Projeto de Algoritmos*
Introdução

Última alteração: 30 de Agosto de 2010

*Transparências elaboradas por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Conteúdo do Capítulo

Algoritmos, Estruturas de Dados e Programas

- Os algoritmos fazem parte do dia-a-dia das pessoas. Exemplos de algoritmos:
  - instruções para o uso de medicamentos,
  - indicações de como montar um aparelho,
  - uma receita de culinária.
- Sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.
- Segundo Dijkstra, um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.
  - Executando a operação $a + b$ percebemos um padrão de comportamento, mesmo que a operação seja realizada para valores diferentes de a e b.

Estruturas de dados

- Estruturas de dados e algoritmos estão intimamente ligados:
  - não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas,
  - assim como a escolha dos algoritmos em geral depende da representação e da estrutura dos dados.
- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa a situação real.
- A seguir, deve ser escolhida a forma de representar esses dados.

Escolha da Representação dos Dados

- A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.
- Considere a operação de adição:
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é
  - Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.
  - Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo da posição de cada dígito).

Programas

- Programar é basicamente estruturar dados e construir algoritmos.
- Programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados.
- Programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.
- Um computador só é capaz de seguir programas em linguagem de máquina (sequência de instruções obscuras e desconfortáveis).
- É necessário construir linguagens mais adequadas, que facilitem a tarefa de programar um computador.
- Uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

Tipos de Dados

- Caracteriza o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função.
- Tipos simples de dados são grupos de valores indivisíveis (como os tipos básicos *integer, boolean, char* e *real* do Pascal).
  - Exemplo: uma variável do tipo *boolean* pode assumir o valor verdadeiro ou o valor falso, e nenhum outro valor.
- Os tipos estruturados em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.

Tipos Abstratos de Dados (TAD)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TADs são utilizados como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.
- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma estrutura de dados.
- Podemos considerar TADs como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados. A definição do tipo e todas as operações ficam localizadas numa seção do programa.

Implementação de TADs (1)

- Considere uma uma lista de inteiros. Poderíamos definir TAD Lista, com as seguintes operações:
  1. faça a lista vazia;
  2. obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorne nulo;
  3. insira um elemento na lista.
- Há várias opções de estruturas de dados que permitem uma implementação eficiente para listas (por ex., o tipo estruturado arranjo).

Implementação de TADs (2)

- Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida.
- Qualquer alteração na implementação do TAD fica restrita à parte encapsulada, sem causar impactos em outras partes do código.
- Cada conjunto diferente de operações define um TAD diferente, mesmo que atuem sob um mesmo modelo matemático.
- A escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.

Medida do Tempo de Execução de um Programa

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de tempo de execução e espaço ocupado.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo particular**.
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Características que devem ser investigadas:
    - análise do número de vezes que cada parte do algoritmo deve ser executada,
    - 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 é investigada.
  - Procura-se identificar um que seja o melhor possível.
  - Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um Algoritmo

- Determinando o menor custo possível para resolver problemas de uma classe, temos a medida da dificuldade inerente para resolver o problema.
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.
- Podem existir vários algoritmos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Medida do Custo pela Execução do Programa

- Tais medidas são inadequadas e os resultados jamais devem ser generalizados:
  - os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - os resultados dependem do *hardware*;
  - quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender desse aspecto.
- Apesar disso, há argumentos a favor de medidas reais de tempo.
  - Ex.: quando há vários algoritmos para resolver um mesmo tipo de problema, todos com um custo de execução dentro da mesma ordem de grandeza.
  - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

Medida do Custo por meio de um Modelo Matemático

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto e ignoramos operações aritméticas, de atribuição e manipulações de índices, entre outras.

Função de Complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f.
- $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.
- Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n.
- Função de **complexidade de espaço**: $f(n)$ mede a memória necessária para executar um algoritmo em um problema de tamanho n.
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente.
- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Exemplo - Maior Elemento (1)

