



Faculdades Integradas Facvest
 Faculdade de Ciência da Computação – FACIC
 Curso de Ciência da Computação – 4ª Fase
 CC11 – Complexidade de Algoritmos
 Professora: Susi Meire F. Carvalho



Parte II – Análise de Algoritmos

EXERCÍCIOS PRELIMINARES

Antes de começar com a análise de algoritmos, vamos fazer alguns exercícios.

Contagem de iterações

Quanto vale k no fim do seguinte procedimento?

```
k := 0
para i := 1 até n faça
  para j := i até n faça
    k := k+1
```

Qual o valor de S no final de cada um dos fragmentos abaixo?

```
S := 0
para i := 1 até n faça
  para j := 1 até i faça
    S := S+1
```

```
S := 0
i := n
enquanto i > 0 faça
  para j := 1 até i faça
    S := S+1
  i := chão(i/2)
```

```
S := 0
i := n
enquanto i > 0 faça
  para j := 1 até n faça
    S := S+1
  i := chão(i/2)
```

```
S := 0
para i := 1 até n faça
  j := i
  enquanto j > 0 faça
    S := S + 1
  j := chão(j/2)
```

Conceito de chão e teto

Para qualquer número real x , denotamos por $\text{chão}(x)$ o maior inteiro $\leq x$. Por exemplo, $\text{chão}(13,97) = 13$. Para qualquer número inteiro positivo n , $(n-1)/2 \leq \text{chão}(n/2) \leq n/2$. Analogamente, denotamos por $\text{teto}(x)$ denota o menor inteiro $\geq x$. Por exemplo, $\text{teto}(13,01) = 14$. Para qualquer número inteiro positivo n , $n/2 \leq \text{teto}(n/2) \leq (n+1)/2$.

Intercalação de seqüências ordenadas

Escreva um algoritmo iterativo para resolver o seguinte problema:

dado um vetor $A[p..r]$ e um índice q tais que $A[p..q]$ e $A[q+1..r]$ estão em ordem crescente rearranjar o vetor $A[p..r]$ de modo que ele fique em ordem crescente.

Para que valores de p , q e r o problema faz sentido?

1 INTRODUÇÃO

A análise de um algoritmo pode ser experimental (prática) ou teórica. No primeiro caso tomamos uma implementação do algoritmo e a submetemos a variados e representativos valores de entrada, monitorando-se a sua execução. O objetivo é obter tabelas ou curvas que indicam a variação do seu comportamento em função de diferentes tipos de entrada. Este tipo de análise possui a utilidade de se conhecer o comportamento de uma certa implementação do algoritmo. Já a abordagem

teórica permite que seja feita uma análise independente de implementação. Esta análise consiste em estimar analiticamente a quantidade de passos (e eventualmente também a quantidade de operações em cada passo) que o algoritmo deverá efetuar diante de diferentes tamanhos e valores de entrada. O resultado desta análise é uma função f , real, não-negativa, da variável $n \geq 0$ que representa o tamanho da entrada, como veremos mais adiante.

Problema e Algoritmo

Um problema é uma pergunta de caráter geral a ser respondida; normalmente, um problema tem vários parâmetros (ou variáveis livres), cujos valores são variáveis. Um problema é descrito através de:

- 1) uma descrição geral de todos os seus parâmetros, e
- 2) um enunciado de quais propriedades a resposta, ou solução, deve satisfazer.

Uma instância de um problema é obtida ao se fixarem valores particulares de todos os parâmetros do problema.

Por exemplo, consideremos o problema de ordenar uma lista finita de n números inteiros. Os parâmetros deste problema consistem de n números inteiros $\langle M_1, M_2, \dots, M_n \rangle$, e a solução consiste nesses mesmos números em ordem, digamos, não-decrescente.

Uma instância desse problema poderia ser a lista $\langle 5, 47, 0, -9 \rangle$ para a qual $n=4$, e a solução é $\langle -9, 0, 5, 47 \rangle$.

Um algoritmo é, em geral, uma descrição passo a passo de como um problema é solucionável. A descrição deve ser finita, e os passos devem ser bem definidos, sem ambigüidades, e executáveis computacionalmente. Diz-se que um algoritmo resolve um problema P se este algoritmo recebe qualquer instância I de P e sempre produz uma solução para esta instância I . Diz-se, então, que o algoritmo resolve I , e que I é um dado de entrada do algoritmo, e a solução é um dado de saída do algoritmo. Enfatizamos que, para qualquer dado de entrada I , o algoritmo deverá ser executável em tempo finito e, além disso, produzir uma solução correta para P .

Suponhamos que se tenha uma descrição passo a passo com todas as propriedades de um algoritmo, exceto unicamente da garantia de que é executável em tempo finito em qualquer instância I . Então, tal descrição será chamada de procedimento.

Por exemplo, um algoritmo para resolver o problema de ordenar uma lista pode ser descrito da seguinte forma:

Algoritmo A: para ordenar uma lista em ordem não-decrescente

Entrada: uma lista (M_1, M_2, \dots, M_n) de $n \geq 2$ números inteiros

Saída: os números da lista de entrada ordenados em ordem não-decrescente

```

1 para j := n-1 até n faça
2   para i := 1 até j faça //supor que  $M_1, \dots, M_{j+1}$  não estão em ordem
3     se  $M_i > M_{i+1}$  então
4       troque os valores de  $M_{i+1}$  e  $M_i$  entre si
5     fim se
6   fim para
7 fim para
8 escreva saída  $(M_1, M_2, \dots, M_n)$ 
```

Durante a sequência de comparações e possíveis trocas, os números inteiros maiores vão sendo deslocados para a direita na lista. Na realidade, o maior número da lista é deslocado para a direita para se tornar M_n , e então este número deverá permanecer nesta posição. Repetimos então o processo descrito acima para a sub-lista M_1, M_2, \dots, M_{n-1} , deslocando o número apropriado para a posição M_{n-1} . Repetindo-se este processo para as sub-listas cada vez mais curtas, acabaremos ordenando todos os números em ordem não decrescente.

Para ilustrar a execução deste algoritmo, vamos supor que a entrada fornecida seja $(5, 47, 0, -9)$ e então mostramos a seguir os valores de i, j, M_i e M_{i+1} na linha 3 e a lista restante na linha 6, conforme a numeração das linhas no algoritmo.

j	i	M_i	M_{i+1}	Lista resultante
3	1	5	47	(5, 47, 0, -9)
3	2	47	0	(5, 0, 47, -9)
3	3	-9	47	(5, 0, -9, 47)
2	1	5	0	(0, 5, -9, 47)
2	2	-9	5	(0, -9, 5, 47)
1	1	0	-9	(-9, 0, 5, 47)

Observamos que para este exemplo de entrada com $n=4$ números na lista foram feitas 6 comparações na linha 4. De um modo geral, é relativamente fácil de se verificar que são feitas $n(n-1)/2$ comparações e, no máximo este mesmo número de trocas na linha 4.

Existem outros algoritmos para este problema que necessitam de um número menos de comparações para achar a solução.

