Introdução*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

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.
- Seqüência 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 bastante simples).
  - 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 (seqüência 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 *int, boolean, char* e *float* de Java).
  - 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's)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TAD's são utilizados extensivamente como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação específica 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 TAD's 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 TAD's

- Considere uma aplicação que utilize 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).
- 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 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 dada 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 bastante 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 deste aspecto.
- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma 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 a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

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

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros$v[0..n-1], n \geq 1$.

```java
package cap1;
public class Max {
  public static int max (int v[], int n) {
    int max = v[0];
    for (int i = 1; i < n; i++)
        if (max < v[i]) max = v[i];
    return max;
  }
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo $f(n) = n - 1, \text{ para } n > 0$.
- Vamos provar que o algoritmo apresentado no programa acima é **ótimo**.

Exemplo - Maior Elemento

- **Teorema**: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.
- **Prova**: Deve ser mostrado, por meio de comparações, que cada um dos $n - 1$ elementos é menor do que algum outro elemento.
- Logo $n - 1$ comparações são necessárias. $\Box$
- O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então o método max da classe Max é ótimo.

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 do método 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

- **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.
- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n 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 de pesquisa mais simples é o que faz a **pesquisa seqüencial**.
- 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 considerando que 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 $p_i = 1/n, 0 \le i < n$.
- Neste 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)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $v[0..n-1], n \geq 1$.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.
- O vetor $maxMin$ definido localmente no método $maxMin1$ é utilizado para retornar nas posições 0 e 1 o maior e o menor elemento do vetor v, respectivamente.

```
package cap1;
public class MaxMin1 {
  public static int [] maxMin1 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```


Exemplo - Maior e Menor Elemento (1)

- Seja $f(n)$ o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo $f(n) = 2(n-1), \text{ para } n > 0$, para o melhor caso, pior caso e caso médio.
- MaxMin1 pode ser facilmente melhorado: a comparação $v[i] < min$ só é necessária quando a comparação $v[i] > max$ é falsa.
- A seguir, apresentamos essa versão melhorada.

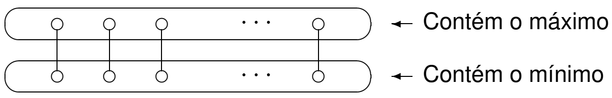
Exemplo - Maior e Menor Elemento (2)

```
package cap1;
public class MaxMin2 {
  public static int [] maxMin2 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      else if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

- Para a nova implementação temos:
  - melhor caso: $f(n) = n - 1$ (quando os elementos estão em ordem crescente);
  - pior caso: $f(n) = 2(n-1)$ (quando os elementos estão em ordem decrescente);
  - caso médio: $f(n) = 3n/2 - 3/2$.
- No caso médio, $v[i]$ 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 v 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)

```
package cap1;
public class MaxMin3 {
  public static int [] maxMin3 (int v[], int n) {
    int max, min, FimDoAnel;
    if ((n % 2) > 0) { v[n] = v[n−1]; FimDoAnel = n; }
    else FimDoAnel = n−1;
    if (v[0] > v[1]) { max = v[0]; min = v[1]; }
    else { max = v[1]; min = v[0]; }
    int i = 2;
    while (i < FimDoAnel) {
      if (v[i] > v[i+1]) {
        if (v[i] > max) max = v[i];
        if (v[i+1] < min) min = v[i+1];
      }
      else {
        if (v[i] < min) min = v[i];
        if (v[i+1] > max) max = v[i+1];
      }
      i = i + 2;
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```


Exemplo - Maior e Menor Elemento (3)

- Os elementos de v são comparados dois a dois e os elementos maiores são comparados com max e os elementos menores são comparados com min.
- Quando n é ímpar, o elemento que está na posição $v[n-1]$ é duplicado na posição $v[n]$ 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 os Algoritmos MaxMin1, MaxMin2 e MaxMin3

- A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- 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 algoritmos	$f(n)$		
	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 elementos de um conjunto com n elementos não ordenados, $n \geq 1$, faz pelo menos $3\lceil n/2 \rceil - 2$ comparações.
- **Prova**: A técnica utilizada define um oráculo que descreve o comportamento do algoritmo por meio de um conjunto de n–tuplas, mais um conjunto de regras associadas 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 um 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

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

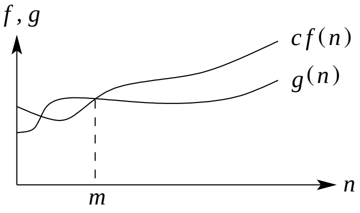
  comparações. $\Box$
- O teorema nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo, então o algoritmo MaxMin3 é **ótimo**.