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[1..n], n \geq 1$.

```
function Max (var A: TipoVetor): integer;
var i, Temp: integer;
begin
  Temp := A[1];
  for i := 2 to N do if Temp < A[i] then Temp := A[i];
  Max := Temp;
end;
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos.
- Logo $f(n) = n - 1, \text{ para } n > 0.$
- Vamos provar que o algoritmo apresentado é **ótimo**.

Exemplo - Maior Elemento (2)

- **Teorema**: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.
- **Prova**: Cada um dos $n - 1$ elementos tem de ser mostrado, por meio de comparações, que é menor do que algum outro elemento.
- Logo $n - 1$ comparações são necessárias. $\square$
- O teorema diz que, se o número de comparações for utilizado como medida de custo, então a função Max é ótima.

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
- No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Melhor Caso, Pior Caso e Caso Médio (1)

- **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $f(n)$.
- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

Melhor Caso, Pior Caso e Caso Médio (2)

- Na análise do caso esperado, uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n é suposta e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.
- Na prática isso nem sempre é verdade.

Exemplo - Registros de um Arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo.
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
- O algoritmo mais simples é o que faz a **pesquisa sequencial**.
- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - melhor caso: $f(n) = 1$ (registro procurado é o primeiro consultado);
  - pior caso: $f(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo);
  - caso médio: $f(n) = (n + 1)/2$.

Exemplo - Registros de um Arquivo

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e para recuperar o i-ésimo registro são necessárias i comparações, então
  $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$.
- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n, 1 \leq i \leq n$.
- Nesse caso $f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2} \cdot$
- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Exemplo - Maior e Menor Elemento (1)

- Encontrar o maior e o menor elemento de $A[1..n], n \geq 1$.
- Um algoritmo simples pode ser derivado do algoritmo para achar o maior elemento.

```
function Max (var A: TipoVetor): integer;
var i, Temp: integer;
begin
  Temp := A[1];
  for i := 2 to N do if Temp < A[i] then Temp := A[i];
  Max := Temp;
end;
```

- Seja $f(n)$ o número de comparações entre os n elementos de As.
- $f(n) = 2(n - 1), \text{ para } n > 0$, no melhor caso, pior caso e caso médio.

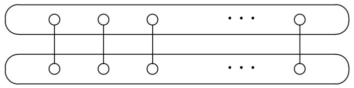
Exemplo - Maior e Menor Elemento (2)

```
procedure MaxMin2 (var A: TipoVetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to N do
    if A[i] > Max then Max := A[i] else if A[i] < Min then Min := A[i];
end;
```

- MaxMin1 pode ser facilmente melhorado: a comparação $A[i] <$ Min só é necessária quando a comparação $A[i] >$ Max dá falso.
- Para a nova implementação temos:
  - melhor caso: $f(n) = n - 1$ (elementos estão em ordem crescente);
  - pior caso: $f(n) = 2(n - 1)$ (elementos estão em ordem decrescente);
  - caso médio: $f(n) = 3n/2 - 3/2$.
- No caso médio, $A[i]$ é maior do que Max a metade das vezes.
- Logo $f(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}$, para $n > 0$.

Exemplo - Maior e Menor Elemento (3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1) Compare os elementos de A aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de $\lceil n/2 \rceil$ comparações.
  2) O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.
  3) O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.



Exemplo - Maior e Menor Elemento (3)