Aspectos Importantes

- Um problema pode, geralmente, ser resolvido por diferentes algoritmos;
- A existência de um algoritmo para resolver um determinado problema não implica, necessariamente, que este problema possa ser realmente resolvido na prática. Há restrições de tempo e espaço;
- A análise de algoritmos pode ser definida como o estudo da estimativa de tempo de execução de algoritmos;
- O tempo de execução de um programa é uma grandeza física que é determinado pelos seguintes aspectos:
 - Tempo que a máquina leva para executar uma instrução ou passo de um programa;
 - A natureza do algoritmo, ou seja, de quantos passos são necessários para se resolver o problema para um dado;
 - O tamanho do conjunto de dados que constitui o problema;
- É necessário ter uma forma de criar medidas de comparação entre algoritmos que resolvem um mesmo problema. Desta forma, é possível determinar:
 - A viabilidade de um algoritmo;
 - Qual é o melhor algoritmo para a solução de um problema;
- O interessante é ter uma comparação relativa entre algoritmos:
 - Assumir que a execução de todo e qualquer passo de um algoritmo leva um tempo fixo e igual a uma unidade de tempo;
 - O tempo de execução de um computador particular não é interessante.

2 MEDIDA DO TEMPO DE EXECUÇÃO DE UM PROGRAMA

Na área de análise de algoritmos existem dois tipos de problemas bem distintos:

(i) Análise de um algoritmo em particular. Qual é o custo de usar um dado algoritmo para resolver um problema específico? Neste caso, características importantes do algoritmo em questão deve, ser investigadas, geralmente uma análise do número de vezes que cada parte do algoritmo deve ser executada, seguida do estudo da quantidade de memória necessária.

(ii) Análise de uma classe de algoritmos. Qual é o algoritmo de menor custo possível para resolver um problema particular? Neste caso, toda uma família de algoritmos para resolver um problema é investigada com o objetivo de identificar um que seja o melhor possível. Isso significa colocar limites para a complexidade computacional dos algoritmos pertencentes à classe.

Quando conseguimos determinar o menor custo possível para resolver problemas de uma determinada classe, como no caso dos algoritmos de ordenação, temos a medida da dificuldade inerente para resolver tais problemas. Ainda mais, quando o custo de um algoritmo é igual ao menor custo possível, então podemos concluir que o algoritmo é ótimo para a medida de custo considerada.

O custo de utilização de um algoritmo pode ser medido de várias maneiras. Uma delas é através da execução do programa em um computador real, sendo o tempo de execução medido diretamente. As medidas de tempo obtidas desta forma são bastante inadequadas e os resultados jamais devem ser generalizados. As principais objeções são: (i) os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras; (ii) os resultados dependem do hardware; (iii) quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto. Apesar disso existem situações particulares como, por exemplo, quando existem vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro da mesma ordem de grandeza.

Uma forma mais adequada de se medir o custo de utilização de um algoritmo é através do uso de um modelo matemático. O conjunto de operações a serem executadas deve ser especificado, assim como o custo associado com a execução de cada operação. Mais usual ainda é ignorar o custo de algumas das operações envolvidas e considerar apenas as operações mais significativas. Por exemplo, para algoritmos de ordenação, consideramos o número de comparações e trocas entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulação de índices, caso existam.

Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade de tempo f , onde $f(n)$ é a medida de tempo necessário para executar um algoritmo para um problema de tamanho n . Se $f(n)$ for uma medida da memória necessária para executar um algoritmo de tamanho n , então f é chamada função de complexidade de espaço do algoritmo. Normalmente utilizamos complexidade de tempo, como será visto daqui em diante. É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Para ilustrar alguns dos conceitos acima, considere o seguinte algoritmo para encontrar o maior elemento de um vetor de inteiros $A[1..n]$, $n \geq 1$.

```
função Max (var A : vetor);
var i,temp: inteiro;
início
    temp := A[1];
    para i := 2 até n faça
        se temp < A[i] então
            temp := A[i];
    max := temp;
fim.
```

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 $f(n) = n-1$, para $n > 0$.

Este algoritmo é ótimo pois qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz, pelo menos, $n-1$ comparações.

A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados. Por isso é comum considerar-se o tempo de execução de um programa como uma função do tamanho da entrada. Entretanto, para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada. Na função apresentada o algoritmo possui a propriedade de que o custo é uniforme sobre todos os problemas de tamanho n . Já para um algoritmo de ordenação, isto não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Temos então que distinguir três cenários: melhor caso, pior caso e caso médio. O melhor caso corresponde ao menor tempo de execução sobre todas as possíveis entradas de tamanho n . O pior caso corresponde ao maior tempo de execução sobre todas as entradas de tamanho n . O caso médio corresponde à média dos tempos de execução de todas as entradas de tamanho n . Geralmente a análise do caso médio é muito mais difícil em virtude de todas as probabilidades a serem analisadas e é comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Entretanto, na prática, isso nem sempre é verdade.

Considere o problema de encontrar o maior elemento e o menor elemento de um vetor de inteiros $A[1..n]$, $n \geq 1$. Um algoritmo simples para resolver este problema pode ser derivado do algoritmo Max apresentado anteriormente, como pode ser visto a seguir.

```
procedimento MaxMin1 (var A : vetor; var max, min: inteiro);
var i: inteiro;

início
    max := A[1];
    min := A[1];
    para i := 2 até n faça
        se A[i] > max então
            max := A[i];
        se A[i] < min então
            min := A[i];
    fim para;
fim.
```

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 $f(n) = 2(n-1)$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Este programa pode ser facilmente melhorado. Basta observar que a comparação $A[i] < \min$ somente é necessária quando o resultado da comparação $A[i] > \max$ é falso. Uma nova versão do algoritmo pode ser vista a seguir.

```

procedimento MaxMin2 (var A : vetor; var max, min: inteiro);
var i: inteiro;
início
    max := A[1];
    min := A[1];
    para i := 2 até n faça
        se A[i] > max então
            max := A[i];
        senão
            se A[i] < min então
                min := A[i];
    fim para;
fim.

```

Nesta nova versão os casos a considerar são:

Melhor caso: $f(n) = n-1$

Pior caso: $f(n) = 2(n-1)$

Caso médio: $f(n) = 3n/2 - 3/2$

O melhor caso ocorre quando os elementos de A estão em ordem crescente. O pior caso ocorre quando os elementos de A estão em ordem decrescente. No caso médio, $A[i]$ é maior do que max a metade das vezes. Logo $f(n) = (n-1 + 2(n-1))/2 = n-1 + (n-1)/2 = 3/2n - 3/2$, para $n > 0$.

Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente para este problema? A resposta é sim. Considere o seguinte algoritmo:

- 1) Compare os elementos de A aos pares, separando-os em dois subconjuntos de acordo com o resultado da comparação, colocando os maiores em um subconjunto e os menores no outro, a um custo de $\text{teto}(n/2)$ comparações.
- 2) O máximo é obtido do subconjunto que contém os maiores elementos a um custo de $\text{teto}(n/2)-1$ comparações.
- 3) O mínimo é obtido do subconjunto que contém os menores elementos a um custo de $\text{teto}(n/2)-1$ comparações.