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 $|g(n)| \leq c \times |f(n)|$.

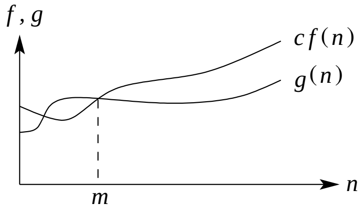


Exemplo:

- 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 mostrado é o menor valor possível, mas qualquer valor maior também 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 $c = 4$.
  - Isso 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

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

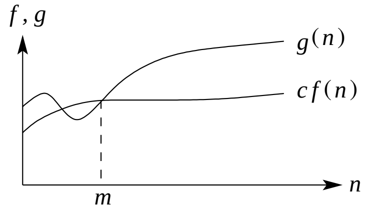
Exemplo: regra da soma $O(f(n)) + O(g(n))$.

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

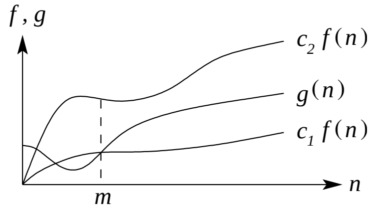
- Especifica um limite inferior para $g(n)$.
- **Definição**: Uma função $g(n)$ é $\Omega(f(n))$ se existirem duas constantes c e m tais que $g(n) \geq cf(n)$, para todo $n \geq m$.
- **Exemplo**: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$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



- 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 positivas 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 $c_2 f(n)$.
- Isto é, para todo $n \geq m$, a função $g(n)$ é igual a $f(n)$ a menos de uma constante.
- Neste 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 constantes 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** ou o comportamento assintótico 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.
- Nestes 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 de execução $O(n)$ 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?
  - depende do tamanho do problema.
  - 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 de **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 de **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 também dobra.
- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e juntando 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 de **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 de **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 de **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 de complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa **força bruta** para 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

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.

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

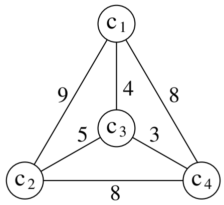
- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n), c > 1$.
- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
- Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais.
- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.
- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo 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 $\times$ 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 freqüência 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 freqüência 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: $O(1)$.
- Seqüência de comandos: determinado pelo maior tempo de execução de qualquer comando da seqüência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é $O(1)$.
- Anel: soma do tempo de execução 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 são 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.

```
package cap1;
public class Ordenacao {
  public static void ordena (int v[], int n) {
    for (int i = 0; i < n - 1; i++) {
      int min = i;
      for (int j = i + 1; j < n; j++)
        if (v[j] < v[min])
          min = j;
      /* Troca v[min] e v[i] */
      int x = v[min];
      v[min] = v[i];
      v[i] = x;
    }
  }
}
```

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento $v[0]$.
- 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

```
    void pesquisa(n) {
(1)   if (n <= 1)
(2)     'inspecione elemento' e termine
      else {
(3)     para cada um dos n elementos 'inspecione elemento';
(4)     pesquisa(n/3);
      }
    }
```

- 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 $T(1)$.

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

- 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 $T(n) = n\sum_{i=0}^{\infty}(1/3)^i = n\left(\frac{1}{1-\frac{1}{3}}\right) = \frac{3n}{2}\cdot$
- 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 $T(n) = \sum_{i=0}^{x-1}\frac{n}{3^i} + T(\frac{n}{3^x}) = n\sum_{i=0}^{x-1}(1/3)^i + 1 = \frac{n(1-(\frac{1}{3})^x)}{(1-\frac{1}{3})} + 1 = \frac{3n}{2} - \frac{1}{2}\cdot$
- Logo, o programa do exemplo é $O(n)$.

A Linguagem de Programação Java

- **Programação orientada a objetos:** nasceu porque algumas **linguagens procedimentais** se mostraram inadequadas para a construção de programas de grande porte.
- Existem dois tipos de problemas:
  1. **Falta de correspondência entre o programa e o mundo real:** Os procedimentos implementam tarefas e estruturas de dados armazenam informação, mas a maioria dos objetos do mundo real contém as duas coisas.
  2. **Organização interna dos programas:** Não existe uma maneira flexível para dizer que determinados procedimentos poderiam acessar uma variável enquanto outros não.