```
procedure MaxMin3 (var A: TipoVetor; var Max, Min: integer);
var i, FimDoAnel: integer;
begin
  if (N mod 2) > 0
  then begin A[N+1] := A[N]; FimDoAnel := N; end else FimDoAnel := N–1;
  if A[1] > A[2]
  then begin Max := A[1]; Min := A[2];  end
  else begin Max := A[2]; Min := A[1];  end;
  i := 3;
  while i <= FimDoAnel do
    begin if A[i] > A[i+1]
    then begin if A[i] > Max then Max := A[i];
               if A[i+1] < Min then Min := A[i+1]; end
    else begin if A[i] < Min then Min := A[i];
               if A[i+1] > Max then Max := A[i+1]; end;
    i := i + 2;
    end;
end;
```


Exemplo - Maior e Menor Elemento (3)

- Os elementos de A são comparados dois a dois e os maiores são comparados com *Max* e os menores com *Min*.
- Quando n é ímpar, o elemento que está na posição $A[n]$ é duplicado na posição $A[n+1]$ para evitar um tratamento de exceção.
- Para esta implementação, $f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Comparação entre MaxMin1, MaxMin2 e MaxMin3

- A tabela apresenta o número de comparações dos programas MaxMin1, MaxMin2 e MaxMin3.
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Os três	$f(n)$		
algoritmos	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

Limite Inferior - Uso de um Oráculo

- Existe possibilidade de obter um algoritmo MaxMin mais eficiente?
- Para responder temos de conhecer o **limite inferior** para essa classe de algoritmos.
- Técnica muito utilizada: uso de um oráculo.
- Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação).
- Para derivar o limite inferior, o oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível necessário para determinar a resposta final.

Exemplo de Uso de um Oráculo

- **Teorema**: Qualquer algoritmo para encontrar o maior e o menor elemento de um conjunto com n elementos não ordenados, $n \geq 1$, faz pelo menos $\lceil 3n/2 \rceil - 2$ comparações.
- **Prova**: Define um oráculo que descreve o comportamento do algoritmo por meio de um conjunto de n–tuplas, mais um conjunto de regras que mostram as tuplas possíveis (estados) que um algoritmo pode assumir a partir de uma dada tupla e uma única comparação.
- Uma 4–tupla, representada por (a, b, c, d), onde os elementos de:
  - $a \rightarrow$ nunca foram comparados;
  - $b \rightarrow$ foram vencedores e nunca perderam em comparações realizadas;
  - $c \rightarrow$ foram perdedores e nunca venceram em comparações realizadas;
  - $d \rightarrow$ foram vencedores e perdedores em comparações realizadas.

Exemplo de Uso de um Oráculo

- O algoritmo inicia no estado $(n, 0, 0, 0)$ e termina com $(0, 1, 1, n-2)$.
- Após cada comparação a tupla (a, b, c, d) consegue progredir apenas se ela assume um dentre os seis estados possíveis abaixo:
  - $(a-2, b+1, c+1, d)$ se $a \geq 2$ (dois elementos de a são comparados)
  - $(a-1, b+1, c, d)$ ou $(a-1, b, c+1, d)$ ou $(a-1, b, c, d+1)$ se $a \geq 1$ (um elemento de a comparado com um de b ou um de c)
  - $(a, b-1, c, d+1)$ se $b \geq 2$ (dois elementos de b são comparados)
  - $(a, b, c-1, d+1)$ se $c \geq 2$ (dois elementos de c são comparados)
  - O primeiro passo requer necessariamente a manipulação do componente a.
  - O caminho mais rápido para levar a até zero requer $\lceil n/2 \rceil$ mudanças de estado e termina com a tupla $(0, n/2, n/2, 0)$ (por meio de comparação dos elementos de a dois a dois).

Exemplo de Uso de um Oráculo

- A seguir, para reduzir o componente b até 1 são necessárias $\lceil n/2 \rceil - 1$ mudanças de estado (mínimo de comparações necessárias para obter o maior elemento de b).
- Idem para c, com $\lceil n/2 \rceil - 1$ mudanças de estado.
- Logo, para obter o estado $(0, 1, 1, n-2)$ a partir do estado $(n, 0, 0, 0)$ são necessárias

$$\lceil n/2 \rceil + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1 = \lceil 3n/2 \rceil - 2$$

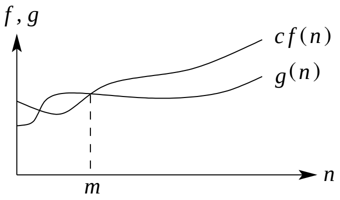
  comparações. □
- O teorema nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo, então o algoritmo MaxMin3 é **ótimo**.

Comportamento Assintótico de Funções

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de n.
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de n)
- O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.

Dominação assintótica

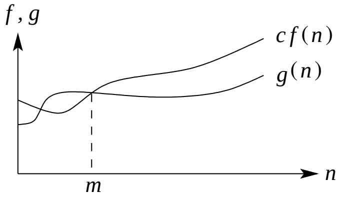
- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **Definição**: Uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$.



- Sejam $g(n) = (n+1)^2$ e $f(n) = n^2$.
- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, desde que $|(n+1)^2| \leq 4|n^2|$ para $n \geq 1$ e $|n^2| \leq |(n+1)^2|$ para $n \geq 0$.

Notação O

- Escrevemos $g(n) = O(f(n))$ para expressar que $f(n)$ domina assintoticamente $g(n)$. Lê-se $g(n)$ é da ordem no máximo $f(n)$.
- Exemplo: quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, significa que existem constantes c e m tais que, para valores de $n \geq m$, $T(n) \leq cn^2$.
- Exemplo gráfico de dominação assintótica que ilustra a notação O.



O valor da constante m é o menor possível, mas qualquer valor maior é válido.

- **Definição**: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq cf(n)$, para todo $n \geq m$.