A implementação do algoritmo acima descrito é apresentada a seguir:

```

procedimento MaxMin3 (var A : vetor; var max, min: inteiro);
var i, fimDoAnel: inteiro;
início
    se (n mod 2) > 0 então
        A[n+1] := A[n];
        fimDoAnel := n;
    senão
        fimDoAnel := n-1;
    fim se;
    se A[1] > A[2] então
        max := A[1];
        min := A[2];
    senão
        max := A[2];
        min := A[1];
    fim se;
    i := 3;
    enquanto i < fimDoAnel faça
        completar laço do algoritmo...
    fim enquanto;
fim.

```

Os elementos de A 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 A[n] é duplicado na posição A[n+1] para evitar o tratamento de exceção. Para esta implementação, $f(n) = n/2 + (n-2)/2 + (n-2)/2 = 3n/2 - 2$, para $n > 0$, para o melhor caso, pior caso e caso médio.

A tabela a seguir apresenta uma comparação entre os três algoritmos.

Tabela 1: Comparação entre os algoritmos para obter o máximo e o mínimo

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	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

3 ANÁLISE ASSINTÓTICA

Ao ver uma expressão como $n+10$ ou n^2+1 , a maioria das pessoas pensa automaticamente em valores pequenos de n, tipicamente valores próximos de zero. Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco a ser executado, mesmo os algoritmos ineficientes. Logo, a análise de algoritmos faz exatamente o contrário: *ignora os valores pequenos* e concentra-se nos *valores enormes* de n. Para valores enormes de n, as funções

$$n^2, (3/2)n^2, 9999n^2, n^2/1000, n^2+100n, \text{ etc.}$$

crescem todas com a mesma velocidade e portanto são todas "equivalentes". Para tal, estuda-se o comportamento assintótico de suas funções de custo. Nessa matemática, interessada somente em valores enormes de n, as funções são classificadas em "ordens" (como as ordens religiosas da Idade Média); todas as funções de uma mesma ordem são "equivalentes". As cinco funções acima, por exemplo, pertencem à mesma ordem.

Ordem O

Convém restringir a atenção a funções assintoticamente não-negativas, ou seja, funções f tais que $f(n) \geq 0$ para todo n suficientemente grande. Mais explicitamente: f é assintoticamente não-negativa se existe n_0 tal que $f(n) \geq 0$ para todo n maior que n_0 .

Agora podemos definir a ordem O. [Isto é uma letra Ó maiúscula. Segundo Knuth, trata-se do ômicron grego maiúsculo.]

DEFINIÇÃO: Uma função $g(n)$ é $O(f(n))$ (lê-se $g(n)$ é da ordem no máximo $f(n)$) se existem duas constantes positivas c e m, tais que $g(n) \leq c \cdot f(n)$, para $n \geq m$.

EXEMPLOS:

- 1) Se $f(n) \leq 9999 g(n)$ para todo $n \geq 1000$ então $f(n) = O(g(n))$. (Mas cuidado: a recíproca não é verdadeira!)
- 2) Seja $g(n) = (n+1)^2$. Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$. Isto porque $(n+1)^2 \leq 4n^2$ para $n \geq 1$.
- 3) $2n^2 - 5 = O(n^2)$, já que $2n^2 - 5 \leq 2n^2$ para qualquer n.
- 4) $2n^3 + n + 10 = O(n^3)$, pois $2n^3 + n + 10 \leq 3n^3$ para $n \geq 3$.
- 5) $n^3 - 2n^2 + 4n = O(n^3)$, pois $n^3 - 2n^2 + 4n \leq 2n^3$ para todo $n > 1$.

ALGUMAS OPERAÇÕES COM A NOTAÇÃO O:

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

Exemplo: Suponha três trechos de programas 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 $O(n^2)$.

Exercícios

- 1) Quais das conjecturas abaixo são verdadeiras? Justifique.

- $10n = O(n)$ $10n^2 = O(n)$ $10n^{55} = O(2^n)$
 2) É verdade que $n^2 + 200n + 300 = O(n^2)$? Justifique.
 3) É verdade que $n^2 - 200n - 300 = O(n)$? Justifique.
 4) Quais das conjecturas abaixo são verdadeiras? Justifique.
 $(3/2)n^2 + (7/2)n - 4 = O(n)$ $(3/2)n^2 + (7/2)n - 4 = O(n^2)$
 5) É verdade que $n^3 - 999999n^2 - 1000000 = O(n^2)$? Justifique.

Ordem Ω (Omega)

A expressão " $f(n) = O(g(n))$ " tem o mesmo sabor que " $f(n) \leq g(n)$ ". Agora precisamos de um conceito que tenha o sabor de " $f(n) \geq g(n)$ ".

DEFINIÇÃO: Uma função $g(n)$ é $\Omega(f(n))$ se existir uma constante $c > 0$, tal que $g(n) \leq c \cdot f(n)$ para um número infinito de valores pra n .

EXEMPLOS:

- 1) Se $f(n) \geq g(n)/100000$ para todo $n \geq 888$ então $f(n) = \Omega(g(n))$. (Mas cuidado: a recíproca não é verdadeira!)
- 2) Qual a relação entre O e Ω ? Não é difícil verificar que $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$.
- 3) Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.
- 4) $n^3 - 2n^2 + 4n = \Omega(n^3)$, pois $n^3 - 2n^2 + 4n \geq 0.5n^3$ para todo $n > 1$.

Ordem Θ (Theta)

Além dos conceitos que têm o sabor de " $f(n) \leq g(n)$ " e de " $f(n) \geq g(n)$ ", precisamos de um que tenha o sabor de " $f(n) = g(n)$ ".

DEFINIÇÃO: Dizemos que uma função $f(n)$ é $\Theta(g(n))$ se $f(n) = g(n)$, ou seja, se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. Trocando em miúdos, para c_1 e c_2 positivos e todo n suficientemente grande $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

EXEMPLO: As funções abaixo pertencem todas à ordem $\Theta(n^2)$:
 n^2 , $(3/2)n^2$, $9999n^2$, $n^2/1000$, $n^2 + 100n$.

Exercícios

- 1) Quais das conjecturas abaixo são verdadeiras? Justifique.
 - a) $(3/2)n^2 + (7/2)n - 4 = \Theta(n^2)$
 - b) $9999n^2 = \Theta(n^2)$
 - c) $n^2/1000 - 999n = \Theta(n^2)$
- 2) As sentenças abaixo são falsas ou verdadeiras? Justifique a sua resposta.
 - a) Se a complexidade de um algoritmo é $f(n)$. Então o número de passos que ele executa para qualquer entrada é $\Omega(f(n))$.
 - b) Se a complexidade de um algoritmo é $f(n)$. Então o número de passos que ele executa para qualquer entrada é $O(f(n))$.
 - c) Para qualquer constante inteira positiva k , k é $O(1)$.

Funções de Complexidade

1. $f(n) = O(1)$. Algoritmos de complexidade $O(1)$ são ditos de complexidade constante. O uso do algoritmo independe do tamanho de n . Neste caso as instruções do algoritmo são executadas um número fixo de vezes.
2. $f(n) = O(\log n)$. Um algoritmo de complexidade $O(\log n)$ é dito ter complexidade logarítmica. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.
3. $f(n) = O(n)$. Um algoritmo de complexidade $O(n)$ é dito ter complexidade linear. Em geral um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho, o tempo de execução dobra.