A Linguagem de Programação Java

- **Programação orientada a objetos:** permite que objetos do mundo real que compartilham propriedades e comportamentos comuns sejam agrupados em classes.
- Estilo de programação diretamente suportado pelo conceito de classe em Java.
- Pode-se também impor restrições de visibilidade aos dados de um programa.
- Classes e objetos são os conceitos fundamentais nas linguagens orientadas a objeto.
- A linguagem Java possui um grau de orientação a objetos maior do que a linguagem C++.
- Java não é totalmente orientada a objetos como a linguagem **S**malltalk.
- Java não é totalmente orientada a objetos porque, por questões de eficiência, foram mantidos alguns tipos primitivos e suas operações.

Principais Componentes de um Programa Java

- Em Java, as funções e os procedimentos são chamados de **métodos.**
- Um **objeto** contém métodos e variáveis que representam seus campos de dados (atributos).
  - Ex: um objeto $painelDeControle$ deveria conter não somente os métodos $ligaForno$ e $desligaForno$, mas também as variáveis $temperaturaCorrente$ e $temperaturaDese\text{-}jada$.
- O conceito de objeto resolve bem os problemas apontados anteriormente.
  - Os métodos $ligaForno$ e $desligaForno$ podem acessar as variáveis $temperaturaCorrente$ e $temperaturaDesejada$, mas elas ficam escondidas de outros métodos que não fazem parte do objeto $painelDeControle$.

Principais Componentes de um Programa Java

- O conceito de classe nasceu da necessidade de se criar diversos objetos de um mesmo tipo.
- Dizemos que um objeto pertence a uma classe ou, mais comumente, que é uma **i**nstância

```
package cap1;
class PainelDeControle {
  private float temperaturaCorrente;
  private float temperaturaDesejada;

  public void ligaForno () {
    // código do método
  }
  public void desligaForno() {
    // código do método
  }
}
```

- A palavra chave **class** introduz a classe $PainelDeControle$.
- A palavra chave **void** é utilizada para indicar que os métodos não retornam nenhum valor.

Principais Componentes de um Programa Java

- Um objeto em Java é criado usando a palavra chave **n**ew
- É necessário armazenar uma referência para ele em uma variável do mesmo tipo da classe, como abaixo:

  PainelDeControle painel1, painel2;

- Posteriormente, cria-se os objetos, como a seguir:

  painel1 = **n**ew PainelDeControle ();
  painel2 = **n**ew PainelDeControle ();

- Outras partes do programa interagem com os métodos dos objetos por meio do operador (**.**), o qual associa um objeto com um de seus métodos, como a seguir:

  painel1.ligaForno ();

Herança e Polimorfismo

- **Herança:** criação de uma classe a partir de uma outra classe.
- A classe é estendida a partir da classe base usando a palavra chave **extends**.
- A classe estendida (**subclasse**) tem todas as características da classe base (**superclasse**) mais alguma característica adicional.
- **Polimorfismo:** tratamento de objetos de classes diferentes de uma mesma forma.
- As classes diferentes devem ser derivadas da mesma classe base.

Herança e Polimorfismo

```
package cap1;
class Empregado {
  protected float salario;
  public float salarioMensal () { return salario; }
  public void imprime () { System.out.println ("Empregado"); }
}
class Secretaria extends Empregado {
  private int velocidadeDeDigitacao;
  public void imprime () { System.out.println ("Secretaria");}
}
class Gerente extends Empregado {
  private float bonus;
  public float salarioMensal () { return salario + bonus; }
  public void imprime () { System.out.println ("Gerente"); }
}
public class Polimorfismo {
  public static void main (String[] args) {
    Empregado empregado = new Empregado ();
    Empregado secretaria = new Secretaria ();
    Empregado gerente = new Gerente ();
    empregado.imprime (); secretaria.imprime ();
    gerente.imprime ();
  }
}
```


