Capítulo I

Complexidade de Algoritmos

"Deus fez os números inteiros,
todo o resto é obra do homem"
Leopold Kronecker

I.1 - INTRODUÇÃO:

Informalmente falando, um algoritmo se caracteriza, essencialmente, por uma seqüência finita de "passos elementares" voltados para a resolução de um determinado problema. Este algoritmo pode ser visto como uma função *f(.)* onde *S = f(E)*, sendo *E* uma entrada qualquer do problema, e *S*, uma solução deste problema. Segundo Knuth, a palavra "algoritmo" é derivada do nome "*al-Khowârizmî*", um matemático persa do século IX. Apesar do primeiro computador ter sido construído na década de 1940, os algoritmos já existiam há muitos anos e eram dedicados, primordialmente, à solução de problemas aritméticos.

Pode-se identificar, entre os modelos de computador atuais, *três* tipos de algoritmos que se diferenciam, basicamente, na maneira como resolvem e tratam seus problemas: os algoritmos determinísticos, randômicos ou probabilísticos e os algoritmos não-determinísticos.

O tempo de execução de um algoritmo determinístico para uma mesma entrada *E* é sempre fixo, ou seja, sucessivas repetições do algoritmo aplicadas a *E* resultam sempre, em uma mesma saída sem que ocorra variação de seu tempo de processamento. Existem situações entretanto, onde a saída e/ou o tempo de processamento poderão ser distintos a cada repetição do algoritmo. Trata-se dos algoritmos randômicos (ou probabilísticos) estudados mais adiante no capítulo III. Nos algoritmos randômicos um número *finito* processadores geram valores aleatoriamente, permitindo, desta forma, que diferentes respostas sejam geradas a cada repetição. Nos algoritmos não-determinísticos, assume-se que um número *infinito* de processadores seja utilizado, cada um deles, gerando valores aleatoriamente. Como abordado mais adiante (seção I.9), esse conceito abstrato de algoritmo será importante na classificação dos problemas de decisão quanto a seu grau de dificuldade. Salvo indicação contrária, consideraremos apenas algoritmos determinísticos.

Suponha que sejam dados um problema π e um algoritmo determinístico que o resolva após um número finito de passos. Normalmente, deseja-se responder questões relativas à eficiência deste algoritmo. Entretanto, o que significa dizer que um algoritmo *é* ou *não* eficiente? Ou ainda, quais serão os parâmetros observados na avaliação de seu desempenho? Pode-se, evidentemente, considerar o tempo e o espaço (memória) exigidos no processamento como importantes fatores na análise de eficiência desse algoritmo.

O processo mais utilizado na avaliação de algoritmos até meados da década de 60 foi a chamada "análise de comportamento médio", que consiste fundamentalmente, em avaliar o tempo de CPU necessário na resolução de determinadas instâncias de um problema. O comportamento médio é extremamente útil, pois fornece informações precisas sobre o desempenho do algoritmo

para aquelas instâncias em particular. Esta análise entretanto se torna bastante inadequada se for considerada uma entrada qualquer para o problema. Outras dificuldades são também incorporadas, além de depender (na maioria dos casos) de uma infinidade de dados de entrada, o tempo de processamento dependerá evidentemente da máquina utilizada, do código gerado pelo compilador, e ainda, de como evolui o tempo de processamento quando instâncias cada vez maiores para o problema forem consideradas. Pode-se constatar por exemplo, que a taxa de variação entre o tempo e o tamanho de uma entrada qualquer será fator determinante no desempenho de um dado algoritmo.

Neste capítulo serão apresentadas algumas ferramentas utilizadas na análise de desempenho (ou complexidade) de um algoritmo e que visam contornar as dificuldades expostas acima. A idéia será buscar uma medida de complexidade (função de complexidade) que independa do computador e da natureza dos dados de entrada. Como discutido posteriormente, essa função será definida no tamanho de uma instância qualquer e retornará valores no tempo. Mais formalmente, espera-se obter uma função de complexidade (de tempo) $T:tam(\pi)\rightarrow Z^+$ onde $tam(\pi)$ representa o tamanho de uma instância *I* qualquer (definida na seção I.4) e Z^+ representa o número total de unidades de tempo.

Analogamente, pode-se fazer uma analise da complexidade de espaço de um determinado algoritmo. Nestes casos, deseja-se obter uma função $E:tam(\pi)\rightarrow Z^+$, onde $tam(\pi)$ representa o tamanho do problema e Z^+ o número de posições de memória exigida pela máquina na execução do algoritmo.

I.2 - ALGORITMOS E PERFORMANCE

Em função do grande desenvolvimento no *hardware* dos computadores modernos digitais somos levados, mesmo que inconscientemente, a ignorar a importância dada à performance de algoritmos. Na verdade, com a evolução tecnológica e a possibilidade de se trabalhar com problemas cada vez maiores a preocupação com o desempenho dos algoritmos se torna fundamental. Nesta seção, são apresentados alguns exemplos onde a taxa de variação entre o tempo de processamento e o tamanho do problema são considerados. Pode-se constatar que, em um casos, o tempo de processamento cresce exponencialmente enquanto o tamanho do problema cresce apenas linearmente. Nestas situações, o ganho obtido com o equipamento pode se tornar irrelevante.

Suponha que 5 algoritmos distintos sejam elaborados para resolução de um determinado problema. A complexidade de cada algoritmo é apresentada na Figura I.1. O parâmetro *n* representa o tamanho do problema. As funções de complexidade associadas (coluna 2), indicarão o número de unidades de tempo utilizado para se processar uma entrada de tamanho *n*. Deseja-se responder, qual o tamanho máximo do problema (para cada algoritmo) de forma a resolvê-lo em um intervalo de tempo pré-determinado (p. ex., *1seg, 1min* ou *1 hora*). Suponha que um *milisegundo* seja a unidade de tempo utilizada.

Note que, no algoritmo exponencial, apenas problemas pequenos são considerados dentro da faixa de tempo estipulada. Os demais algoritmos (Figura I.1) são *polinomiais*. Cabe notar que, embora a função de complexidade de tempo $T(n) = n\log n$ não seja polinomial (algoritmo A2), ela também será considerada "polinomial" já que pode ser limitada superiormente por um polinômio de grau $k \geq 2$.

Como discutido mais adiante, a grande maioria dos algoritmos considerados são executados em tempo polinomial ou exponencial no tamanho da entrada. Há funções entretanto, onde a taxa de crescimento é superior à polinomial mas inferior à exponencial (p. ex., $f(n)=n^{\log n}$). Pode-se ter também funções onde a taxa de crescimento seja superior à exponencial (p. ex., $f(n) =O(n!)$).

ALGORITMOS	COMPLEXID. DE TEMPO	TAMANHO MÁXIMO DO PROBLEMA		
		1 Seg.	1 Min.	1 Hora
A1	n	1000	$6x10^4$	$3,6x10^6$
A2	$nlogn$	140	4893	$2x10^5$
A3	n^2	31	244	1897
A4	n^3	10	39	153
A5	2^n	9	15	21

Figura I.1: Tamanho máximo de um problema

Considere agora uma geração futura de computadores dez vezes mais rápido que os atuais. O quadro da Figura I.2 mostra como cresce o tamanho desses problemas à medida que a velocidade de processamento é aumentada.

Observe que no algoritmo A_5 apenas um acréscimo de 3,3 unidades no tamanho do problema para um computador 10 vezes mais rápido! Esses dados nos mostram a importância que a análise de complexidade exerce sobre o tempo de processamento. Deve-se lembrar entretanto que, algoritmos exponenciais quando aplicados a problemas pequenos podem ter um desempenho melhor que um algoritmo polinomial. Suponha por exemplo *dois* algoritmos de complexidades 10^5n e 2^n respectivamente. Para $n \leq 21$, o algoritmo exponencial terá um desempenho superior ao algoritmo polinomial. Verifique!

ALGORITMOS	COMPLEXID. DE TEMPO	Tam. Máx. Comp. Atuais	Tam. Máx. Comp. Futuros
A1	n	S_1	$10.S_1$
A2	$nlogn$	S_2	$\approx S_2$
A3	n^2	S_3	$3,16.S_3$
A4	n^3	S_4	$2,15.S_4$
A5	2^n	S_5	$S_5+3,3$

Figura I.2: Tamanho máximo de um problema

Outros fatores devem ser discutidos na avaliação da complexidade como, por exemplo, o desempenho médio do algoritmo (complexidade de caso médio). Um algoritmo de complexidade exponencial nem sempre executará um número exponencial de passos para qualquer entrada. Um exemplo clássico é o algoritmo simplex para o problema de programação linear (para maiores detalhes vide Bazaraa et al.[1990]) . O algoritmo simplex tem em média, processamento polinomial, embora possua complexidade teórica exponencial.