Exemplos de Notação O

- **Exemplo**: $g(n) = (n+1)^2$.
  - Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$.
  - Isto porque $(n+1)^2 \leq 4n^2$ para $n \geq 1$.
- **Exemplo**: $g(n) = n$ e $f(n) = n^2$.
  - Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
  - Entretanto $f(n)$ não é $O(n)$.
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq cn$.
  - Logo $c \geq n$ para qualquer $n \geq m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

Exemplos de Notação O

- **Exemplo**: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.
- **Exemplo**: $g(n) = \log_5 n$ é $O(\log n)$.
  - O $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$.
  - Como $n = c^{\log_c n}$, tomando o logaritmo base b em ambos os lados da igualdade, temos que $\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c$.

Operações com a Notação O

$$\boxed{\begin{array}{rcll}
f(n) & = & O(f(n)) & \\
c \times O(f(n)) & = & O(f(n)) & c = constante \\
O(f(n)) + O(f(n)) & = & O(f(n)) & \\
O(O(f(n))) & = & O(f(n)) & \\
O(f(n)) + O(g(n)) & = & O(max(f(n), g(n))) & \\
O(f(n))O(g(n)) & = & O(f(n)g(n)) & \\
f(n)O(g(n)) & = & O(f(n)g(n)) &
\end{array}}$$

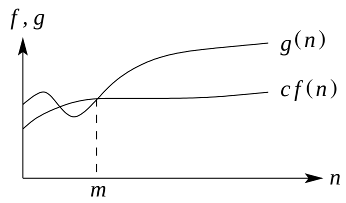
Exemplo: regra da soma $O(f(n)) + O(g(n))$.

- Suponha três trechos cujos tempos são $O(n)$, $O(n^2)$ e $O(n \log n)$.
- O tempo de execução dos dois primeiros é $O(max(n, n^2))$, que é $O(n^2)$.
- O tempo dos três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$.

Exemplo: O produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é $n \log n + kn + O(\sqrt{n} \log n)$.

Notação Ω

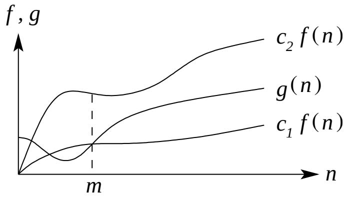
- Especifica um limite inferior para $g(n)$.
- **Definição**: Uma função $g(n)$ é $\Omega(f(n))$ se existirem duas constantes c e m tais que $g(n) \geq cf(n)$, para todo $n \geq m$.
- **Exemplo**: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.
- **Exemplo**: Seja $g(n) = n$ para n ímpar $(n \geq 1)$ e $g(n) = n^2/10$ para n par $(n \geq 0)$. Nesse caso $g(n)$ é $\Omega(n^2)$, para $c = 1/10$ e $n = 0, 2, 4, 6, \ldots$



Para todos os valores à direita de m, o valor de $g(n)$ está sobre ou acima do valor de $cf(n)$.

Notação Θ

- **Definição**: Uma função $g(n)$ é $\Theta(f(n))$ se existirem constantes c_1, c_2 e m tais que $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$.
- Exemplo gráfico para a notação Θ



- Dizemos que $g(n) = \Theta(f(n))$ se existirem constantes c_1, c_2 e m tais que, para todo $n \geq m$, o valor de $g(n)$ está sobre ou acima de $c_1 f(n)$ e sobre ou abaixo de $c_2 f(n)$.
- Para todo $n \geq m$, $g(n)$ é igual a $f(n)$ a menos de uma constante.
- Nesse caso, $f(n)$ é um **limite assintótico firme**.

Exemplo de Notação Θ

- Seja $g(n) = n^2/3 - 2n$. Vamos mostrar que $g(n) = \Theta(n^2)$.
- Temos de obter c_1, c_2 e m tais que $c_1 n^2 \leq \frac{1}{3}n^2 - 2n \leq c_2 n^2$ para todo $n \geq m$.
- Dividindo por n^2 leva a $c_1 \leq \frac{1}{3} - \frac{2}{n} \leq c_2$.
- O lado direito da desigualdade será sempre válido para qualquer valor de $n \geq 1$ quando escolhemos $c_2 \geq 1/3$.
- Escolhendo $c_1 \leq 1/21$, o lado esquerdo da desigualdade será válido para qualquer valor de $n \geq 7$.
- Logo, escolhendo $c_1 = 1/21$, $c_2 = 1/3$ e $m = 7$, verifica-se que $n^2/3 - 2n = \Theta(n^2)$.
- Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

Notação o

- Usada para definir um limite superior que não é assintoticamente firme.
- **Definição**: Uma função $g(n)$ é $o(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq g(n) < cf(n)$ para todo $n \geq m$.
- **Exemplo**: $2n = o(n^2)$, mas $2n^2 \neq o(n^2)$.
- Em $g(n) = O(f(n))$, a expressão $0 \leq g(n) \leq cf(n)$ é válida para alguma constante $c > 0$, mas em $g(n) = o(f(n))$, a expressão $0 \leq g(n) < cf(n)$ é válida para todas as constantes $c > 0$.
- Na notação o, a função $g(n)$ tem um crescimento muito menor que $f(n)$ quando n tende para infinito.
- Alguns autores usam $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ para a definição da notação o.

Notação ω

- Por analogia, a notação ω está relacionada com a notação Ω da mesma forma que a notação o está relacionada com a notação O.
- **Definição**: Uma função $g(n)$ é $\omega(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq cf(n) < g(n)$ para todo $n \geq m$.
- **Exemplo**: $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$.
- A relação $g(n) = \omega(f(n))$ implica $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$, se o limite existir.

Classes de Comportamento Assintótico

- Se f é uma **função de complexidade** para um algoritmo F, então $O(f)$ é considerada a **complexidade assintótica** do algoritmo F.
- A relação de dominação assintótica permite comparar funções de complexidade.
- Entretanto, se as funções f e g dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
- Nesses casos, o comportamento assintótico não serve para comparar os algoritmos.
- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$.
- Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante.

Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.