Objetos e Tipos Genéricos

- Uma estrutura de dados é genérica quando o tipo dos dados armazenados na estrutura é definido na aplicação que a utiliza (**objetos genéricos**).
- Um **objeto genérico** pode armazenar uma referência para um objeto de qualquer classe (classe **Object** em Java).
- Os mecanismos de **herança** e **polimorfismo** que permitem a implementação de estruturas de dados genéricas.

```
package cap1.objetogenerico;
public class Lista {
  private static class Celula {
    Object item; Celula prox;
  }
  private Celula primeiro, ultimo;
}
```

- O objeto $item$ é definido como um objeto genérico, assim $Lista$ pode ter objetos de classes distintas em cada item

Objetos e Tipos Genéricos

- Para evitar que se declare o tipo de cada objeto a ser inserido ou retirado da lista, a **Versão 5** da linguagem Java introduziu um mecanismo de definição de um tipo genérico.
- **Tipo genérico:** definição de um parâmetro de tipo que deve ser especificado na aplicação que utiliza a estrutura de dados:

```
package cap1.tipogenerico;
public class Lista<T> {
  private static class Celula<T> {
    T item;
    Celula<T> prox;
  }
  private Celula<T> primeiro, ultimo;
}
```

- O objeto $item$ tem de ser uma instância de um tipo genérico T que será fornecido quando um objeto da classe $Lista$ for instanciado.
  - Para instanciar uma lista de inteiros basta declarar o comando “Lista<Integer> lista = new Lista<Integer>();”.

Sobrecarga

- A **sobrecarga** acontece quando determinado objeto se comporta de diferentes formas.
- É um tipo de **p**olimorfismo *ad hoc*, no qual um identificador representa vários métodos com computações distintas.

```
public float salarioMensal (float desconto) {
  return salario + bonus - desconto;
}
```

- O programa acima apresenta um exemplo de sobrecarga do método $salarioMensal$ da classe $Gerente$ mostrada em um programa anterior, em que um $desconto$ é subtraído de $salario + bonus$.
- Note que o método $salarioMensal$ do programa acima possui uma assinatura diferente da assinatura apresentada no programa anterior.

Sobrescrita

- A ocultação de um método de uma classe mais genérica em uma classe mais específica é chamada de **s**obrescrita
- Por exemplo, o método $imprime$ da classe $Empregado$ apresentada nas parte de Herança e Polimorfismo, foi sobrescrito nas classes $Gerente$ e $Secretaria$.
- Para sobrescrever um método em uma subclasse é preciso que ele tenha a mesma assinatura na superclasse.

Programa Principal

```
package cap1;
class ContaBancaria {
  private double saldo;
  public ContaBancaria (double saldoInicial) {
    saldo = saldoInicial;
  }
  public void deposito (double valor) {
    saldo = saldo + valor;
  }
  public void saque (double valor) {
    saldo = saldo − valor;
  }
  public void imprime () {
    System.out.println ("saldo=" + saldo);
  }
}
public class AplicacaoBancaria {
  public static void main (String[] args) {
    ContaBancaria conta1 = new ContaBancaria (200.00);
    System.out.print ("Antes da movimentacao, ");
    conta1.imprime ();
    conta1.deposito (50.00); conta1.saque (70.00);
    System.out.print ("Depois da movimentacao, ");
    conta1.imprime ();
  }
}
```


Programa Principal

- Programa anterior modela uma conta bancária típica com as operações: cria uma conta com um saldo inicial; imprime o saldo; realiza um depósito; realiza um saque e imprime o novo saldo;
- A classe $Contabancaria$ tem um campo de dados chamado $saldo$ e três métodos chamados $deposito$, $saque$ e $imprime$.
- Para **c**ompilar o Programa acima a partir de uma linha de comando em MS-DOS ou Linux, fazemos:

  javac -d **./** $AplicacaoBancaria.java$

  e para executá-lo, fazemos:

  java $cap1.AplicacaoBancaria$
- A classe $ContaBancaria$ tem um método especial denominado **construtor**, que é chamado automaticamente sempre que um novo objeto é criado com o comando **new** e tem sempre o mesmo nome da classe.

Modificadores de Acesso

- **Modificadores de acesso:** determinam quais outros métodos podem acessar um campo de dados ou um método.
- Um campo de dados ou um método que seja precedido pelo modificador **private** pode ser acessado somente por métodos que fazem parte da mesma classe.
- Um campo de dados ou um método que seja precedido pelo modificador **public** pode ser acessado por métodos de outras classes.
  - Classe modificada com o modificador **public** indica que a classe é visível externamente ao pacote em que ela foi definida (classe $AplicacaoBancaria$, **package** cap1).
  - Em cada arquivo de um programa Java só pode existir uma classe modificada por **p**ublic, e o nome do arquivo deve ser o mesmo dado à classe.
- Os campos de dados de uma classe são geralmente feitos **private** e os métodos são tornados **public**.