Antes de passar a uma discussão mais formal sobre o tamanho de um problema e das funções complexidade é fundamental definir, mais precisamente, um modelo teórico de computador. Nele, serão abstraídas as principais características de uma máquina mais sofisticada.

I.3 - MÁQUINA RAM (Random Access Machine)

Como avaliar o tempo de execução de um algoritmo, sem executá-lo propriamente em uma determinada máquina? Antes de contar o número total de passos realizados por um algoritmo, deve-se, inicialmente, observar que esses passos são diferentes entre si. O tempo de uma adição por exemplo, será diferente do tempo gasto em uma multiplicação, o tempo de uma divisão será

diferente de uma comparação e assim por diante. Portanto, é importante que se considere o tempo de cada instrução separadamente somando-as apenas ao final do processo. Deve-se lembrar ainda, que o tempo de processamento de uma determinada instrução não será o mesmo em máquinas diferentes. Portanto, é fundamental que se construa um modelo teórico de computador e que seja capaz de gerar informações aproximadas sobre o tempo de processamento independentemente do computador utilizado.

O modelo *RAM* (*Random Access Machine*) é um modelo hipotético de máquina constituído de duas fitas (para entrada e saída de dados respectivamente), uma unidade de controle e processamento e uma memória (Figura I.3). A natureza exata das instruções não é relevante, sendo semelhantes às encontradas em um computador real. Haverá operações de entrada e saída, transferência de informação entre memória e registradores, endereçamento indireto, operações aritméticas e desvios. Cada registrador ou posição de memória poderá armazenar e acessar valores inteiros como uma unidade, sem ter acesso à representação do número.

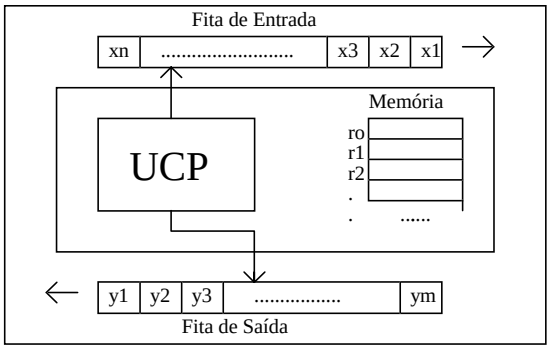

Figura I.3: Modelo de uma máquina RAM

As operações aritméticas permitidas são +, -, x, /. Um algoritmo poderá ainda comparar dois valores e avaliar a raiz quadrada de um inteiro positivo.

Obviamente outros modelos teóricos de computador podem ser utilizados como, por exemplo, a máquina de Turing. Entretanto, uma função de complexidade obtida em qualquer outro modelo de computador, deverá ser expressa equivalentemente em uma máquina RAM (Hopcroft[1979]). Assim, funções polinomiais/exponenciais de tempo nos demais modelos, deverão ser polinomiais/exponenciais no modelo RAM.

O modelo RAM pode ainda ser subdividido em *dois* tipos de máquina. No primeiro, máquina RAM com custo uniforme, cada instrução é executada em uma unidade de tempo. Este modelo é considerado excessivamente poderoso já que não pode ser simulado polinomialmente por uma máquina de Turing. Isto ocorre pois, com a operação de multiplicação, inteiros extremamente grandes podem ser gerados (Motwani&Raghavan[1995]). Eliminando-se as demais operações aritméticas, além da soma e subtração, o modelo RAM se torna polinomialmente equivalente à máquina de Turing. No segundo modelo, máquina RAM com custo logarítmico, cada instrução é processada em um tempo proporcional ao logaritmo de seus operandos. Neste caso, o novo modelo incluindo todas as operações aritméticas descritas acima, será polinomialmente equivalente à máquina de Turing (Motwani&Raghavan[1995]).

Por razões de simplicidade e clareza, utiliza-se normalmente a máquina RAM com custo uniforme. Assim, as fitas de entrada e saída serão formadas por uma seqüência de células, cada uma delas podendo armazenar um número inteiro de tamanho arbitrário. Além disso, não haverá limite para o número de células na memória. Esta abstração será possível quando:

- o tamanho do problema for suficientemente pequeno para ser armazenado na memória principal,
- os inteiros utilizados nos cálculos forem suficientemente pequenos para serem armazenados em uma única palavra do computador, em outras palavras, o tamanho dos operandos será limitado polinomialmente no tamanho da entrada,

Adicionalmente, a máquina RAM poderá gerar, em um único passo, valores aleatórios distribuídos uniformemente em um conjunto limitado no tamanho do problema. (máquina RAM probabilística). Para maiores detalhes sobre este assunto vide Papadimitriou[1994].

Formulado nosso modelo teórico de máquina passemos agora para a definição de tamanho de um problema:

I.4 - TAMANHO DE UM PROBLEMA

Considere π um problema qualquer, ou seja, um enunciado que indique quais propriedades a solução deverá satisfazer e uma descrição formalizada de todos os seus parâmetros (dados de entrada). Fazendo-se uma associação de valores (numéricos ou não) a esses parâmetros define-se uma *instância* de π. Considere o seguinte exemplo:

Exemplo I.1: (Problema da Mochila)

Neste problema, o objetivo será encher uma mochila de capacidade M com no máximo n objetos distintos. Sabendo-se que cada objeto tem um valor diferente, como encher a mochila (sem ultrapassar sua capacidade máxima) maximizando o valor dos objetos presentes em seu interior? Mais formalmente, tem-se o problema:

$$\textit{maximizar} \sum_{j=1}^{n} c_j x_j$$

$$\textit{sujeito a}: \begin{cases} \sum_{j=1}^{n} p_j x_j \le M \\ x_j = 0 \ \textit{ou}\ 1 \qquad \textit{onde} \quad j = 1,..,n \end{cases}$$

onde n, b, c_j e a_j são parâmetros inteiros positivos e x_j é variável de decisão. Os parâmetros são especificados abaixo:

c_j - custo (valor) da peça j
M - capacidade total da mochila (peso)
n - número total de objetos
p_j - peso da peça j

Assim, uma instância I para o problema da mochila será definida por uma $(2n+1)$-upla de inteiros positivos $(c_1,...,c_n,p_1,...,p_n,M)$ onde $p_j \le M$ e $j=1,..,n$. •

O conceito de tamanho de problema, será fundamental na definição de função de complexidade. Assim, dado um problema π e uma instância I qualquer associada, o *tamanho* desse problema (denotado por $tam(\pi)$), será definido como um número inteiro positivo que representa a quantidade dos dados de entrada presentes em I.

A noção exata de quantidade dependerá do problema considerado. Em algumas situações, deseja-se determinar o número total de elementos presentes em uma dada entrada sem se preocupar com os valores assumidos por cada elemento dessa instância. Em outras, o valor de cada elemento

presente na entrada será considerado. Para exemplificar essa situação considere os *dois* exemplos seguintes. Em ambos, uma máquina *RAM* com custo uniforme estará sendo utilizada.

Exemplo I.2 (Busca em um lista desordenada)**:**

Seja dado uma lista $L=\{a_1,a_2,...,a_n\}$ com n elementos inteiros. Suponha que L esteja desordenada e que se deseje encontrar um elemento inteiro x nesta lista. Note que n representará o tamanho do problema. A "natureza" dos elementos presentes na lista (valor numérico dos elementos) não será importante, e sim, a quantidade total de seus elementos (parâmetro do problema)! •

Exemplo I.3 (Fatorial)**:**

Suponha que se deseje calcular o fatorial de um valor inteiro n positivo. Uma entrada para este problema consistirá apenas na leitura do número n. Note, entretanto, que o tamanho do problema será n e não 1 (*um*)! Informalmente falando, estaremos interessados apenas na "natureza" da instância obtida e *não* em sua quantidade. Neste caso, apenas o valor da entrada será fator determinante no cálculo de complexidade. •

Como discutido na seção anterior, pode-se definir o tamanho de um problema π, como sendo o número total de *bits* utilizados na codificação de sua entrada (máquina *RAM* com custo logarítimico). Esta pode ser uma medida mais realista na multiplicação de *dois* inteiros por exemplo. Para evitar conflitos na representação do tamanho do problema, indicar-se-á, em cada caso, qual modelo de máquina esta sendo utilizado. Salvo indicação contrária, apenas a máquina *RAM* com custo uniforme será considerada.

