre eles.
- *Descrição das estruturas de dados utilizadas.* Uma explicação sobre as estruturas de dados utilizadas, as operações disponíveis e como são implementadas. Você pode fazer essa descrição utilizando desenhos ou escrevendo.
- *Descrição do formato de entrada dos dados.* Uma descrição simples e clara dizendo quais são os dados de entrada e como o programa irá recebê-los. Por exemplo:

³Baseado em notas de um ex-monitor de Ciência da Computação.

“A entrada para o programa consiste de um conjunto de descrição dos edifícios. Em cada linha haverá somente uma descrição. Cada descrição é composta por três números inteiros separados por um ou mais brancos na seguinte ordem: coordenada esquerda do edifício, altura do edifício, coordenada direita do edifício.”

- *Descrição do formato de saída dos dados.* Uma descrição simples e clara dizendo como o programa apresentará os resultados ao usuário. Por exemplo:

“O programa irá gerar uma seqüência de números representado a linha do horizonte. Números que estiverem nas posições ímpares representam coordenadas e números nas posições pares alturas.”

- *Explicação sobre como utilizar o programa* Por exemplo:

“Para executar o programa da linha do horizonte, digite na linha de comando: `lh.exe arqIn arqOut arqErro`, onde `arqIn` é o arquivo que contém os dados de entrada, ...”

- *Estudo da complexidade do programa.* O estudo da complexidade deve analisar os principais procedimentos e/ou funções do programa mostrando qual é a sua complexidade final. (O ideal é que isto seja feito em qualquer situação.) Não se esqueça de especificar o que significa o parâmetro n que aparece nos estudos de complexidade. Por exemplo:

“Seja n o número de cidades. O loop no procedimento `GeraLista` é executado n vezes. Este loop só contém comandos de leitura e atribuição que são $O(1)$. Logo a complexidade deste procedimento é $O(n)$ no pior caso.”

- *Listagem do programa fonte.*
- *Listagem dos testes executados (se for o caso).* A listagem dos testes deve conter os dados recebidos pelo programa (dados de entrada) e os resultados apresentados (dados de saída).

Programa fonte

Procure observar os seguintes aspectos no seu programa fonte:

- *Modularidade.* Não faça programas de um módulo só. Divida as tarefas a serem executadas em módulos.
- *Comentários.* Veja o item “Comentários” na seção que descreve o panaché.
- *Identação.* Veja o item “Identação” na seção que descreve o panaché.
- *Passagem de parâmetros.* Procure ser consistente na ordem e no tipo de passagem de parâmetros. Por exemplo, suponha que três procedimentos diferentes têm que acessar os dados de uma mesma tabela. Não faz sentido se um dos procedimentos receber a tabela por valor, o outro por referência e o último acessar a tabela como variável global.
- *Variáveis globais.* Evite ao máximo a utilização de variáveis globais, por que elas compartilham dados entre as diversas partes do programa de uma maneira que não é explícita, o que pode levar a erros difíceis de serem achados.
- *Nomes de variáveis.* Veja o item “Nomes de variáveis” na seção que descreve o panaché.

Testes

Procure fazer testes relevantes como por exemplo casos de exceções.

Apresentação

A apresentação do trabalho é importante. **Hoje**, o usuário ou cliente do seu TRABALHO é o professor ou o monitor do curso. **Amanhã** será o seu chefe ou um cliente para o qual você estará desenvolvendo um produto.

Coloque toda a documentação num só documento, utilizando papel do mesmo tipo, grampeado e sem a rimalina (“beiradas” que servem para guiar o papel na impressora). Verifique se a impressora está funcionando bem, com a fita no lugar. Nunca rasure a listagem do programa ou os resultados de testes. Desenhos das estruturas de dados, títulos, comentários adicionados à documentação, etc, podem ser feitos a mão. No entanto, é recomendável que você utilize um editor e/ou formatador de textos.

Ao entregar a documentação do programa junto com o disquete, entregue tudo dentro de um saco plástico ou envelope para evitar que se perca o disquete. Perda de qualquer material fora do especificado acima não será de responsabilidade do professor.

Comentários finais

Se você utiliza a rede do DCC, não se esqueça de deixar sempre acessível o diretório da disciplina do seu *home directory* para leitura (r) e execução (x) tanto para usuários que fazem parte do seu grupo (grupo grad, do qual o monitor também faz parte) como para outros usuários (outros, do qual o professor faz parte).

5 Documentação e qualidade⁴

Em geral, a documentação de um produto é a parte do “produto como um todo” que recebe menos atenção. No entanto, pesquisas feitas na última década com vários clientes mostrou que a *qualidade da documentação* tornou-se um fator de fundamental importância no sucesso de um produto. Melhorias na qualidade da documentação devem ser feitas não somente no material que o cliente recebe, isto é, na descrição do produto e manuais de uso e/ou operação, treinamento e manutenção, mas também na documentação técnica do produto que é usada durante todo o ciclo de seu desenvolvimento, isto é, especificação, projeto e atividades de produção e manufatura.

Por que a documentação *merece* uma prioridade maior na estratégia de desenvolvimento de produtos em qualquer empresa?