4. $f(n) = O(n \log n)$. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois juntando as soluções.
5. $f(n) = O(n^2)$. Um algoritmo de complexidade $O(n^2)$ é dito ter complexidade quadrática. Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro do outro. Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.
6. $f(n) = O(n^3)$. Um algoritmo de complexidade $O(n^3)$ é dito ter complexidade cúbica. Algoritmos desta ordem de complexidade são úteis apenas para resolver pequenos problemas.
7. $f(n) = O(2^n)$. Um algoritmo de complexidade $O(2^n)$ é dito ter complexidade exponencial. Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa força bruta para resolvê-los.

Exemplo

Algoritmo para calcular x^n

```
potencial (x,n);
início
  i := n;
  y := 1;
  enquanto i > 0 faça
    y := y * x;
    i := i - 1;
  fim enquanto;
  retorne y;
fim.
```

- Linhas (1), (2) e (6) (exclui-se o início e o fim) são executadas uma única vez (3 unidades de tempo).
- O laço das linhas (3) a (5) é executado n vezes ($2 \cdot n$ unidades de tempo).
- O tempo de execução é $2n+3$ que é $O(n)$.

Para o Algoritmo A apresentado anteriormente, cujo tempo de execução foi de $n(n-1)/2$, dizemos que sua complexidade no pior caso é de $O(n^2)$. Já no melhor caso são executadas apenas $n-1$ comparações, o que reflete em uma complexidade de $\Omega(n)$. Para encontrar o tempo de execução no caso médio basta somar o tempo de execução no melhor caso ao tempo de execução no pior caso e dividir por 2: $(n(n-1)/2 + (n-1))/2$, que é igual a $n^2+n-1/2$, o que nos permite concluir que no caso médio, a complexidade do algoritmo A é $\Theta(n^2)$. À primeira vista a resposta parece extremamente vaga. Mas a verdade é que é impossível dar uma resposta mais precisa (a menos que se conheça muito bem o computador que será usado para executar o algoritmo).

A resposta é até bastante informativa num sentido relativo: ela garante que se o expoente for multiplicado por 10, então o tempo de execução será multiplicado por 100 desde que n seja suficientemente grande; isso é certamente menos agradável que multiplicar o tempo por 10, mas é menos desastroso que multiplicar o tempo por 1000.

Com relação à análise assintótica de algoritmos temos que fazer algumas considerações:

- A complexidade de tempo de um algoritmo é apenas um fato sobre ele. Um algoritmo assintoticamente mais barato pode ser muito mais difícil de ser implementado do que um outro algoritmo com complexidade de tempo maior.
- A superioridade assintótica só é evidente quando os problemas são suficientemente grandes.

Tabela 2: Comparação de várias funções de complexidade

Funções de Complexidade (ou de custo)	Tamanho da Entrada n		
	20	40	60
n	0.00002 seg	0.00004 seg	0.00006 seg

$n \log n$	0.00009 seg.	0.00021 seg.	0.00035 seg.
n^2	0.0004 seg	0.0016 seg	0.0036 seg
n^3	0.008 seg	0.64 seg	0.216 seg
2^n	1 seg	12,7 dias	366 sec.
3^n	58 min	3855 sec.	10^{13} sec.

Tabela 3: Influência do aumento de velocidade dos computadores, no tamanho t do problema

Função de complexidade (custo de tempo)	Computador atual	Computador 100 vezes mais rápido	Computador 1000 vezes mais rápido
n	t	$100 t$	$1000 t$
$n \log n$	t_1	$25.5 t_1$	$140.2 t_1$
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	$t_4 + 6.6$	$t_4 + 10$
3^n	t_5	$t_5 + 4$	$t_5 + 6$

4 TÉCNICAS DE ANÁLISE DE ALGORITMOS

Infelizmente não existe um conjunto completo de regras para analisar programas. Alguns princípios a serem seguidos são apresentados a seguir:

- O tempo de execução de um comando de atribuição, de leitura ou de escrita pode ser considerado como $O(1)$. Existem exceções para as linguagens que permitem a chamada de funções em comandos de atribuição, ou quando atribuições envolvem vetores de tamanho arbitrariamente grande.
- O tempo de execução de uma seqüência de comandos é determinado pelo maior tempo de execução de qualquer comando da seqüência. As instruções consecutivas são somadas.
- O tempo de execução de um comando de decisão (if/else, por exemplo) é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo para avaliar a condição, que é $O(1)$.
- O tempo para executar um laço é a soma do tempo de execução do corpo do laço mais o tempo de avaliar a condição para terminação, multiplicado pelo número de iterações do laço. Geralmente o tempo de avaliação da condição para terminação é $O(1)$. No caso de aninhamento de laços, analisa-se os mais internos. O tempo total de execução de uma instrução dentro de um grupo de laços aninhados é o tempo de execução da instrução multiplicado pelo produto dos tamanhos de todos os laços.
- Quando o programa possui procedimentos/funções não recursivos, o tempo de execução de cada procedimento deve ser computado separadamente um a um, iniciando com os procedimentos que não chamam outros procedimentos. Em seguida devem ser avaliados os procedimentos que chamam os procedimentos que não chamam outros procedimentos, utilizando os tempos dos procedimentos já avaliados. Este processo é repetido até chegar no programa principal.
- Quando um programa possui procedimentos recursivos, para cada procedimento é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento. Para descobrir esta $f(n)$, deve-se resolver uma relação de recorrência, o que não é trivial. (Vide exemplo)
- Tipos de análise:
 - **Pior caso:** indica o maior tempo obtido, levando em consideração todas as entradas possíveis.
 - **Melhor caso:** indica o menor tempo obtido, levando em consideração todas as entradas possíveis.
 - **Média:** indica o tempo médio obtido, considerando todas as entradas possíveis.

Para ilustrar estes conceitos, será apresentado um exemplo.

Exemplo: Considere o seguinte algoritmo para obter o maior e o menor elemento de um vetor de inteiros $A[1..n]$, $n \geq 1$:

```

procedimento MaxMin4 (Linf, Lsup : inteiro; var max, min: inteiro);
var max1, max2, min1, min2, meio: inteiro;
início
  se Lsup - Linf <= 1 então
    se A[Linf] < A[Lsup] então
      max := A[Lsup];
      min := A[Linf];
    senão
      max := A[Linf];
      min := A[Lsup];
  fim se
  senão
    meio := (Linf + Lsup) div 2;
    MaxMin4(Linf, meio, max1, min1);
    MaxMin4(meio+1, Lsup, max2, min2);
    se max1 > max2 então
      max := max1
    senão
      max := max2;
    se min1 < min2 então
      min := min1
    senão
      min := min2;
  fim se;
fim.

```

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

$f(n) = 1$, para $n \leq 2$;

$f(n) = f(\text{chão}(n/2)) + f(\text{teto}(n/2)) + 2$, para $n > 2$;

Quando $n = 2^i$, para algum inteiro positivo i então

$$\begin{aligned}
 f(n) &= 2f(n/2) + 2 \\
 2f(n/2) &= 4f(n/2) + 2 \times 2 \\
 4f(n/2) &= 8f(n/2) + 2 \times 2 \times 2 \\
 &\vdots \\
 2^{i-2}f(n/2) &= 2^{i-1}f(n/2^{i-1}) + 2^{i-1}
 \end{aligned}$$

Adicionando lado a lado, obtemos

$$\begin{aligned}
 f(n) &= 2^{i-1}f(n/2^{i-2}) + \sum_{k=1}^{i-1} 2^k \\
 &= 2^{i-1}f(2) + 2^i - 2 \\
 &= 2^{i-1} + 2^i - 2 \\
 &= 3n/2 - 2
 \end{aligned}$$

Logo,

$$f(n) = 3n/2 - 2$$

para o melhor caso, pior caso e caso médio.