- Um programa com tempo $O(n)$ é melhor que outro com tempo $O(n^2)$.
- Porém, as constantes de proporcionalidade podem alterar esta consideração.
- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
  - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possúi tempo $100n$.
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Principais Classes de Problemas

- $f(n) = O(1)$.
  - Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.
  - Uso do algoritmo independe de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.
- $f(n) = O(\log n)$.
  - Um algoritmo de complexidade $O(\log n)$ é dito ter **complexidade logarítmica**.
  - Típico em algoritmos que transformam um problema em outros menores.
  - Pode-se considerar o tempo de execução como menor que uma constante grande.
  - Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.
  - Para dobrar o valor de $\log n$ temos de considerar o quadrado de n.
  - A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Principais Classes de Problemas

- $f(n) = O(n)$.
  - Um algoritmo de complexidade $O(n)$ é dito ter **complexidade linear**.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.
  - Cada vez que n dobra de tamanho, o tempo de execução dobra.
- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.
  - Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões.
  - Quando n é 2 milhões, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Principais Classes de Problemas

- $f(n) = O(n^2)$.
  - Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**.
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
  - Quando n é mil, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução é multiplicado por 4.
  - Úteis para resolver problemas de tamanhos relativamente pequenos.
- $f(n) = O(n^3)$.
  - Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**.
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Principais Classes de Problemas

- $f(n) = O(2^n)$.
  - Um algoritmo de complexidade $O(2^n)$ é dito ter **complexidade exponencial**.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
  - Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.
- $f(n) = O(n!)$.
  - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa **força bruta** na solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40 \rightarrow$ um número com 48 dígitos.

Comparação de Funções de Complexidade (1)

Função de custo	Tamanho n					
	10	20	30	40	50	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	0,0.35 s	0,0036 s
n^3	0,001 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.

Comparação de Funções de Complexidade (2)

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Algoritmos Polinomiais × Algoritmos Exponenciais

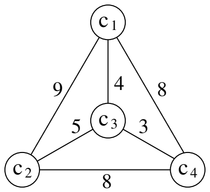
- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n), c > 1$.
- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
- Os algoritmos polinomiais são mais úteis na prática que os exponenciais.
- Algoritmos exponenciais são geralmente variações de pesquisa exaustiva.
- Algoritmos polinomiais são geralmente obtidos mediante melhor entendimento da estrutura do problema.
- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.

Algoritmos Polinomiais × Algoritmos Exponenciais

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- Exemplo: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.
- Também existem algoritmos exponenciais que são muito úteis na prática.
- Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Exemplo de Algoritmo Exponencial

- Um **caixeiro viajante** deseja visitar n cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.
- A figura ilustra o exemplo para quatro cidades c_1, c_2, c_3, c_4, em que os números nos arcos indicam a distância entre duas cidades.



- O percurso $< c_1, c_3, c_4, c_2, c_1 >$ é uma solução para o problema, cujo percurso total tem distância 24.

Exemplo de Algoritmo Exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há $(n - 1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é $n!$.
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria $50! \approx 10^{64}$.
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.
- O problema do caixeiro viajante aparece com frequência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.

Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo;
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas,
  - produtos,
  - permutações,
  - fatoriais,
  - coeficientes binomiais,
  - solução de **equações de recorrência**.

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: $O(1)$.
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é $O(1)$.
- Anel: soma do tempo do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente $O(1)$), multiplicado pelo número de iterações.
- **Procedimentos não recursivos**: cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos. Avalia-se então os que chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos**: associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos.

Procedimento não Recursivo

Algoritmo para ordenar os n elementos de um conjunto A em ordem ascendente.