Algumas vezes será mais apropriado descrever o tamanho de uma entrada por *dois* ou mais parâmetros (ao invés de *um*). Se a entrada de determinado algoritmo é um grafo (vide Ahuja et al.[1993]), o tamanho do problema poderá ser descrito pelo número de vértices e arestas desse grafo.

Considere que sejam dados um problema π e um algoritmo A que o resolva. O problema que se coloca agora é: definido o tamanho de π como obter uma função de complexidade para o algoritmo A? Esta questão é tratada na seção seguinte:

I.5 - FUNÇÕES DE COMPLEXIDADE

Como comentado anteriormente na seção I.1, uma função de *complexidade de tempo* será do tipo $T:tam(\pi)\rightarrow Z^+$, onde $tam(\pi)$ representa o tamanho de uma instância I qualquer de um problema π, e Z^+, representa o número total de unidades de tempo utilizados no processamento de I.

Será entretanto que instâncias distintas de mesmo tamanho possuem sempre o mesmo tempo de processamento? Obviamente que não. Suponha, no exemplo I.2 acima, que o tamanho de uma instância qualquer da lista L seja sempre n. Observe que o tempo de busca do elemento x nesta lista será função de sua posição na lista. Ou seja, se $x = a_1$, bastará uma comparação do procedimento de busca para resolução do problema. Por outro lado, se $x = a_n$ ou $x \notin L$, n comparações serão realizadas. Assim, dada uma instância I de tamanho n seu tempo de processamento não será computado de forma única, não definindo portanto uma função! Discutimos abaixo uma maneira de contornar esta situação?

Considere A um algoritmo qualquer utilizado na resolução do problema π, e $M=\{E_1,E_2,...,E_m\}$ um conjunto finito com m entradas distintas de mesmo tamanho (o conjunto M também pode ser infinito). Seja $T(E_k)$, o tempo de A na computação da entrada E_k e $Pr(E_k)$, a probabilidade de ocorrência de E_k. Tem-se então *três* medidas de complexidade, a saber:

a) Complexidade de Pior Caso: $T_W = \max_{1 \le k \le m} \{T(E_k)\}$

b) Complexidade de Melhor Caso: $T_B = \min_{1 \le k \le m} \{T(E_k)\}$

c) Complexidade de Caso Médio: $T_M = \sum_{1 \le k \le m} \Pr(E_k).T(E_k)$

A complexidade de melhor caso representa, evidentemente, a situação ideal dos dados de entrada do algoritmo não sendo considerada, portanto, uma medida apropriada de complexidade na maioria dos casos.

A complexidade de caso médio como definido acima tem obviamente grande interesse prático. Entretanto, para vários problemas, sua estimativa se torna bastante trabalhosa já que é importante que se defina, de antemão, uma distribuição estatística associada aos dados de entrada. Além disso, vários problemas admitem uma quantidade infinita (enumerável ou não) de dados de entrada e a determinação dos tempos de processamento a elas associadas se torna, por vezes, impraticável.

Na complexidade de pior caso, uma cota superior para o tempo de processamento será determinada. Será também de grande interesse prático e poderá ser calculada mais facilmente que a complexidade de caso médio.

Note que, escolhendo um dos três itens acima, será possível determinar o tempo de processamento em função do tamanho do problema.

Notação: Para simplificar a notação acima, salvo indicação contrária, assumi-se que $T=T_W$, $T=T_B$ ou $T=T_M$ na análise de complexidade. •

Antes de tentar determinar uma expressão para a função de tempo *T(.)*, deve-se avaliar, individualmente, o tempo de cada instrução σ_j do algoritmo *A*. Supondo $t(\sigma_j)$, o número de unidades de tempo gasto na execução de σ, o tempo total na execução do algoritmo será dado por:

$$T = \sum_j k_j . t(\sigma_j) \qquad \text{(I)}$$

onde:

σ_j - *j-ésima* instrução do algoritmo (máq. *RAM*),
k_j - número de execuções de σ_j
$t(\sigma_j)$ - tempo de execução de σ_j

Note que o número de execuções k_j da instrução σ_j será uma função do tamanho do problema (ou seja, $k_j = k_j\ (tam(\pi))$). Na complexidade de pior caso, k_j irá representar o número máximo de execuções (passos) de σ_j. É fácil ver portanto que o tempo *T(.)* será, de fato, função do tamanho do problema. Assim, a complexidade de tempo, será o maior número de passos possível (no pior caso) para execução completa de *A*.

No critério de custo uniforme cada instrução RAM requer uma unidade de tempo e cada registro requer uma unidade de espaço. Portanto, a função de tempo (I) pode ser rescrita de maneira mais simples como abaixo:

$$T = \sum_{j=1}^{q} k_j , \qquad \text{(II)}$$

onde *q* é o número total de instruções do algoritmo.

Será que a simplificação introduzida acima (onde $t(\sigma_j) = 1, \forall \sigma_j$) acarretará uma grande distorção no tempo de processamento? A proposição I.1 abaixo (deixada como exercício) mostra que esta distorção será de apenas uma constante.

Proposição I.1: Sejam $\sigma_1,\sigma_2,...,\sigma_m$ instruções de um algoritmo *A* com seus respectivos tempos de processamento $t(\sigma_1), t(\sigma_2),....,t(\sigma_m)$. Seja T_0, o tempo de processamento calculado com a utilização da função *(I)* acima. Fazendo-se $t(\sigma_1)=t(\sigma_2)=....=t(\sigma_m)=1$ obtêm-se um novo tempo de processamento T onde $T=c.T_0$ e c é constante. •

A função de complexidade *T(.)* (representada por (I) ou (II)) será chamada de função de *complexidade local de tempo*. Analogamente, a *complexidade local de espaço* de um algoritmo indica a quantidade total de memória exigida por um algoritmo durante sua execução. No critério de Custo Uniforme essa memória irá corresponder ao número total de registros (células) utilizadas. Lembre-se que, neste caso, cada registro armazena um inteiro de tamanho arbitrário. Pode-se representar a complexidade de espaço por uma função $E:tam(P)\rightarrow Z^+$ onde *tam(P)* representa o tamanho de nosso problema e Z^+, o número total de unidades de espaço utilizados.

O exemplo I.5 a seguir, trata das complexidades locais de tempo e espaço no pior caso.

Exemplo I.5: (Pior Caso)

Considere $p(x) = a_n x^n + a_{n-1}x^{n-1}+...+a_0$ um polinômio qualquer de grau *n* com coeficientes inteiros. Dado $x_0 \in Z^+$, deseja-se calcular simplesmente o valor $p(x_0)$.

Considere inicialmente o seguinte algoritmo:

Algoritmo I.1: Cálculo de $p(x_0)$:
Início
leia $(n, a_n, a_{n-1},..,a_0 , x_0)$;
$p \leftarrow a_0$;
para $j:=1$ **até** n **faça**
$p \leftarrow p + a_j x_0^j$;
fim;
imprima (p);
fim.

Figura I.4: Cálculo de $p(x_0)$

No cálculo de x^j (potenciação), $j-1$ multiplicações são executadas. Tem-se ainda, na estrutura de repetição mais 4 operações a saber: incremento de *j*, comparação com *n*, adição e multiplicação. Dessa forma o "*loop*" interno será executado *n* vezes e a cada execução serão feitas $(3+j)$ operações. No final das repetições quando $j=n$ (no *para... faça...*), mais um incremento de *j* e uma comparação com *n* serão realizadas. Assim o tempo total será:

$$T(n) = 2 + \sum_{j=1}^{n}(3 + j) = n\left(\frac{n+1}{2}\right) + 3n + 2 = \frac{n^2 + 7n + 4}{2}$$

Na complexidade de espaço são necessários $n+1$ registros para armazenar as constantes a_n, $a_{n-1},..,a_0$, *três* registros para se armazenar n,x_0 *e p*, e um acumulador. Necessita-se portanto, de um total de $n+5$ posições de memória. Assim, $E(n)=n+5$.