Modificadores de Acesso

- Modificador protected: utilizado para permitir que somente subclasses de uma classe mais genérica possam acessar os campos de dados precedidos com **protected**.
- Um campo de dados ou um método de uma classe declarado como **static** pertence à classe e não às suas instâncias, ou seja, somente um campo de dados ou um método será criado pelo compilador para todas as instâncias.
- Os métodos de uma classe que foram declarados **static** operam somente sobre os campos da classe que também foram declarados **static**.
- Se além de **static** o método for declarado **public** será possível acessá-lo com o nome da classe e o operador (**.**).

Modificadores de Acesso

```
package cap1;
class A {
  public static int total;
  public int media;
}
public class B {
  public static void main (String[] args) {
    A a = new A(); a.total = 5; a.media = 5;
    A b = new A(); b.total = 7; b.media = 7;
  }
}
```

- No exemplo acima, o campo de dados $total$ pertence somente à classe A, enquanto o campo de dados $media$ pertence a todas as instâncias da classe A.
- Ao final da execução do método $main$, os valores de $a.total$ e $b.total$ são iguais a 7, enquanto os valores de $a.media$ e $b.media$ são iguais a 5 e 7, respectivamente.

Interfaces

- Uma **i**nterface em Java é uma **classe abstrata** que não pode ser instanciada, cujos os métodos devem ser **p**ublic e somente suas assinaturas são definidas
- Uma **i**nterface é sempre implementada por outras classes.
- Utilizada para prover a especificação de um comportamento que seja comum a um conjunto de objetos.

```
package cap1;
import java.io.*;
public class Max {
  public static Item max (Item v[], int n) {
    Item max = v[0];
    for (int i = 1; i < n; i++)
        if (max.compara (v[i]) < 0) max = v[i];
    return max;
  }
}
```

- O programa acima apresenta uma versão generalizada do programa para obter o máximo de um conjunto de inteiros.

Interfaces

- Para permitir a generalização do tipo de dados da chave é necessário criar a interface $Item$ que apresenta a assinatura do método abstrato $compara$.

```
package cap1;
public interface Item {
  public int compara (Item it);
}
```

- A classe $MeuItem$, o tipo de dados da chave é definido e o método $compara$ é implementado.

```
package cap1;
import java.io.*;
public class MeuItem implements Item {
  public int chave;
  // outros componentes do registro

  public MeuItem (int chave) { this.chave = chave; }
  public int compara (Item it) {
    MeuItem item = (MeuItem) it;
    if (this.chave < item.chave) return -1;
    else if (this.chave > item.chave) return 1;
    return 0;
  }
}
```

Interfaces

```
package cap1;
public class EncontraMax {
  public static void main (String[] args) {
    MeuItem v[] = new MeuItem[2];
    v[0] = new MeuItem (3); v[1] = new MeuItem (10);
    MeuItem max = (MeuItem) Max.max (v, 2);
    System.out.println ("Maior chave: " + max.chave);
  }
}
```

- O programa acima ilustra a utilização do método $compara$ apresentado.
- Note que para atribuir a um objeto da classe $MeuItem$ o valor máximo retornado pelo método max é necessário fazer uma conversão do tipo $Item$ para o tipo $MeuItem$, conforme ilustra a penúltima linha do método $main$.

Pacotes

- A linguagem Java permite agrupar as classes e as interfaces em pacotes(do inglês, **p**ackage.
- Convenientes para organizar e separar as classes de um conjunto de programas de outras bibliotecas de classes, evitando colisões entre nomes de classes desenvolvidas por uma equipe composta por muitos programadores.
- Deve ser realizada sempre na primeira linha do arquivo fonte, da seguinte forma por exemplo:

  package cap1;
- É possível definir subpacotes separados por ".", por exemplo, para definir o subpacote $arranjo$ do pacote $cap3$ fazemos:

  package cap3.arranjo;
- A utilização de uma classe definida em outro pacote é realizada através da palavra chave **import**. O comando abaixo possibilita a utilização de todas as classes de um pacote:

  import cap3.arranjo.*;