```
    procedure Ordena (var A: TipoVetor);
    { ordena o vetor A em ordem ascendente }
    var  i, j, min, x: integer;
    begin
(1)   for i := 1 to n-1 do
        begin
(2)     min := i;
(3)     for j := i+1 to n do
(4)       if A[j] < A[min]
(5)       then min := j;
        { troca A[min] e A[i] }
(6)     x := A[min];
(7)     A[min] := A[i];
(8)     A[i] := x;
        end;
    end;
```

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento $A[1]$.
- Repita as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$, até que reste apenas um.

Análise do Procedimento não Recursivo

Anel Interno

- Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
- Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que serSS sempre executado.
- O tempo para incrementar o índice do anel e avaliar sua condição de terminação é $O(1)$.
- O tempo combinado para executar uma vez o anel é $O(max(1, 1, 1)) = O(1)$, conforme regra da soma para a notação O.
- Como o número de iterações é $n - i$, o tempo gasto no anel é $O((n - i) \times 1) = O(n - i)$, conforme regra do produto para a notação O.

Análise do Procedimento não Recursivo

Anel Externo

- Contém, além do anel interno, quatro comandos de atribuição. $O(max(1,(n-i),1,1,1)) = O(n-i)$.
- A linha (1) é executada $n-1$ vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de $(n-i)$: $\sum_1^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$
- Considerarmos o número de comparações como a medida de custo relevante, o programa faz $(n^2)/2 - n/2$ comparações para ordenar n elementos.
- Considerarmos o número de trocas, o programa realiza exatamente $n-1$ trocas.

Procedimento Recursivo