Pode-se obter uma melhora na complexidade local de tempo do algoritmo utilizando-se o método de Horner. Neste método, o polinômio *p(.)* será rescrito na forma de um "ninho" de

binômios. Abaixo é apresentado um quadro para o cálculo de $p(x_0)$ considerando-se um polinômio de grau *3:*

x_0	a_3	a_2	a_1	a_0
	↓	a_3x_0	$a_2x_0 + a_3x_0^2$	$a_1x_0 + a_2x_0^2 + a_3x_0^3$
	a_3	$a_2 + a_3x_0$	$a_1 + a_2x_0 + a_3x_0^2$	$a_0 + a_1x_0 + a_2x_0^2 + a_3x_0^3$

Figura I.5: Tabela de Horner

Tem-se portanto: $p(x_0) = ((a_3x_0+a_2)x_0+a_1)x_0+a_0$. O algoritmo apresentado abaixo resolve o problema para um polinômio de grau *n*.

Algoritmo I.2: Método de Horner;
Início
leia $(n, a_n, a_{n-1},..,a_0 , x_0)$;
$p \leftarrow a_n$;
para $i:=n-1$ **até** *0* **faça**
$p \leftarrow a_i + p.x_0$;
fim;
imprima (*p*);
fim.

Figura I.6: Cálculo de p(x_0) através do método de Horner

Observe agora que *4* operações serão realizadas dentro da estrutura de repetição. Note ainda que ao final do *loop* serão executadas mais 2 operações (decremento e comparação). Logo, a complexidade local de tempo será igual a $T(n)=4n+2$.

Analogamente ao algoritmo anterior tem-se um comportamento linear para a complexidade local de espaço do algoritmo I.2 acima (verifique).

Na análise de complexidade, considera-se normalmente a complexidade de tempo de um determinado algoritmo. As conclusões obtidas sobre a complexidade de espaço são análogas e seguem diretamente. O próximo exemplo ilustra uma aplicação da complexidade de caso médio:

Exemplo I.6: (Complexidade de caso médio)

Suponha que se deseje encontrar o tempo médio de busca de um elemento *x* em uma lista $A=\{a_1,a_2,...,a_n\}$ com *n* elementos. Considere o algoritmo I.3 a seguir:

Algoritmo I.3: Busca;
Início
leia (*A,x*);
$a_{n+1} \leftarrow x$;
$i \leftarrow 1$;
enquanto $a_i \neq x$ **faça**
$i \leftarrow i+1$;
imprime solução *(A, i)*;
fim.

Figura I.7: Busca elemento em uma lista desordenada

Note que se *x* não pertence a lista então $x = a_{n+1}$. Como discutido acima, a função de complexidade de caso médio será:

$$T_M = \sum_{k=1}^{m} \Pr(E_k).T(E_k)$$

onde *m* é o número total de entradas de tamanho *n.*

Apesar de se ter um número infinito de entradas, será possível classificá-las em *n+1* classes distintas a saber:

$$E_1 = \{A / a_1 = x\}$$
$$E_2 = \{A / a_2 = x\}$$
$$\ldots\ldots\ldots\ldots$$
$$E_n = \{A / a_n = x\}$$
$$E_{n+1} = \{A / x \notin A\}$$

Note que E_i para $i=1,2,..,n$ representa o conjunto de todas as listas *A* tal que $a_i = x$ e E_{n+1}, o conjunto de todas as listas *A* tal que $x \notin A$. Observe portanto que na função de complexidade de caso médio tem-se $m=n+1$. Deve-se determinar agora as probabilidades e os tempos de execução de cada uma das $n+1$ entradas. Suponha por exemplo que $Pr(E_i)=Pr(E_j)$ para $i=1,..,n$ e $j=1,..,n$ (distribuição uniforme). Se *q* é a probabilidade de se ter $x \in A$ então:

$$\Pr(E_i) = q/n \text{ , para } i = 1,..,n$$
$$\Pr(E_{n+1}) = 1 - q$$

Analisando-se o algoritmo de busca acima conclui-se diretamente que $T(E_i) = i$ será o número de comparações para cada entrada E_i. Substituindo $Pr(E_i)$ e $T(E_i)$ na expressão da complexidade média tem-se:

$$T = T_M = \sum_{i=1}^{n+1} \Pr(E_i).t(E_i) = \sum_{i=1}^{n} \frac{q}{n}.i + (1-q).(n+1) = (n+1).\frac{(2-q)}{2}$$

Observe que se $q=1$ tem-se em média $(n+1)/2 \approx n/2$ comparações! •

A maior dificuldade existente na análise do comportamento médio de um algoritmo é definir a distribuição da entrada de dados associada ao problema. Freqüentemente, assume-se que as instâncias sejam igualmente prováveis, entretanto, a determinação de uma amostra representativa da entrada pode, em algumas casos, não ser muito precisa. Observe ainda que, no cálculo de cada $T(E_i)$, contabilizou-se apenas o número de comparações realizadas ($T(E_i) = i$). Como discutido a seguir, este tipo de abordagem irá simplificar enormemente o cálculo de complexidade. A complexidade local de tempo (ou espaço) será substituída pela complexidade assintótica.

I.6 - COMPLEXIDADE ASSINTÓTICA

A determinação de uma expressão exata para as complexidades locais de tempo ou espaço de um determinado algoritmo são freqüentemente desgastantes e desnecessárias. Em um modelo teórico de máquina, a complexidade local já é uma aproximação grosseira de uma máquina real, não sendo necessário portanto, um grande esforço analítico para obtê-la. Por que não observar apenas o comportamento assintótico das funções de complexidade quando o tamanho do problema se torna arbitrariamente grande? Neste caso, será suficiente provar se as complexidades obtidas tem um comportamento linear, quadrático, exponencial, etc.

Na determinação da complexidade assintótica faz-se, freqüentemente, uma simplificação das constantes que aparecem na expressão da complexidade local. Neste caso, pretende-se

determinar apenas a taxa de variação do tempo de processamento em relação ao tamanho do problema. Em outras palavras deseja-se obter a ordem de grandeza da função de complexidade local. Abaixo é apresentada uma definição mais formal de complexidade assintótica:

Definição I.1: (Complexidade Assintótica)

Uma função de complexidade local *T(n)* será de ordem *O(f(n))* se, e somente se, existirem constantes positivas *c* e n_0 tal que $0 \leq T(n) \leq c.f(n), \ \forall n \geq n_0$. •

Note que *c.f(n)* define um *limite superior* para *T(n)*. Graficamente, tem-se a seguinte situação representada na Figura I.8.

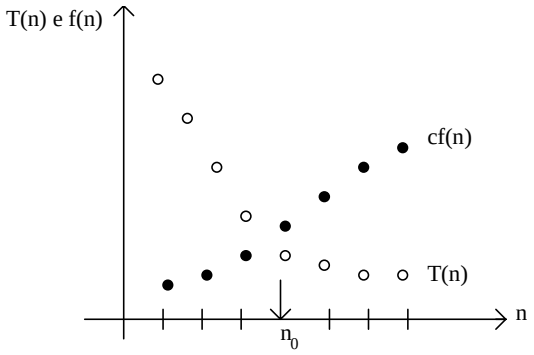

Figura I.8: Complexidade Assintótica

Pode-se representar por *O(f(n))*, o conjunto de todas as funções *T(n)* tais que $0 \leq T(n) \leq c.f(n), \ \forall n \geq n_0$.. Neste caso, pode-se dizer simplesmente que $T(n) \in O(f(n))$[1].

Portanto, para mostrar que uma dada função *T(n)* pertence a *O(f(n))* deve-se exibir ou garantir a existência de duas constantes *c* e n_0 tal que a definição descrita acima se verifique. Como determinar entretanto as constantes *c* e n_0? Para responder a essa pergunta considere inicialmente a seguinte definição formal de limite:

Definição I.2: (Limite):

Diz-se que $\lim_{n\to\infty} h(n) = k$ se, e somente se, $\forall \varepsilon > 0, \ \exists n_o \ tal \ que \ \|h(n) - k\| \leq \varepsilon, \ \forall n \geq n_0$. •

Assim, se $lim_{n\to\infty} T(n)/f(n) = k < \infty$ tem-se, da definição de limite, que: $\forall \varepsilon > 0, \exists n_0$ tal que $||T(n)/f(n) - k|| \leq \varepsilon, \ \forall n \geq n_0$. Ou seja, tem-se $k - \varepsilon \leq T(n)/f(n) \leq k + \varepsilon, \ \forall n \geq n_0$. Tomando esta última desigualdade e fazendo $c = k + \varepsilon$, conclui-se que: $\exists$ *c* e n_0 tal que $T(n) \leq c.f(n), \ \forall n \geq n_0$, ou seja *T(n) é O(f(n))* (ou simplesmente $T(n) \in O(f(n))$). Note que a existência de n_0 na definição de complexidade assintótica é garantida pela definição formal de limite. Este tipo de abordagem é interessante, já que ele nos exige, da talvez árdua tarefa de determinar n_0 explicitamente.