A primeira razão se deve a complexidade cada vez maior dos produtos produzidos. A habilidade de projetar e desenvolver um produto e transferir a tecnologia (informação) de um grupo funcional numa empresa para um outro grupo funcional (dentro da mesma empresa ou não) é um fator chave no sucesso de desenvolvimento de um produto.

A segunda razão é a possibilidade de se reutilizar componentes de um produto. Isto só será possível se houver uma documentação adequada de módulos do sistema. Naturalmente, a documentação não é o meio em si para obter uma maior reutilização que só é conseguida através de uma prática adequada de desenvolvimento.

Imagine qual seria a **SUA** reação ao usar um produto (hardware ou software) se a documentação desse produto estivesse pessimamente escrita, difícil de entender, desatualizada ou incompleta.

Fazer uma boa documentação é uma prática que se aprende com o tempo. Por isso, comece a praticar desde já.

6 Literate programming⁵

Literate programming is the combination of documentation and source together in a fashion suited for reading by human beings. In general, literate programs combine source and documentation in a single file. Literate programming tools then parse the file to produce either readable documentation or compilable source. The WEB style of literate programming was created by D.E. Knuth during the development of his T_EX typesetting software.

All the original work revolves around a particular literate programming tool called WEB. Knuth says:

The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like T_EX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a web that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbours. The typographic tools provided by T_EX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages such as C or Fortran make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

⁴Baseado no artigo “Changes in the Documentation Engineering Process,” G. Brooks, **Electrical Communication**, 2nd Quarter 1994.

⁵Retirado do artigo “Frequently Asked Questions” do newsgroup comp.programming.literate.

7 Referências Bibliográficas

- Projeto e Análise de Algoritmos. Nivio Ziviani. Livros Técnicos e Científicos.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. MIT Press, Hardcover, Published June 1990, 1028 pages, ISBN 0262031418. Livro muito bom que cobre os principais aspectos de projeto e análise de algoritmos.
- The Design and Analysis of Computer Algorithms. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Addison-Wesley, Hardcover, Published June 1974, 470 pages, ISBN 0201000296. Livro clássico de projeto e análise de algoritmos.
- Data Structures and Algorithms. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Addison-Wesley, Hardcover, Published January 1983, ISBN 0201000237.
- Foundations of Computer Science. Alfred V. Aho, Jeffrey D. Ullman W.H. Freeman, Hardcover, Published January 1995, ISBN 0716782847.
- Computer Algorithms: Introduction to Design and Analysis. Sara Baase, Allen van Gelder. Addison-Wesley, Hardcover, Published November 1999, 688 pages, ISBN 0201612445.
- Introduction to Algorithms: A Creative Approach Udi Manber. Addison-Wesley, Hardcover, Published March 1989, 478 pages, ISBN 0201120372
- The Art of Computer Programming, Volumes 1-3 Boxed Set. Donald E. Knuth. Addison-Wesley, Hardcover slipcase edition, Published October 1998, ISBN 0201485419. Clássico da área de projeto e análise de algoritmos: Vol 1: Fundamental Algorithms, Third Edition, 656 pages. Vol 2: Seminumerical Algorithms, Third Edition, 704 pages. Vol 3: Sorting and Searching, Second Edition, 736 pages.
- An Introduction to the Analysis of Algorithms. Robert Sedgewick, Philippe Flajolet. Addison-Wesley, Hardcover, Published June 1996, 512 pages, ISBN 020140009X.
- Algorithms in ... Robert Sedgewick. Addison-Wesley. Mesmo livro em diferentes linguagens de programação.
- Data Structures, Algorithms, and Software Principles in C. Thomas A. Standish. Addison-Wesley, Hardcover, Published January 1995, 748 pages, ISBN 0201591189.
- Concrete Mathematics: A Foundation for Computer Science, 2nd Edition. Ronald Graham, Oren Patashnik, Donald E. Knuth. Addison Wesley, Hardcover, Published March 1994, 657 pages, ISBN 0201558025. Apresenta várias técnicas matemáticas usadas na análise de algoritmos.
- The Algorithm Design Manual. Steven S. Skiena, Steve Skiena. Springer Verlag, Hardcover, Published November 1997, 496 pages. Bk&Cd Rom edition, ISBN 0387948600.
- Combinatorial Algorithms: Generation, Enumeration, and Search. Donald L. Kreher, Douglas R. Stinson. CRC Press, Hardcover, Published December 1998, 300 pages, ISBN 084933988X.
- Information Retrieval: Data Structures and Algorithms. William Frakes, Richardo Baeza-Yates (editores). Prentice-Hall, Textbook binding, Published June 1992, 464 pages, ISBN 0134638379.
- Parallel Computing: Theory and Practice. Michael J. Quinn. McGraw-Hill, Hardcover, Published September 1993, ISBN 0070512949.
- Fundamentals of Computer Algorithms. Ellis Horowitz, Sahni Sartaj. W.H. Freeman, Hardcover, Published June 1978, ISBN 0716780453.
- Handbook of Algorithms and Data Structures in Pascal and C, 2nd Edition. Gaston Gonnet, Ricardo Baeza-Yates. Addison Wesley, Paperback, Published May 1991, ISBN 0201416077
- Writing Efficient Programs. Jon Louis Bentley. Prentice-Hall.