```
     Pesquisa(n);
(1)  if n ≤ 1
(2)  then 'inspecione elemento' e termine
     else begin
(3)         para cada um dos n elementos 'inspecione elemento';
(4)         Pesquisa(n/3);
            end;
```

- Para cada procedimento recursivo é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento.
- Obtemos uma equação de recorrência para $f(n)$.
- **Equação de recorrência**: maneira de definir uma função por uma expressão envolvendo a mesma função.

Análise do Procedimento Recursivo

- Seja $T(n)$ uma função de complexidade que represente o número de inspeções nos n elementos do conjunto.
- O custo de execução das linhas (1) e (2) é $O(1)$ e o da linha (3) é exatamente n.
- Usa-se uma **equação de recorrência** para determinar o nº de chamadas recursivas.
- O termo $T(n)$ é especificado em função dos termos anteriores $T(1)$, $T(2)$, ..., $T(n-1)$.
- $T(n) = n + T(n/3), \;\; T(1) = 1$ (para $n = 1$ fazemos uma inspeção)
- Por exemplo, $T(3) = T(3/3) + 3 = 4$, $T(9) = T(9/3) + 9 = 13$, e assim por diante.
- Para calcular o valor da função seguindo a definição são necessários $k-1$ passos para computar o valor de $T(3^k)$.

Exemplo de Resolução de Equação de Recorrência

- Sustitui-se os termos $T(k)$, $k < n$, até que todos os termos $T(k)$, $k > 1$, tenham sido substituídos por fórmulas contendo apenas $T(1)$.

$$\begin{aligned} T(n) &= n + T(n/3) \\ T(n/3) &= n/3 + T(n/3/3) \\ T(n/3/3) &= n/3/3 + T(n/3/3/3) \\ &\vdots \\ T(n/3/3\cdots/3) &= n/3/3\cdots/3 + T(n/3\cdots/3) \end{aligned}$$

- Adicionando lado a lado, temos $T(n) = n + n\cdot(1/3) + n\cdot(1/3^2) + n\cdot(1/3^3) + \cdots + (n/3/3\cdots/3)$ que representa a soma de uma série geométrica de razão $1/3$, multiplicada por n, e adicionada de $T(n/3/3\cdots/3)$, que é menor ou igual a 1.

Exemplo de Resolução de Equação de Recorrência

$T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots +$
$+ (n/3/3 \cdots /3)$

- Se desprezarmos o termo $T(n/3/3 \cdots /3)$, quando n tende para infinito, então $T(n) = n \sum_{i=0}^{\infty}(1/3)^i = n\left(\frac{1}{1-\frac{1}{3}}\right) = \frac{3n}{2}$.
- Se considerarmos o termo $T(n/3/3/3 \cdots /3)$ e denominarmos x o número de subdivisões por 3 do tamanho do problema, então $n/3^x = 1$, e $n = 3^x$. Logo $x = \log_3 n$.
- Lembrando que $T(1) = 1$ temos $T(n) = \sum_{i=0}^{x-1} \frac{n}{3^i} + T(\frac{n}{3^x}) = n \sum_{i=0}^{x-1}(1/3)^i + 1 = \frac{n(1-(\frac{1}{3})^x)}{(1-\frac{1}{3})} + 1 = \frac{3n}{2} - \frac{1}{2}$.
- Logo, o programa do exemplo é $O(n)$.

A Linguagem de Programação Pascal (1)

- Os programas apresentados no livro usam apenas as características básicas do Pascal.
- São evitadas as facilidades mais avançadas disponíveis em algumas implementações.
- Não apresentaremos a linguagem na sua totalidade, apenas examinamos algumas características.

A Linguagem de Programação Pascal (2)

- As várias partes componentes de um programa Pascal são:

program	cabeçalho do programa
label	declaração de rótulo para **goto**
const	definição de constantes
type	definição de tipos de dados
var	declaração de variáveis
procedure ou **function**	declaração de subprogramas
begin	
⋮	comandos do programa
end	

Tipos em Pascal

- Regra geral: tornar explícito o tipo associado quando se declara uma constante, variável ou função.
- Isso permite testes de consistência durante o tempo de compilação.
- A definição de tipos permite ao programador alterar o nome de tipos existentes e criar um número ilimitado de outros tipos.
- No caso do Pascal, os tipos podem ser:
  - *simples*,
  - *estruturados* e
  - *apontadores*.

Tipos Simples (1)

- São grupos de valores indivisíveis (integer, boolean, char e real).
- Tipos simples adicionais podem ser enumerados por meio de:
  - listagem de novos grupos de valores; Exemplo:

    type cor = (vermelho, azul, rosa);
    ⋮
    var c : cor;
    ⋮
    c := rosa;

Tipos Simples (2)

- indicação de subintervalos. Exemplo:

  type ano = 1900..1999;
  type letra = 'A'..'Z';
  ⋮
  var a : ano;
  var b : letra;

  as atribuições a:=1986 e b:='B' são possíveis, mas a:=2001 e b:=7 não o são.

Tipos Estruturados

- Definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.
- Existem quatro tipos estruturados primitivos:
  - **Arranjos**: tabela n-dimensional de valores homogêneos de qualquer tipo. Indexada por um ou mais tipos simples, exceto o tipo real.
  - **Registros**: união de valores de tipos quaisquer, cujos campos podem ser acessados pelos seus nomes.
  - **Conjuntos**: coleção de todos os subconjuntos de algum tipo simples, com operadores especiais $*$ (interseção), $+$ (união), $-$ (diferença) e in (pertence a) definidos para todos os tipos conjuntos.
  - **Arquivos**: sequência de valores homogêneos de qualquer tipo. Geralmente é associado com alguma unidade externa.

Tipo Estruturado Arranjo - Exemplo

type cartão = **array** [1..80] **of** char;
type matriz = **array** [1..5, 1..5] **of** real;
type coluna = **array** [1..3] **of** real;
type linha = **array** [ano] **of** char;
type alfa = **packed array** [1..n] **of** char;
type vetor = **array** [1..n] **of** integer;

A constante n deve ser previamente declarada

const n = 20;

Dada a variável

var x: coluna;

as atribuições x[1]:=0.75, x[2]:=0.85 e x[3]:=1.5 são possíveis.

Tipo Estruturado Registro - Exemplo (1)