Considere o seguinte exemplo:

Exemplo I.6: Seja $T(n) = 5n^2 + n$. A função *T(n)* é $O(n^2)$? Note que fazendo *c = 6* tem-se $5n^2 + n \leq 6n^2$. Calculando $lim_{n\to\infty} T(n)/f(n)$:

$$\lim_{n\to\infty} \frac{5n^2 + n}{n^2} < 6 < \infty .$$

[1] Alguns autores utilizam a notação: $T(n) = O(f(n))$ (vide Cormem et al. [1990]).

Pode-se garantir portanto, da definição de limite, a existência de um inteiro n_0 tal que $5n^2 + n \leq 6n^2,\ \forall n \geq n_0$. •

Pode-se mostrar também que $T(n)= 5n^2+n$ é $O(n^3)$ ou $O(n^4)$. Entretanto, estaremos interessados no menor limite superior possível. Para se evitar conflitos dessa natureza, diz-se simplesmente que $T(n)$ é $o(n^3)$ ou $o(n^4)$. Mais formalmente se $T(n)$ é $o(f(n))$ então $lim_{n\to\infty} T(n)/f(n) = 0$.

Exemplo I.7: $T(n) = 3^n$ é $O(2^n)$? Suponha que existam k e n_0 tais que $3^n \leq k.2^n,\ \forall n \geq n_0$. Segue que:

$$\left(\frac{3}{2}\right)^n \leq k, \quad \forall\ n \geq n_0$$

Calculando o limite:

$$\lim_{n\to\infty}\left(\frac{3}{2}\right)^n \to +\infty \leq k \qquad \text{(absurdo!)}$$

Chegando portanto a uma contradição. Logo, $T(n)=3^n$ não será $O(2^n)$. Pode-se mostrar entretanto que 2^n é $O(3^n)$, ou mais precisamente, que 2^n é $o(3^n)$. Verifique! •

Pode-se estabelecer, através da notação O(.), uma hierarquia de funções de complexidade. Abaixo são apresentados alguns exemplos:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset \ldots \subset O(2^n) \subset O(n!) \subset \ldots$$

Da mesma forma, se $T(.)$ representa a complexidade de melhor caso de um algoritmo pode-se definir um *limite inferior* para $T(.)$. O limite inferior será representado por $\Omega(.)$.

Definição I.3: (Limite Inferior)

Seja $T(n)$ a complexidade local (de melhor caso) de um dado algoritmo. A função de complexidade $T(n)$ é $\Omega(f(n))$ se existem constantes c e n_0 *tais que* $T(n) \geq cf(n),\ \forall n \geq n_0$. •

Exemplo I.8: Seja $T(n) = 2n^2 + n$ a complexidade de melhor caso de um dado algoritmo. Deseja-se mostrar que $T(n)$ é $\Omega(n^2)$. De fato, bastará fazer k=1 e n_0=1. Assim: $2n^2 + n \geq n^2,\ \forall n \geq 1$. Pode-se mostrar também que $T(n)$ é $\Omega(n)$ ou $\Omega(1)$, entretanto, deve-se buscar normalmente, o maior limite inferior possível. Analogamente à notação o(.), pode-se utilizar ω(.) para $\Omega(.)$. Assim, em nosso exemplo, pode-se dizer que $T(n)$ é $\omega(n)$ ou $\omega(1)$. •

Considere agora a seguinte definição:

Definição I.4: (Limite):

Diz-se que $\lim_{n\to\infty} h(n) = +\infty$ se, e somente se, $\forall A > 0, \exists\ n_o\ tal que\ h(n) > A,\ \forall n \geq n_0$. •

Observe por exemplo que, se $lim_{n\to\infty} T(n)/f(n) = +\infty$ então $T(n)$ é $\omega(f(n))$. Verifique!

Em determinadas situações, um algoritmo de complexidade $T(n)$ será $O(f(n))$ e $\Omega(f(n))$ simultaneamente. Neste caso, $T(n)$ será $\theta(f(n))$. Mais formalmente, tem-se:

$$\theta(g(n)) = \{T(n) : \exists\, c_1, c_2, n_0 > 0 \;\; tais\ que \;\; 0 \le c_1 g(n) \le T(n) \le c_2 g(n), \;\; \forall \;\; n \ge n_0\}$$

Exemplo I.9:

a) Para encontrar o maior elemento em uma lista desordenada de *n* elementos pode-se construir facilmente um algoritmo de complexidade local *T(n)* e ordem *O(n)* e *Ω(n)* respectivamente. Neste caso, tem-se um algoritmo de ordem *θ(n)*. Um algoritmo com essa característica será chamado *algoritmo ótimo.*

b) Para encontrar um elemento *x* em uma lista desordenada de *n* elementos, tem-se obviamente *O(n)* passos no pior caso. No melhor caso entretanto, apenas uma comparação será realizada. Assim, *T(n)* terá limite inferior igual a *Ω(1)*. Neste caso o algoritmo não será ótimo.•

Suponha agora um algoritmo que se divida em duas ou mais partes independentes. A seguir, serão apresentadas algumas regras que nos ajudarão no cálculo de complexidade assintótica de um determinado algoritmo:

Proposição I.2 (Regra das Somas):

Um algoritmo *A* se divide em duas partes independentes A_1 e A_2 onde A_1 de complexidade é $T_1(n)$ é de ordem *O(f(n))* e A_2 de complexidade $T_2(n)$ é de ordem *O(g(n))* então $T(n) = T_1(n)+T_2(n)$ e A será de ordem *O(max(f(n),g(n)))*.

Prova: Sejam C_1, C_2, n_1, n_2 constantes positivas tais que:

$$T_1(n) \le C_1 . f(n), \qquad \forall\, n \ge n_1$$
$$T_2(n) \le C_2 . g(n), \qquad \forall\, n \ge n_2$$

fazendo-se $n_0 = max(n_1, n_2)$ e adicionando as duas equações acima tem-se:

$$T_1(n) + T_2(n) \le C_1 . f(n) + C_2 . g(n) \le (C_1 + C_2) . max(f(n), g(n)), \quad \forall\, n \ge n_0 .$$

Portanto, da definição tem-se que *T(n)* é de ordem *O(max(f(n),g(n)))* •

Proposição I.3 (Regra do Produto): Se um algoritmo A contém dois "aninhamentos" A_1 *e* A_2, de complexidade $T_1(n)$ *e* $T_2(n)$ com ordem *O(f(n))* e *O(g(n))* respectivamente, então *A* será de complexidade local $T(n) = T_1(n).T_2(n)$ e de ordem *O(f(n).g(n))*.

Prova: Sejam C_1, C_2, n_1, n_2 constantes positivas tais que:

$$T_1(n) \le C_1 . f(n), \qquad \forall\, n \ge n_1$$
$$T_2(n) \le C_2 . g(n), \qquad \forall\, n \ge n_2$$

fazendo-se $n_0 = max(n_1, n_2)$ e $C = C_1 . C_2$ conclui-se que:

$$T(n) = T_1(n).T_2(n) \le C.(f(n).g(n)), \quad \forall\ n \ge n_0 .$$

Portanto, *T(n)* será de ordem *O(f(n).g(n))*. •

Proposição I.4: (Propriedades Transitiva e Reflexiva)

Sejam $f, g\ e\ h: N \to \Re$ funções de complexidade:

a) Se *f(n)* é de ordem *O(g(n))* e *g(n)* é de ordem *O(h(n))* então *f(n)* é de ordem *O(h(n)).*
b) *g(n)* é de ordem *O(f(n))* e *f(n)* é de ordem *O(g(n))* se, e somente se, *O(f(n))=O(g(n)).*
c) *g(n)* é de ordem *Of((n))* e *f(n) não* é de ordem *O(g(n))* se, e somente se, *O(f(n))⊂O(g(n)).*

Prova:
a) Tem-se da definição de complexidade assintótica que:

$$f(n) \le C_1.g(n), \qquad \forall\, n \ge n_1$$
$$g(n) \le C_2.h(n), \qquad \forall\, n \ge n_2$$