Algumas Dicas de Análise

- Identificar a operação fundamental usada no algoritmo. A análise dessa operação fundamental identifica o tempo de execução. Isso pode evitar a análise linha-por-linha do algoritmo.
- Quando um algoritmo, em uma passada de uma iteração, divide o conjunto de dados de entrada em uma ou mais partes, tomado cada uma dessas partes e processando separada e recursivamente, e depois juntando os resultados, este algoritmo é possivelmente $n \log(n)$

- Um algoritmo é $\log(n)$ se ele leva tempo constante $O(1)$ para dividir o tamanho do problema, geralmente pela metade. Por exemplo, a pesquisa binária.
- Se o algoritmo leva tempo constante para reduzir o tamanho do problema em um tamanho constante, ele será $O(n)$
- Algoritmos combinatoriais são exponenciais. Por exemplo, o problema do caixeiro viajante.



5 ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

Neste item serão analisados alguns dos algoritmos de ordenação mais conhecidos.

Método da Inserção direta

Neste método, considera-se o segmento já ordenado como sendo formado pelo primeiro elemento do vetor de elementos. Os demais elementos, ou seja do 2º ao último, pertencem ao segmento não ordenado. A partir desta situação inicial toma-se um a um dos elementos não ordenados, a partir do primeiro e, por busca seqüencial realizada da direita para esquerda no segmento já ordenado, localiza-se sua posição relativa correta.

Exemplo

Vetor original	<table><tr><td>18</td><td>15</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr></table>							18	15	7	9	23	16	14
18	15	7	9	23	16	14								
Divisão inicial	<table><tr><td>18</td><td>15</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr></table> <div><div></div><div></div><div>ordenado</div><div>não ordenado</div></div>							18	15	7	9	23	16	14
18	15	7	9	23	16	14								
Primeira iteração	<table><tr><td>15</td><td>18</td><td>7</td><td>9</td><td>23</td><td>16</td><td>14</td></tr></table>							15	18	7	9	23	16	14
15	18	7	9	23	16	14								
Segunda iteração	<table><tr><td>7</td><td>15</td><td>18</td><td>9</td><td>23</td><td>16</td><td>14</td></tr></table>							7	15	18	9	23	16	14
7	15	18	9	23	16	14								
Terceira iteração	<table><tr><td>7</td><td>9</td><td>15</td><td>18</td><td>23</td><td>16</td><td>14</td></tr></table>							7	9	15	18	23	16	14
7	9	15	18	23	16	14								
Quarta iteração	<table><tr><td>7</td><td>9</td><td>15</td><td>18</td><td>23</td><td>16</td><td>14</td></tr></table>							7	9	15	18	23	16	14
7	9	15	18	23	16	14								
Quinta iteração	<table><tr><td>7</td><td>9</td><td>15</td><td>16</td><td>18</td><td>23</td><td>14</td></tr></table>							7	9	15	16	18	23	14
7	9	15	16	18	23	14								
Sexta iteração (última)	<table><tr><td>7</td><td>9</td><td>14</td><td>15</td><td>16</td><td>18</td><td>23</td></tr></table>							7	9	14	15	16	18	23
7	9	14	15	16	18	23								

Implementação

```

procedimento inserção_direta (V: vetor, n: inteiro);
var i, j : inteiro;
inicio
  para i := 2 até n faça
    k := 1;
    j := i-1;
    elem := V[i];
    enquanto j >= 1 e k = 1 faça
      se elem < c[j] então
        V[j+1] := V[j]
        j := j-1;
      senão
        k := j + 1;
    fim se;
  fim enquanto;
  V[k] := elem;

```

```

    fim para;
fim;

```

Análise do desempenho

O desempenho do método de inserção direta é fortemente influenciado pela ordem inicial dos elementos no vetor a ser ordenado. A situação mais desfavorável para o método é aquela em que o vetor a ser ordenado se apresenta na ordem contrária à desejada. Neste caso são necessárias as seguintes comparações e transposições:

```

1º elemento : 1
2º elemento : 2
3º elemento : 3
      ⋮
(n-1)º elemento : n-1

```

O total corresponderá à soma dos números de operações efetuadas em cada iteração:

$$1 + 2 + 3 + \dots + (n - 1)$$

que é igual à soma dos termos de uma progressão aritmética cujo primeiro termo é 1 e o último é (n-1):

$$S = (1 + (n-1)) / 2(n-1)$$

$$S = (n^2 - n) / 2$$

Por outro lado, o melhor caso para o método é aquele no qual os elementos já se apresentam previamente ordenados. Nesta situação, cada elemento a ser inserido sempre será maior do que aqueles que já o foram. Isto significa que nenhuma transposição é necessária. De qualquer maneira é necessário pelo menos uma comparação para localizar a posição do elemento. Logo, neste caso (melhor caso), o método efetuará um total de n-1 iterações para dar o vetor como ordenado.

Para o desempenho no caso médio temos:

$$((n - 1) + ((n^2 - 2) / 2)) / 2 = (n^2 + n - 2) / 4 = O(n^2)$$

A implementação dos métodos de ordenação apresentados a seguir está em Pascal. As análises de desempenho destes métodos serão realizadas em sala de aula.

Método da Bolha - Bubblesort

Neste método, o princípio geral é aplicado a todos os pares consecutivos de elementos. Este processo é executado enquanto houver pares consecutivos de elementos não ordenados. Quando não restarem mais pares, o vetor estará classificado.

Exemplo:

Vetor: 28 26 30 24 25

Primeira varredura:

```

28 26 30 24 25   compara par (28,26): troca.
26 28 30 24 25   compara par (28,30): não troca.
26 28 30 24 25   compara par (30,24): troca.
26 28 24 30 25   compara par (30,25): troca.
26 28 24 25 30   fim da primeira varredura.

```

Segunda varredura:

```

26 28 24 25 30   compara par (26,28): não troca.
26 28 24 25 30   compara par (28,24): troca.
26 24 28 25 30   compara par (28,25): troca.
26 24 25 28 30   fim da segunda varredura.

```

Terceira varredura:

```

26 24 25 28 30   compara par (26,24): troca.
24 26 25 28 30   compara par (26,25): troca.
24 25 26 28 30   fim da terceira varredura.

```

Como não restam mais pares não ordenados, o processo de classificação é dado por concluído.

Examinando-se o comportamento deste método, vemos que eventualmente, dependendo da disposição dos elementos, numa certa varredura pode colocar mais do que um elemento na sua posição definitiva. Isto ocorrerá sempre que houver blocos de elementos já previamente ordenados, como no exemplo abaixo:

Vetor: 13 11 25 10 18 21 23

A primeira varredura produzirá o seguinte resultado:

11 13 10 18 21 23 25

Implementação

Apresentação em sala de aula.

Análise do desempenho

Apresentação em sala de aula.