Pacotes

- É possível utilizar determinada classe de um pacote sem importá-la, para isso basta prefixar o nome da classe com o nome do pacote durante a declaração de uma variável. Exemplo:

  cap3.arranjo.Lista lista;
- Para que uma classe possa ser importada em um pacote diferente do que ela foi definida é preciso declará-la como pública por meio do modificador **public**.
- Se o comando **package** não é colocado no código fonte, então Java adiciona as classes daquele código fonte no que é chamado de pacote *default*
- Quando o modificador de um campo ou método não é estabelecido, diz-se que o campo ou método possui visibilidade *default*, ou seja, qualquer objeto de uma classe do pacote pode acessar diretamente aquele campo (ou método).

Classes Internas

- Java permite realizar aninhamento de classes como abaixo:

```
package cap1;
public class Lista {
  // Código da classe Lista
  private class Celula {
    // Código da classe Celula
  }
}
```

- Classes internas são muito úteis para evitar conflitos de nomes.
- Os campos e métodos declarados na classe externa podem ser diretamente acessados dentro da classe interna, mesmo os declarados como **protected** ou **private**, mas o contrário não é verdadeiro.
- As classes externas só podem ser declaradas como públicas ou com visibilidade *default*.
- As classes internas podem também ser qualificadas com os modificadores **p**rivate, **p**rotected e **s**tatic e o efeito é mesmo obtido sobre qualquer atributo da classe externa.

O Objeto *this*

- Toda instância de uma classe possui uma variável especial chamada **t**his, que contém uma referência para a própria instância.
- Em algumas situações resolve questões de ambigüidade.

```
package cap1;
public class Conta {
  private double saldo;
  public void alteraSaldo (double saldo) {
    this.saldo = saldo;
  }
}
```

- No exemplo acima, o parâmetro $saldo$ do método $alteraSaldo$ possui o mesmo nome do campo de instância $saldo$ da classe $Conta$.
- Para diferenciá-los é necessário qualificar o campo da instância com o objeto **t**his.

Exceções

- As exceções são erros ou anomalias que podem ocorrer durante a execução de um programa.
- Deve ser obrigatoriamente representada por um objeto de uma subclasse da classe ***Throwable***, que possui duas subclasses diretas: (i) ***Exception*** e (ii) ***Error***
- Uma abordagem simples para tratar uma exceção é exibir uma mensagem relatando o erro ocorrido e retornar para quem chamou ou finalizar o programa, como no exemplo abaixo:

```
int divisao (int a, int b) {
  try {
    if (b == 0) throw new Exception ("Divisao por zero");
    return (a/b);
  }
  catch (Exception objeto) {
    System.out.println ("Erro:" + objeto.getMessage());
    return (0);
  }
}
```


Exceções

- O comando **try** trata uma exceção que tenha sido disparada em seu interior por um comando **throw**
- O comando **throw** instancia o objeto que representa a exceção e o envia para ser capturado pelo trecho de código que vai tratar a exceção.
- O comando **catch** captura a exceção e fornece o tratamento adequado.
- Uma abordagem mais elaborada para tratar uma exceção é separar o local onde a exceção é tratada do local onde ela ocorreu.
- Importante pelo fato de que um trecho de código em um nível mais alto pode possuir mais informação para decidir como melhor tratar a exceção.

Exceções

- No exemplo abaixo a exceção não é tratada no local onde ela ocorreu, e esse fato é explicitamente indicado pelo comando **t**hrows

```
int divisao (int a, int b) throws {
  if (b == 0) throw new Exception ("Divisao por zero");
  return (a/b);
}
```

- Considerando que o método $divisao$ está inserido em uma classe chamada $Divisao$, o trecho de código abaixo ilustra como capturar o objeto exceção que pode ser criado no método:

```
Divisao d = new Divisao ();
try {
  d.divisao (3, 0);
}
catch(Exception objeto) {
  System.out.println("Erro:"+objeto.getMessage());
}
```

Saída de Dados

- Os tipos primitivos e objetos do tipo *String* podem ser impressos com os comandos

  System.out.print (var);
  System.out.println (var);

- O método *print* deixa o cursor na mesma linha e o método *println* move o cursor para a próxima linha de saída.