Fazendo $n_0 = max(n_1, n_2)$ e $C = C_1.C_2$ *para* $n \ge n_0$ conclui-se que : $f(n) \le C_1.g(n) \le C.h(n)$ $\forall\, n \ge n_0$. Logo *f(n)* é de ordem *O(h(n)).*

b) (→) Considere uma função *T(n)* qualquer pertencente a *O(f(n)).* Como *f(n)* pertence a *O(g(n))* tem-se, da propriedade transitiva que *T(n)* pertence a O(g(n)). Assim, qualquer que seja *T(n)* em *O(f(n))* tem-se *T(n)* pertencente a *O(g(n)),* ou seja, *O(f(n))⊆O(g(n)).* De maneira análoga, pode-se mostrar que *O(f(n))⊇O(g(n)).* Segue então que *O(f(n))=O(g(n)).*
(←) É fácil mostrar que *f(n)* pertence a *O(f(n)).* Como *O(f(n))=O(g(n))* segue que *f(n)* pertence a *O(g(n)).* De maneira análoga pode-se mostrar que *g(n)* pertence a *O(f(n)).*

c) (→) Seja *T(n)* qualquer tal que *T(n) ∈ O(f(n)).* Como *f(n) ∈ O(g(n)),* tem-se do item (a) que *T(n) ∈ O(g(n))* (prop. transitiva). Logo *O(f(n))⊆O(g(n)).* Suponha por absurdo que, *O(f(n)) = O(g(n)).* Como *g(n) ∈ O(g(n))* segue então que *g(n) ∈ O(f(n))* (contradição!). Logo *O(f(n)) ⊂ O(g(n)).*
(←) Temos que *f(n) ∈ O(f(n)).* Como *O(f(n)) ⊂ O(g(n))* então *f(n)∈O(g(n)).* Se *g(n) ∈ O(f(n))* segue da proposição anterior (item b) que *O(g(n)) = O(f(n))* o que é absurdo! Portanto *g(n) ∉ O(f(n)).* •

Proposição I.5: Se $lim_{n\to\infty} f(n)/g(n) = k$ onde $0 < k < \infty$ então *O(f(n)) = O(g(n)).* Se $k = 0$ então *O(f(n)) ⊂ O(g(n)).*

Prova: Considere inicialmente o caso onde $k > 0$. Da definição de limite tem-se: $\forall\, \varepsilon > 0, \exists\, n_0$ tal que, $k-\varepsilon \le f(n)/g(n) \le k+\varepsilon, \forall\, n \ge n_0$.. Ao escolher $k > \varepsilon$ tem-se: *(a)* $f(n) < (k+\varepsilon).g(n), \forall\, n \ge n_0$. e *(b)* $g(n) < (1/(k-\varepsilon)).f(n), \forall\, n \ge n_0$ (verifique). Logo, de *(a)* e *(b)* respectivamente temos *f(n) ∈ O(g(n))* e *g(n) ∈ O(f(n)).* Da proposição anterior, segue que *O(f(n)) = O(g(n)).*

Suponha agora $k = 0$. Da definição de limite tem-se: $\forall\, \varepsilon > 0, \exists\, n_0$ tal que, $-\varepsilon \le f(n)/g(n) \le \varepsilon, \forall\, n \ge n_0$. Assim, da segunda parte da desigualdade: $\forall\, \varepsilon > 0, \exists\, n_0$ tal que, $f(n) \le \varepsilon.g(n), \forall\, n \ge n_0$, ou seja, fazendo c=ε, tem-se, da definição de complexidade assintótica que *f(n) ∈ O(g(n)).* Ainda, da segunda parte da desigualdade conclui-se que: $\forall\, \varepsilon > 0, \exists\, n_0$ tal que $g(n) > (1/\varepsilon).f(n), \forall\, n \ge n_0$. Da definição I.4, $lim_{n\to\infty} f(n)/g(n) = +\infty$. Provou-se portanto que *g(n) ∉ O(f(n)).* Como *f(n) ∈ O(g(n)),* tem-se da proposição anterior (item 3) que *O(f(n)) ⊂ O(g(n)).* •

DefiniçãoI.5: A definição de complexidade assintótica pode ser estendida facilmente para o caso onde o tamanho do problema seja representado por *dois* ou mais parâmetros. Dada uma função *g(n,m),* representa-se por *O(g(n,m))* o seguinte conjunto de funções:

$$O(g(n,m) = \{f(n,m) : existem\ constantes\ positivas\ c, n_0\ e\ m_0\ tais\ que\ 0 \le f(n,m) \le cg(n,m)$$
$$\forall\ n \ge n_0\ e\ m \ge m_0\}$$

A definições de $\Omega(g(n,m))$ e $\theta(g(n,m))$ são análogas. Este tipo de representação é bastante útil quando se trabalha com algoritmos em grafos. Neste caso, n representa o número de vértices e m é o número de arestas.

Quando se trabalha com outros modelos de máquina (diferentes da máquina RAM), pode ocorrer que a expressão de complexidade assintótica obtida nestes modelos seja distinta. Ao mudar de uma máquina para outra é importante que se preserve algumas características do modelo original. Considere então a seguinte definição:

Definição I.6: (Funções relacionadas polinomialmente)

Duas funções $f_1: N \to \Re$ e $f_2: N \to \Re$ são *relacionadas polinomialmente* se existirem funções $p_1: \Re \to \Re$ e $p_2: \Re \to \Re$ tais que $f_1(n) \le p_1(f_2(n))$ e $f_2(n) \le p_2(f_1(n))\ \forall n \ge 0$. •

Como exemplo, considere *duas* funções $f_1(n) \le 2n^2$ e $f_2(n)$. Fazendo $p_1(x) = 2x$ na primeira desigualdade tem-se que $2n^2 \le p_1(n^5) = 2n^5$, $\forall n \ge 0$. Na segunda desigualdade, fazendo $p_2(x) = x^5$ chega-se a $n^5 \le p_2(2n^2) = 8n^6$, $\forall n \ge 0$.

Ao se trabalhar com uma máquina de Turing, máquina RAM ou máquina RASP (Aho et. al.[1974]) tem-se funções de complexidade relacionadas polinomialmente. Isto significa dizer que, se uma função de complexidade é polinomial (ou exponencial) em um dado modelo ela será também polinomial (ou exponencial) nos demais modelos. Maiores detalhes sobre este assunto podem ser encontrados em Aho et. al.[1974].

I.7 - COMPLEXIDADE DE ALGORITMOS RECURSIVOS

Um procedimento recursivo consiste, basicamente, na utilização direta ou indireta, de um procedimento dentro do mesmo procedimento que o define. Normalmente, todo processo recursivo exigirá a definição de uma fórmula de recorrência e de uma base da recursão. Esta técnica é amplamente utilizada em métodos do tipo divisão e conquista (para maiores detalhes vide Cormen et al. [1990]).

O exemplo seguinte ilustra uma aplicação da recursividade ao problema de ordenação.

I.7.1 - Algoritmo Quicksort:

Suponha que se queira ordenar uma lista A qualquer com n valores inteiros positivos. Baseado na estratégia de divisão e conquista, Hoare[1962], propôs um algoritmo de ordenação diferente do *Mergesort*. No *Mergesort* dividi-se a lista original A em partes iguais A_1 e A_2 respectivamente. Faz-se então uma ordenação de cada uma das partes seguida de uma concatenação das sub-listas. No *Quicksort* a divisão em duas sub-listas é feita utilizando-se um elemento *chave* (ou pivô) de forma que, no final do processo, as sub-listas ordenadas não precisem mais ser concatenadas! Na sub-lista A_1 são armazenados todos os elementos menores que *chave* enquanto que, na sub-lista A_2 apenas os elementos maiores que *chave*. Após a divisão, repeti-se o processo para A_1 e A_2 separadamente.