Método dos Incrementos Decrescentes - Shellsort

Este método explora o fato de que o método da inserção direta apresenta desempenho aceitável quando o número de elementos é pequeno e/ou estes já possuem uma ordenação parcial.

Para implementar o método, o vetor $V[1..n]$ é dividido em segmentos assim formados:

Segmento 1 : $V[1], V[h+1], V[2h+1], \dots$

Segmento 2 : $V[2], V[h+2], V[2h+2], \dots$

\vdots

Segmento h : $V[h], V[h+h], V[2h+h], \dots$

onde h é um incremento.

Exemplo

Convenção: os números acima de cada célula indicam os índices dos elementos do vetor. Os números abaixo de cada célula indicam o número do segmento ao qual cada célula pertence.

Primeiro passo: $h = 4$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	25	49	12	18	23	45	38	53	42	27	13	11	28	10	14
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Segundo passo: $h = 2$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
11	23	10	12	17	25	27	13	18	28	45	14	53	42	49	38
1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2

Terceiro (e último) passo: $h = 1$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	12	11	13	17	14	18	23	27	25	45	28	49	38	53	42
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Após o último passo, o vetor resultará classificado:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	11	12	13	14	17	18	23	25	27	28	38	42	45	49	53

Implementação

Apresentação em sala de aula.

Análise do desempenho

Apresentação em sala de aula.

Método da Seleção Direta (otimizada)

Neste método, a seleção do menor elemento é feita por pesquisa sequencial. Uma vez encontrado o menor elemento, este é permutado com o que ocupa a posição inicial do vetor, que fica, assim, reduzido de um elemento. O processo de seleção é repetido para a parte restante do vetor, sucessivamente, até que todos os elementos tenham sido selecionados e colocados em suas posições.

Exemplo

Suponha que deseja ordenar o vetor:

9 25 10 18 5 7 15 3

Segundo o princípio formulado, a ordenação desse vetor de 8 elementos demandará 7 iterações, conforme mostrado na tabela a seguir.

iteração	vetor	elemento selecionado	permutação	vetor ordenado até a posição
1	9 25 10 18 5 7 15 3	3	9 ↔ 3	
2	3 25 10 18 5 7 15 9	5	25 ↔ 5	1
3	3 5 10 18 25 7 15 9	7	10 ↔ 7	2
4	3 5 7 18 25 10 15 9	9	18 ↔ 9	3
5	3 5 7 9 25 10 15 18	10	10 ↔ 25	4
6	3 5 7 9 10 25 15 18	15	15 ↔ 25	5
7	3 5 7 9 10 15 25 18	18	25 ↔ 18	6
	3 5 7 9 10 15 18 25			8

Observe que a última iteração (sétima) encerra a classificação, pois se as $n-1$ menores chaves do vetor estão em suas posições definitivas corretas, então a maior (e última) automaticamente também estará.

Implementação

Apresentação em sala de aula.

Análise do desempenho

Apresentação em sala de aula.

Método de Partição e Troca - Quicksort

O desempenho do Quicksort é tão espetacular que seu inventor o chamou assim, o que quer dizer “ordenação rápida”. De fato, comparado com os demais métodos, é o que apresenta, em média, o menor tempo de classificação. Isto, porque, embora tenha um desempenho logarítmico como muitos outros, é o que apresenta menor número de operações elementares por iteração. Isto significa que, mesmo que tenha que efetuar uma quantidade de iterações proporcional a $n \log_2 n$, cada uma delas será mais rápida.

Implementação

Este método de ordenação consiste em dividir a área de ordenação em duas partições, de modo que os elementos da partição da esquerda sejam menores ou iguais aos elementos da direita. O método é então aplicado de forma recursiva sobre cada partição.

Algoritmo apresentado em sala de aula.

Exemplo e análise do desempenho

Apresentação em sala de aula.

6 MÉTODOS DE PROJETO DE ALGORITMOS

Para a maioria dos problemas, as técnicas simples de Análise Estruturada ou de Análise Orientada a Objetos aliadas a um pouco de bom senso e conhecimentos sobre Estruturas de Dados são suficientes para que nós possamos elaborar um algoritmo para resolver este problema.

Exemplos:

- Criar um sistema de contabilidade para uma empresa,
- Criar um programa que gerencie um estoque de um depósito.

Existem alguns problemas porém, onde o caminho a ser seguido para a solução não é tão direto. São problemas onde os algoritmos para resolvê-los ou parte deles não são tão óbvios.

Estes são problemas, onde nós temos de aplicar técnicas de projeto de algoritmos mais elaboradas para resolvê-los.

Exemplos:

- Criar um algoritmo eficiente para ordenar 50.000 endereços,
- Criar um algoritmo eficiente para distribuir tarefas entre máquinas de uma empresa de maneira a reduzir o tempo de ociosidade dessas máquinas,
- Criar um algoritmo eficiente para compactar arquivos com determinadas características,
- Criar um algoritmos de roteamento aproximado para uma empresa de entrega de pacotes.

Para a solução de problemas assim, existem conjuntos de técnicas que dividem os problemas em classes a apresentam linhas gerais para o projeto de um algoritmo.

Este item examina alguns métodos de projeto de algoritmos. Os métodos considerados são: Divisão e Conquista, Método Guloso e Programação Dinâmica. Esses três métodos baseiam-se na idéia de decomposição de problemas complexos em outros mais simples, cujas soluções serão combinadas para fornecer uma solução para o problema original. Eles diferem na maneira de proceder: o primeiro – mais geral – é recursivo, e os outros dois – geralmente usados em problemas de otimização combinatória – são iterativos.

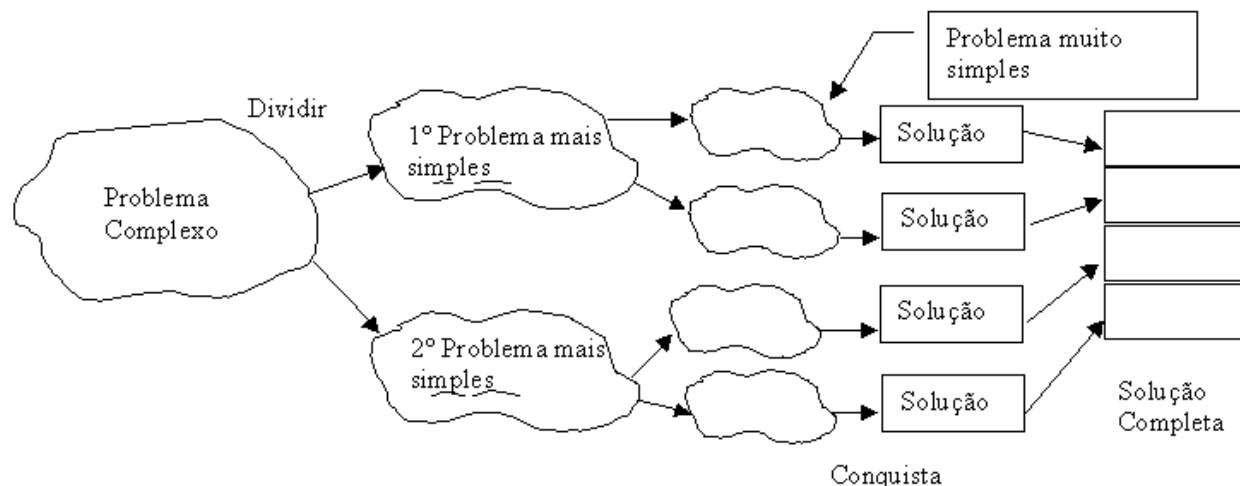
Divisão e conquista

Historicamente, o termo “divisão e conquista” foi originado pelos generais napoleônicos (1800-1840) que aplicavam uma estratégia de dividir o exército inimigo em vários subexércitos separados, para poder vencer cada uma das partes facilmente. O método de desenvolvimento de algoritmos por divisão e conquista reflete esta estratégia militar.