Entrada de Dados

- Todo programa em Java que tenha leitura de dados tem de incluir o comando no início do programa

  import java.io.*;

- Método para ler do teclado uma cadeia de caracteres terminada com a tecla *Enter*:

  public static String getString () **throws** IOException {
  InputStreamReader inputString = **new** InputStreamReader (System.in);
  BufferedReader buffer = **new** BufferedReader (inputString);
  String s = buffer.readLine (); **return** s;
  }

- Método para realizar a entrada de um caractere a partir do teclado:

  public static char getChar () **t**hrows **I**OException {
  String s = getString ();
  return s.charAt (0);
  }

- Se o que está sendo lido é de outro tipo, então é necessário realizar uma conversão.

Diferenças entre Java e C++

- A maior diferença entre Java e C++ é a ausência de apontadores em Java(não utiliza apontadores explicitamente).
- Java trata tipos de dados primitivos, tais como **int**, **double** e **float**, de forma diferente do tramento dado a objetos.
- Em Java, uma referência pode ser vista como um apontador com a sintaxe de uma variável.
- A linguagem C++ tem variáveis referência, mas elas têm de ser especificadas de forma explícita com o símbolo &.
- Outra diferença significativa está relacionada com o operador de atribuição (=):
  - C++: após a execução de um comando com operador (=), passam a existir dois objetos com os mesmos dados estáticos.
  - Java: após a execução de um comando com operador (=), passam a existir duas variáveis que se referem ao mesmo objeto.

Diferenças entre Java e C++

- Em Java e em C++ os objetos são criados utilizando o operador **new**, entretanto, em Java o valor retornado é uma referência ao objeto criado, enquanto em C++ o valor retornado é um apontador para o objeto criado.
- A eliminação de apontadores em Java tem por objetivo tornar o *software* mais seguro, uma vez que não é possível manipular o endereço de *conta1*, evitando que alguém possa acidentalmente corromper o endereço.
- Em C++, a memória alocada pelo operador **new** tem de ser liberada pelo programador quando não é mais necessária, utilizando o operador **d**elete.
- Em Java, a liberação de memória é realizada pelo sistema de forma transparente para o programador (**coleta de lixo**, do inglês *garbage collection*).

Diferenças entre Java e C++

- Em Java, os objetos são passados para métodos como referências aos objetos criados, entretanto, os tipos primitivos de dados em Java são sempre passados por valor
- Em C++ uma passagem por referência deve ser especificada utilizando-se o &, caso contrário, temos uma passagem por valor.
- No caso de tipos primitivos de dados, tanto em Java quanto em C++ o operador de igualdade (==) diz se duas variáveis são iguais.
- No caso de objetos, em C++ o operador diz se dois objetos contêm o mesmo valor e em Java o operador de igualdade diz se duas referências são iguais, isto é, se apontam para o mesmo objeto.
- Em Java, para verificar se dois objetos diferentes contêm o mesmo valor é necessário utilizar o método $equals$ da classe $Object$ (O programador deve realizar a sobrescrita desse método para estabelecer a relação de igualdade).

Diferenças entre Java e C++

- Em C++ é possível redefinir operadores como $+$, $-$, $*$, $=$, de tal forma que eles se comportem de maneira diferente para os objetos de uma classe particular, mas em Java, não existe sobrecarga de operadores.
- Por questões de eficiência foram mantidos diversos tipos primitivos de dados, assim variáveis declaradas como um tipo primitivo em Java permitem acesso direto ao seu valor, exatamente como ocorre em C++.
- Em Java, o tipo **boolean** pode assumir os valores **false** ou **true** enquanto em C++ os valores inteiros 0 e 1
- O tipo **byte**não existe em C++.
- O tipo **char** em Java é sem sinal e usa dois bytes para acomodar a representação
- O tipo **Unicode** de caracteres acomoda caracteres internacionais de linguas tais como chinês e japonês.

Diferenças entre Java e C++

- O tipo **short** tem tratamento parecido em Java e C++.
- Em Java, o tipo **int** tem sempre 32 *bits*, enquanto em C++ de tamanho, dependendo de cada arquitetura do computador onde vai ser executado.
- Em Java, o tipo **float** usa o sufixo F (por exemplo, 2.357F) enquanto o tipo **d**ouble não necessita de sufixo.
- Em Java, o tipo **long** usa o sufixo L (por exemplo, 33L); quaisquer outros tipos inteiros não necessitam de sufixo.