O exemplo seguinte ilustra as principais etapas presentes no *Quicksort*. Considere uma lista desordenada A como no exemplo seguinte:

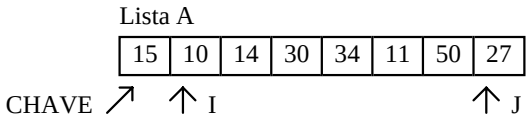

Figura I.9: Lista A desordenada

Note que são criados dois índices *I* e *J* apontando para o segundo e o último elementos respectivamente. Além disso, separa-se o primeiro elemento *A[1]=15* (denominado elemento *chave* ou *pivô*). O objetivo inicial será determinar a posição correta dessa *chave* na lista *A*. Assim, incrementa-se *I* sempre que algum elemento *A[I]*, menor do que *chave* for encontrado. Analogamente, decrementa-se *J* sempre que um elemento *A[J]* maior do que *chave* for encontrado. Essa situação pode ser expressa como na figura seguinte:

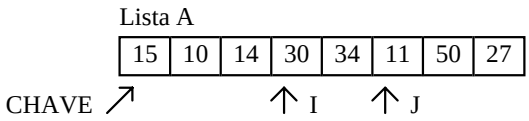

Figura I.10: Atualizações de I e J

Caso se tenha $I < J$ os elementos *A[I]* e *A[J]* são trocados de posição e o processo repetido até que $I \geq J$. A nova situação é apresentada abaixo:

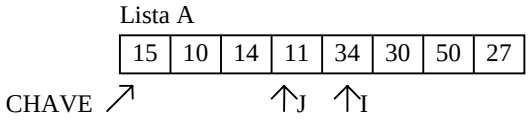

Figura I.11: I > J

Note que a posição correta do elemento *chave* é indicada pelo índice *J*! O próximo passo será trocar a posição do elemento *chave* com *A[J]*. Assim:

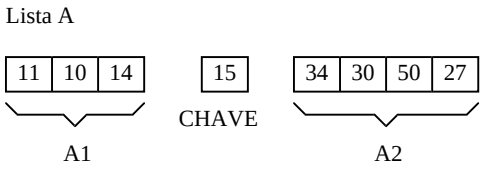

Figura I.12: Divisão da Lista A

Bastará aplicar agora o mesmo processo para as sub-listas A_1 e A_2 separadamente através de chamadas recursivas. Finalmente, tem-se a seguinte situação:

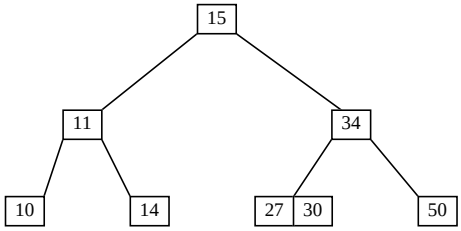

Figura I.13: Lista A ordenada

Observe que as chamadas recursivas são executadas até que *0, 1* ou *2* elementos sobre em cada sub-lista (base da recursão). Caso se tenha apenas *um*, ou nenhum elemento ao final do processo, nenhuma operação será realizada. Entretanto, se a *A* contiver *2* elementos, apenas uma comparação será realizada. Resumindo, tem-se o seguinte procedimento:

```
Procedimento I.4: Quicksort (inf, sup);
Início
        Se (sup - inf < 2) então
                Se (sup - inf = 1) então                {temos 2 elementos}
                        Se (A[inf]>A[sup]) então
                                troca (A[inf], A[sup]);
        senão
                I ← inf;
                J ← sup;
                chave ← A[inf];
                Enquanto J > I faça
                        I ← I + 1;
                        Enquanto (A[I] < chave) faça
                                I ← I + 1;
                        Enquanto (A[J] > chave) faça
                                J ← J - 1;
                        Se (J > I) então
                                troca(A[I], A[J]);
                fim Enq;
                troca (A[inf],A[J]);
                Quicksort (inf, I-1);
                Quicksort (J+1, sup);
        fim;
fim.
```

Figura I.14: Algoritmo *Quicksort*

Note no procedimento *Quicksort*, que se o elemento *chave* for o maior de todos os elementos de *A*, o índice *I* será incrementado até *n+1*. Para evitar um erro decorrente de um acesso indevido a esta posição (inexistente na lista *A*), bastará criar um novo elemento fazendo $A(n+1)=\infty$ no programa principal. Verifique!

I.7.1.1 - Complexidade do Quicksort (Pior Caso)

Observando a algoritmo acima pode-se eleger (sem perda de generalidade) a instrução *A[inf]>A[sup]* como operação elementar na base da recursão. Note que *3* situações distintas serão possíveis. Se *sup-inf < 0,* a chamada recursiva irá corresponder a uma sub-lista vazia. Neste caso a operação elementar não será realizada. O mesmo ocorre se *sup-inf =0* (sub-lista com apenas *1* elemento). Segue então que *T(0)=0* e *T(1)=0* respectivamente. Se *sup-inf =1* tem-se uma sub-lista com 2 elementos. Neste caso, *T(2)=1*.

No cálculo da fórmula de recorrência ($n \geq 3$) deve-se atualizar os índices *I* e *J* até que se tenha $I \geq J$. Só então, as chamadas recursivas serão realizadas (etapa de divisão). Observe novamente que *3* etapas devem ser analisadas.

Na etapa de atualização dos índices *I* e *J* conclui-se facilmente que *O(cn)* passos serão executados. Os índices *I* e *J* deverão percorrer a lista de tamanho *n* até se cruzarem.

Nas chamadas recursivas, deve-se estabelecer inicialmente como o vetor original *A* será dividido. Essa divisão entretanto dependerá da natureza dos dados de entrada. Na pior divisão possível, pode-se construir uma configuração onde *n-1* elementos estarão presentes em uma das sub-listas e 0 elementos na outra. Neste caso, *T(n)* se divide em *T(n-1)* e *T(0)* respectivamente. Isto ocorrerá sempre que o vetor original *A* já estiver ordenado! É fácil ver que esta é a pior divisão possível já que, na chamada de complexidade *T(n-1)*, o tamanho do problema original é reduzido em apenas uma unidade! Assim, um maior número de chamadas deverão ser executadas até que se chegue à base da recursão. O mesmo não ocorreria se *A* fosse dividida em duas partes de mesmo tamanho (Verifique!).

Tem-se portanto a seguinte situação:

$$\begin{cases} T(0) = 0,\ T(1) = 0 \ e\ T(2) = 1, & n \le 2 \quad (\textit{base da recursão}) \\ T(n) = cn + T(0) + T(n-1), & n > 2 \quad (\textit{fórmula de recorrênci a}) \end{cases}$$

Desenvolvendo a recorrência acima conclui-se que:

$$\begin{aligned} T(n) &= cn + T(n\text{-}1) \\ &= cn + c(n\text{-}1) + T(n\text{-}2) \\ &= cn + c(n\text{-}1) + c(n\text{-}2) + T(n\text{-}3) \end{aligned}$$

Após *k* passos tem-se: $T(n) = cn + c(n\text{-}1) + c(n\text{-}(k\text{-}1)) + T(n\text{-}k).$ (I)

Espera-se obter, ao final do processo *n-k=2* (base da recursão). Assim, substituindo $n-k=2$ em (I) tem-se:

$$\begin{aligned} T(n) &= c(n + (n\text{-}1) + (n\text{-}2) + \ldots + 4 + 3) + T(2) \\ &= c\sum_{i=3}^{n} i + T(2) = c\left(\left(\sum_{i=1}^{n} i\right) - 3\right) + 1 = c\left(\frac{n(n+1)}{2} - 3\right) + 1 \end{aligned}$$

Segue portanto que *T(n)* é de ordem $O(n^2)$. A seção seguinte trata da análise de complexidade média do algoritmo *Quicksort*.

I.7.1.2 - Complexidade do Quicksort (Caso Médio):

Apesar da complexidade teórica de pior caso do *Quicksort* ter se mostrado inferior ao *Mergesort*, testes realizados empiricamente mostram um bom desempenho do *Quicksort* quando comparado ao *Mergesort* (vide Knuth[1973]). Pode-se mostrar em média (para uma entrada gerada aleatóriamente) que o *Quicksort* (assim como o *Mergesort*) é processado em *O(nlogn)* passos.