A estratégia de divisão e conquista para resolução de problemas – através de decomposições e recombinações – fornece um método poderoso e versátil para o projeto de algoritmos. Alguns exemplos, ilustrando a grande aplicabilidade de divisão e conquista, podem ser dados por algoritmos:

- de busca em tabelas, como buscas sequencial e binária;
- de ordenação, como ordenação por seleção, por intercalação (*mergesort*) e por particionamento (*quicksort*);
- para multiplicação de matrizes (de matrizes e de números binários, por exemplo);
- para seleção (para determinar o máximo e o mínimo, etc.).

A estratégia de divisão e conquista opera decompondo um problema em subproblemas independentes, resolvendo-os e combinando as soluções obtidas em uma solução para o problema original. Esse processo recursivo de decomposições e recombinações funciona da seguinte maneira. Dada uma entrada, se ela é suficientemente simples, obtemos diretamente uma saída correspondente. Caso contrário, ela é decomposta em entradas mais simples, para as quais aplicamos o mesmo processo, obtendo saídas correspondentes, que são então combinadas em uma saída para a entrada original.



Exemplo 1: Busca binária

Seja $L = (M_1, M_2, \dots, M_n)$ uma lista de n elementos e consideremos o problema, dado um elemento x , verificar se x está presente em L ou não. Se x está presente, deseja-se determinar o índice j tal que $M_j = x$; senão, digamos que $j = 0$ será uma resposta aceitável.

Os elementos M_i na lista L podem pertencer a um conjunto bem simples, como, por exemplo, o conjunto dos números inteiros, ou, de um modo geral, podem ser registros, de arquivos ou bancos de dados, relativamente longos. Desta forma, para se medir a rapidez de um algoritmo para resolver este problema escolhe-se, justificadamente, a operação de comparação entre dois elementos.

Um algoritmo trivial para resolver este problema consiste em comparar, iterativamente, x com M_1 , x com M_2 , e assim por diante, até se encontrar um M_j tal que $M_j = x$, ou até se concluir que tal M_j não existe, e então, a resposta será negativa, fazendo-se $j = 0$. É óbvio que este algoritmo tem complexidade pessimista da ordem $O(n)$.

Suponhamos que os elementos M_i da lista L pertencem a um conjunto com relação de ordem linear. E então, podemos aplicar qualquer algoritmo de ordenação sobre L , este problema seria resolvido por um algoritmo $O(n \log n)$. Tendo L ordenado, veremos um algoritmo desenvolvido por divisão e conquista que resolve o problema em tempo $O(n \log n)$.

Algoritmo busca_binária: para verificar se um dado elemento está presente numa lista.

Entrada: $(x, n, L = (M_1, M_2, \dots, M_n))$, onde $n \leq 1$ e x, M_1, M_2, \dots, M_n são elementos de um conjunto com relação de ordem linear; L está em ordem crescente;

Saída: (resposta, j), onde j é o índice de um elemento $M_j = x$ em L , se a resposta = “presente”, e $j = 0$ se a resposta = “ausente”.

```

se n=1 então
  se x = Mn então
    retorne("presente",n)
  senão
    retorne("ausente",0)
fim se;
senão
  k := teto((n+1)/2);
  se x < Mk então
    (resposta,j) := busca_binária(x, k-1, M1, ..., Mk-1)
  senão
    (resposta,j) := busca_binária(x, n-k+1, Mk, ..., Mn)
    j := j+k-1;
  fim se;
  retorne(resposta,j);
fim se

```


$T(1) = 1$, $T(n) = 1 + T(\text{teto}(n/2))$, para $n > 1$
 $T(n) = O(\log n)$

Exemplo 2: Máximo e mínimo de uma lista

O algoritmo MaxMin4 visto anteriormente, utiliza a recursividade para dividir o a lista inicial em duas listas de comprimentos aproximadamente iguais, garantindo maior rapidez em relação aos algoritmos triviais que consideram o primeiro elemento como Max e Min temporário, comparando-os com os seguintes e atualizando-os, quando necessário.

Método guloso

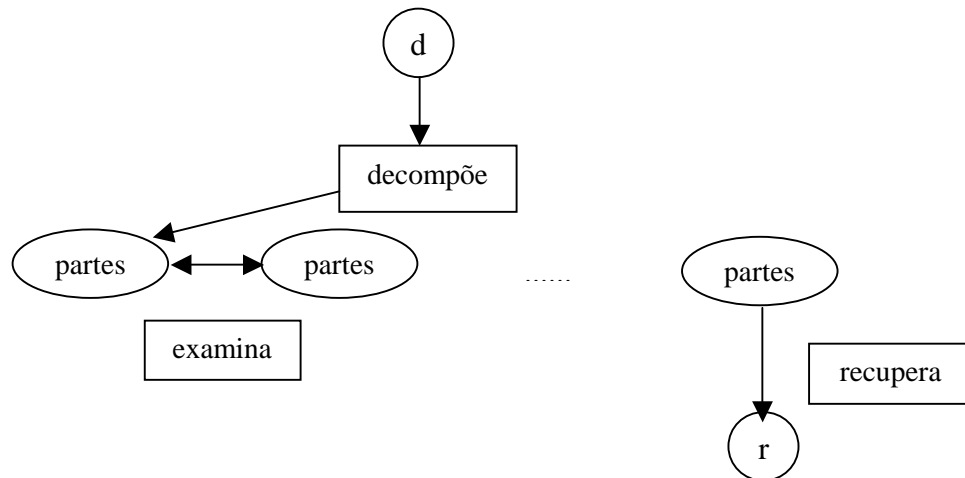
A palavra “guloso” do nome deste método é uma tentativa de traduzir “greedy”, que algumas preferem traduzir por “ganancioso”.

O método guloso é útil principalmente para resolver problemas de otimização combinatória, cuja solução pode ser alcançada por uma seqüência de decisões. Em cada passo, após selecionar um elemento, decide-se se ele é viável – vindo a fazer parte da solução – ou não. Após uma seqüência de decisões, a solução do problema é alcançada.

Na seqüência de decisões, nenhum elemento é examinado mais de uma vez: ou ele fará parte da saída, ou será descartado. Frequentemente, a entrada do problema vem classificada para selecionar, a cada passo, um elemento da entrada e decidir incluí-lo na solução ou não. A cada ponto, onde temos de tomar uma decisão, tomamos aquela que nos parece ser a melhor neste momento, sem analisar se ela vai ou não nos prejudicar no futuro (caminho do menor esforço).