```
type data    = record
                  dia : 1..31;
                  mês : 1..12;
               end;
type pessoa  = record
                  sobrenome     : alfa;
                  primeironome  : alfa;
                  aniversário   : data;
                  sexo          : (m, f);
               end;
```


Tipo Estruturado Registro - Exemplo (2)

Declarada a variável

```
var p: pessoa;
```

valores particulares podem ser atribuídos:

```
p.sobrenome          := 'Ziviani';
p.primeironome       := 'Patricia';
p.aniversário.dia    := 21;
p.aniversário.mês    := 10;
p.sexo               := f;
```


Tipo Estruturado Conjunto - Exemplo

```
type conjint   = set of 1..9;
type conjcor   = set of cor;
type conjchar  = set of char;
```

O tipo cor deve ser previamente definido

```
type cor = (vermelho, azul, rosa);
```

Declaradas as variáveis

```
var ci : conjint;
var cc : array [1..5] of conjcor;
var ch : conjchar;
```


Tipo Estruturado Conjunto - Exemplo

Valores particulares podem ser construídos e atribuídos:

```
ci     := [1,4,9];
cc[2]  := [vermelho..rosa];
cc[4]  := [ ];
cc[5]  := [azul, rosa];
```

Prioridade: “interseção” precede “união” e “diferença”, que precedem “pertence a”.

[1..5,7] $*$ [4,6,8]	é [4]
[1..3,5] $+$ [4,6]	é [1..6]
[1..3,5] $-$ [2,4]	é [1,3,5]
2 **in** [1..5]	é true

Tipo Estruturado Arquivo - Exemplo

```
program Copia (Velho, Novo);  { copia o arquivo Velho no arquivo Novo }
const N = 30;
type TipoAlfa = array [1..N] of char;
     TipoData = record dia: 1..31; mes: 1..12; end;
     TipoPessoa = record
                    Sobrenome    : TipoAlfa;
                    PrimeiroNome : TipoAlfa;
                    Aniversario  : TipoData;
                    Sexo         : (mas,fem);
                  end;
var Velho, Novo: file of TipoPessoa;
    Registro    : TipoPessoa;
begin
  reset (Velho);  rewrite (Novo);
  while not eof (Velho) do
    begin read (Velho, Registro); write(Novo, Registro); end
end.
```

O programa ao lado copia o conteúdo do arquivo Velho no arquivo Novo. (Atribuição de nomes de arquivos externos ao programa varia de compilador para compilador.)

Tipo Apontador (1)

- São úteis para criar estruturas de dados **encadeadas**, do tipo listas, árvores e grafos.
- Um apontador é uma variável que referencia outra variável **alocada dinamicamente**.
- Em geral, a variável referenciada é definida como um registro com um apontador para outro elemento do mesmo tipo.

Tipo Apontador (2)

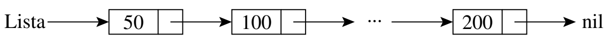
Exemplo:

```
type Apontador = ^No;
type No = record
            Chave: integer
            Apont: Apontador;
          end;
```

Dada uma variável

```
var Lista: Apontador;
```

é possível criar uma lista como a ilustrada.



Separador de Comandos

- O **ponto e vírgula** atua como um separador de comandos.
- Quando mal colocado, pode causar erros que não são detectados em tempo de compilação.

Exemplo: o trecho de programa abaixo está sintaticamente correto. Entretanto, o comando de adição serSS executado sempre, e não somente quando o valor da variável a for igual a zero.

```
if a = 0 then;
a := a + 1;
```

Passagem de Parâmetros (1)

- **Por valor** ou **por variável** (ou **referência**).
- A passagem de parâmetro por variável deve ser utilizada se o valor pode sofrer alteração dentro do procedimento, e o novo valor deve retornar para quem chamou o procedimento.

Passagem de Parâmetros (2)

Exemplo: SomaUm recebe o parâmetro x por valor e o parâmetro y por variável.

```
program Teste;
var a, b: integer;
  procedure SomaUm (x: integer; var y: integer);
  begin
    x := x + 1;    y := y + 1;
    writeln('Procedimento SomaUm:', x, y);
  end;
begin { Programa principal }
  a := 0;  b := 0;  SomaUm (a,b);
  writeln('Programa principal:', a, b);
end.
```

Resultado da execução:

```
Procedimento SomaUm : 1  1
Programa principal  : 0  1
```