É a técnica de utilizar a *optimalidade* local para resolver o problema: Observando o problema como um todo, a cada ponto onde uma decisão é tomada, temos um estado do problema. A técnica “gulosa” consiste em só observarmos o contexto “local” deste estado para escolhermos o próximo estado.

Dois exemplos típicos de algoritmos gulosos são os algoritmos para a Árvore Expandida de Custo Mínimo de Kruskal e de Dijkstra. Em ambos os algoritmos, escolhemos o próximo vértice somente pelo critério “qual é o vértice livre que está mais próximo” ?.



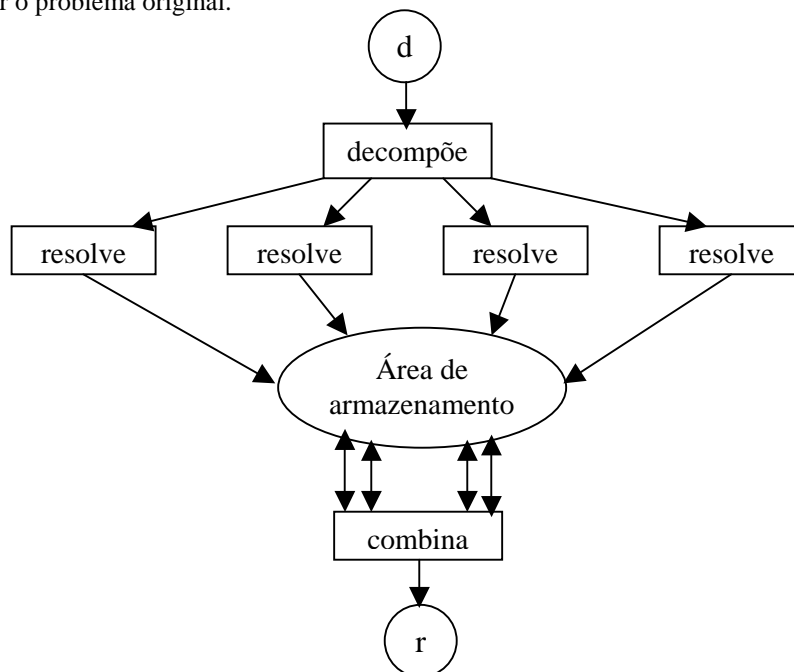
Os exemplos a seguir serão apresentados em sala de aula.

Exemplo 1: Árvore expandida de custo mínimo (análise em Terada, 1991, p. 151)
 (Algoritmo de Kruskal, visto em sala de aula)

Exemplo 2: Caminho de custo mínimo (análise em Terada, 1991, p. 158)
 (Algoritmo de Dijkstra, visto em sala de aula)

Programação dinâmica

A estratégia de programação dinâmica costuma ser aplicada a certos problemas de otimização. A programação dinâmica aborda um dado programa, dividindo-o em subproblemas mínimos, solucionando-os os subproblemas, guardando os resultados parciais, combinando subproblemas menores e sub-resultados para obter e resolver problemas maiores, até recompor e resolver o problema original.



Note que o problema é decomposto uma única vez e, além disso, os subproblemas menores são gerados antes dos subproblemas maiores. Assim, esse método é chamado ascendente ao contrário dos métodos recursivos, que são chamados descendentes.

Exemplo 1: Multiplicação de matrizes

Considere o seguinte exemplo:

$$M := \begin{matrix} M_1 \\ 100 \times 3 \end{matrix} \times \begin{matrix} M_2 \\ 3 \times 10 \end{matrix} \times \begin{matrix} M_3 \\ 10 \times 50 \end{matrix} \times \begin{matrix} M_4 \\ 50 \times 30 \end{matrix} \times \begin{matrix} M_5 \\ 30 \times 5 \end{matrix}$$

Calcular M por meio de $\{(M_1 \times M_2) \times M_3\} \times M_4 \times M_5$, requer: $(100 \times 10 \times 3) + (100 \times 10 \times 50) + (100 \times 50 \times 30) + (100 \times 30 \times 5)$, ou seja, 218.000 operações.

Já agrupando de outra maneira as multiplicações, obtém-se um resultado bem diferente. Por exemplo, calculando M por $M_1 \times \{M_2 \times [(M_3 \times M_4) \times M_5]\}$, usamos $(10 \times 50 \times 30) + (10 \times 30 \times 5) + (3 \times 10 \times 5) + (100 \times 3 \times 5)$, ou seja, 18.150 operações.

Esse exemplo ilustra bem o fato de que a ordem em que são realizadas as multiplicações pode influir substancialmente no número total de operações requeridas. O problema consiste, então, em determinar uma sequência ótima de multiplicações, i. e. com custo mínimo.

Um algoritmo que enumere todas as seqüências possíveis de multiplicações, calcule os respectivos números totais de operações requeridas, para, só então, escolher uma seqüência ótima terá complexidade exponencial em n (número de matrizes), o que é inviável na prática para n relativamente grande. Utilizando a programação dinâmica, poderemos diminuir a ordem de complexidade para um ordem polinomial.

Algoritmo MultMat: para determinação do custo mínimo para a obtenção de um produto de n matrizes.

Entrada: (b_0, b_1, \dots, b_n) , onde b_{i-1} e b_i são as dimensões de uma matriz M_i

Saída: (m_{1n}) , o custo mínimo para se obter o produto $M_1 * M_2 * \dots * M_n$

```
para i := 1 até n faça
    m[i,i] := 0;
para u := 1 até n-1 faça
    para i := 1 até n-u faça
        j := i + u; // u = j - i
        m[i,j] := min ( (m[i,k] + m[k+1,j]) + (b[i-1] x b[k] x b[j]))
                      i<=k<j
retorne m[1,n];
```

Análise da complexidade: apresentação em sala de aula.

Exemplo 2: Caixeiro Viajante (Ziviani, 1998 – p. 17 e Terada, 1991 – p. 182)

Apresentação em sala de aula.

6 CLASSES DE PROBLEMA P, NP E NP-COMPLETO

Apresentação em sala de aula

REFERÊNCIAS BIBLIOGRÁFICAS

AZEREDO, Paulo. Métodos de Classificação de Dados. Rio de Janeiro : Campus, 1996.

FEOFILOFF, Paulo. Análise de Algoritmos. 1999. Disponível em: <<http://www.ime.usp.br/~pf/mac338-1999/1999.htm>>. Acesso: outubro 2002.

FIGUEIREDO, Jorge C. A. de. Análise de Algoritmos. 2001. Disponível em: <<http://www.labpetri.dsc.ufpb.br/~abrant/atal20012>>. Acesso: outubro 2002.

SILVA Neto, Pedro Soares da. Projeto e Análise de Algoritmos. 2001. Disponível em: <<http://ipanema.ime.eb.br/~sneto/paa/>>. Acesso: outubro 2002.

TERADA, Ruto. Desenvolvimento de Algoritmos e Estruturas de Dados. São Paulo : Makron Books, 1991.

TOSCANI, Laura Vieira; VELOSO, Paulo A. S. Complexidade de algoritmos: análise, projeto e métodos. Porto Alegre : Instituto de Informática da UFRGS : Sagra Luzzatto, 2001.

ZIVIANI, Nivio. Projeto de Algoritmos. São Paulo : Pioneira, 1999.