Assim, suponha inicialmente que a probabilidade do elemento *chave* ser o *i-ésimo* elemento seja *1/n* (distribuição uniforme). Para calcular a análise de performance média deve-se analisar todas as chamadas recursivas possíveis, ou seja, todas as possíveis divisões do vetor original *A*. Utilizando a média ponderada conclui-se que:

$$\frac{1}{n}\sum_{k=1}^{n}\left(T_M(k-1) + T_M(n-k)\right)$$

representa o compartamento médio de todas chamadas recursivas (lembre-se que $T_M(.)$ representa o tempo médio para se processar uma entrada de tamanho n). Resumindo todos os casos, tem-se:

$$\begin{cases} T_M(0)=0,\ T_M(1)=0 \ e\ T_M(2)=1, & n \le 2 \quad (base\ \ da\ \ recursão) \\ T_M(n)=cn+\frac{1}{n}\sum_{k=1}^{n}\left(T_M(k-1)+T_M(n-k)\right), & n>2 \quad (fórmula\ \ de\ recorrênci\ a) \end{cases}$$

Desenvolvendo a fórmula de recorrência tem-se que:

$$\begin{aligned} T_M(n) &= cn+\frac{1}{n}.\sum_{k=1}^{n}\left(T_M(k-1)+T_M(n-k)\right) \\ &= cn+\frac{2}{n}.\left(T_M(0)+T_M(1)+..+T_M(n-1)\right) \end{aligned}$$

Multiplicando por n ambos os lados tem-se:

$$nT_M(n)=cn^2+2\sum_{i=0}^{n-1}T_M(i) \qquad (I)$$

Trocando n por $n-1$ em (I):

$$(n-1)T_M(n-1)=c(n-1)^2+2\sum_{i=0}^{n-2}T_M(i) \qquad (II)$$

Subtraindo (II) de (I) segue:

$$\begin{aligned} nT_M(n)-(n-1)T_M(n-1) &= c(n^2-(n-1)^2)+2T_M(n-1) \\ &= c(2n-1)+2T_M(n-1) \end{aligned}$$

Fazendo as devidas simplificações na expressão acima chega-se a:

$$nT_M(n)=(n+1)T_M(n-1)+c(2n-1)$$

Multiplicando ambos os lados por $\frac{1}{n(n+1)}$:

$$\frac{T_M(n)}{(n+1)}=\frac{T_M(n-1)}{n}+c.\frac{(2n-1)}{n(n+1)}<\frac{T_M(n-1)}{n}+\frac{2c}{n+1}$$

Assim:

$$\begin{aligned} \frac{T_M(n)}{n+1} &< \frac{2c}{n+1}+\frac{T_M(n-2)}{n-1}+\frac{2c}{n} \\ &< \frac{2c}{n+1}+\frac{2c}{n}+\frac{2c}{n-1}+\frac{T_M(n-3)}{n-2} \\ &< \frac{2c}{n+1}+\frac{2c}{n}+\frac{2c}{n-1}+\ldots+\frac{2c}{n-k+2}+\frac{T_M(n-k)}{n-k+1} \end{aligned}$$

Espera-se obter $n-k=2$ (base da recursão). Como $T_M(2)=1$ tem-se:

$$\frac{T_M(n)}{n+1}<2c\sum_{j=4}^{n+1}\frac{1}{j}+\frac{T_M(2)}{3}$$

Note que o somatório acima pode ser majorado pela integral de *1/j* (vide Figura I.15). Assim:

$$\frac{T_M(n)}{n+1} < 2c\int_3^{n+1}\frac{1}{j}.dj + \frac{1}{3} = 2c(\ln(n+1) - \ln 3) + \frac{1}{3}$$
$$< 2c.\ln(n+1) - 2c.\ln 3 + \frac{1}{3}$$

Fazendo-se as devidas simplificações chega-se a:

$$T_M(n) < 2cn\ln((n+1)/3) + d$$

onde **c** e **d** são constantes.

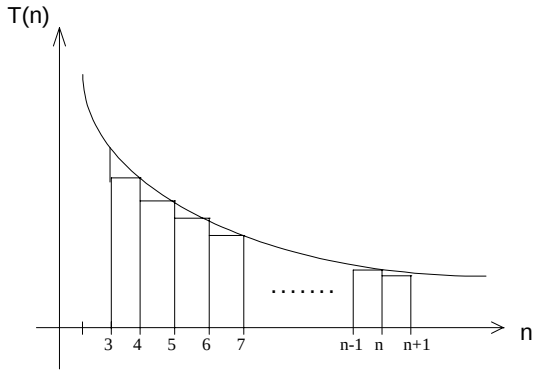

Figura I.15: Integral de 1/j

Observe que, fazendo a mudança de base e simplificando novamente a expressão conclui-se finalmente que $T_M(n)$ é *O(nlog n).* •

I.8 – LIMITE INFERIOR DE PROBLEMAS

Em algumas situações, deseja-se determinar qual a complexidade inerente a um determinado problema (limite inferior). Sua complexidade deverá ser independente dos algoritmos existentes ou não em determinado momento. A obtenção do limite inferior entretanto, pode ser bastante difícil para uma grande quantidade de problemas. O que se faz nestes casos é iniciar a análise do problema com um limite inferior qualquer e ir aumentando (se possível) esse limite até que não seja mais possível aumentá-lo. Teremos chegado, neste caso, a um limite inferior desejado para o problema. Mais formalmente tem-se a seguinte definição:

Definição I.7: (Limite Inferior de um Problema)

Diz-se que *Ω(f(n))* define um limite inferior de um problema π quando qualquer algoritmo que resolver π requerer, pelo menos, *O(f(n))* passos no pior caso. •

Exemplo I.10: Se $\Omega(n^2)$ é um limite inferior de um problema π não poderá existir nenhum algoritmo com complexidade assintótica de pior caso inferior a $O(n^2)$. •

Note neste caso que o limite $\Omega(f(n))$ é o melhor (maior) possível. Pode-se dizer ainda que um algoritmo ótimo tem menor a complexidade (de pior caso) entre todos os algoritmos possíveis na resolução de um dado problema.

Como comentado anteriormente, o algoritmo *Quicksort* (para uma grande quantidade de entradas) possui desempenho computacional superior quando comparado ao *Mergesort* (sendo ambos $\Omega(nlogn)$). Pode-se mostrar na verdade, que $\Omega(nlogn)$ é o limite inferior para o problema de ordenação (para maiores detalhes vide Horowitz&Sahni[1978]).

I.9 – PROBLEMAS DE DECISÃO E ALGORITMOS NÃO-DETERMINÍSTICOS

Como classificar os problemas resolvidos ou não pelos algoritmos e ferramentas disponíveis em dado momento? Uma primeira classificação diz respeito aos problemas que podem ou não ser resolvidos algoritmicamente (problemas computáveis e não-computáveis). Entretanto, considerando-se apenas os problemas computáveis, como classifica-los em função do grau de dificuldade presente em sua resolução? Note por exemplo que se não resolvemos um problema em tempo polinomial (no tamanho de sua entrada), isto não significa que se possa faze-lo posteriormente! Informalmente falando, como classificar tais problemas independentemente dos algoritmos existentes ou não em dado momento? Esta pergunta começou a ser respondida nos primórdios de 1970 quando uma nova área teve origem na ciência da computação. A teoria de complexidade computacional, buscava, a partir de então, categorizar e classificar problemas computáveis e assim melhor avaliar a qualidade dos algoritmos atualmente disponíveis. Problemas com limite inferior exponencial por exemplo podem ser classificados como *intratáveis*, já que, comprovadamente, será impossível a obtenção de algoritmos (nos modelos de máquina tradicionais) que nos dêem uma solução exata em tempo polinomial. Problemas que possam ser resolvidos por algoritmos polinomiais no tamanho de sua entrada serão chamados de "tratáveis" e pertencerão à classe *P* dos problemas computáveis (Garey&Jonhson[1979]).

Apesar do incansável esforço de uma enorme quantidade de pesquisadores nas mais diversas áreas de atuação, alguns problemas computáveis permanecem sem uma resolução satisfatória. Décadas de trabalho envolvendo abordagens diferenciadas não demonstraram ser suficientes na resolução de determinados problemas em um tempo de processamento aceitável (tempo polinomial). Esse grande número tentativas frustradas levaram a maioria dos pesquisadores a acreditar que tais problemas são inerentemente difíceis e que não poderão ser resolvidos eficientemente. A teoria da NP-completude (Garey&Jonhson[1979]) resolve em parte tais problemas. Apesar dela não provar se tais problemas podem ou não ser resolvidos em tempo polinomial, ela garante que problemas NP-completo se equivalem quanto a seu grau de dificuldade, no sentido que, se um problema desta classe puder ser resolvido em tempo polinomial todos os demais problemas também serão! Esse resultado surpreendente garante que cada um dos problemas desta classe será tão difícil quanto os demais. Neste sentido pode-se dizer que tais problemas são computacionalmente equivalentes. Fica a dúvida entretanto, se existem ou não algoritmos polinomiais para esses problemas.

Apesar dos aspectos negativos levantados acima, a teoria da NP-completude tem aspectos positivos que merecem ser ressaltados. O fato de sabermos a *priori* que um determinado problema é NP-Completo poderá obviamente refletir favoravelmente na estratégia de resolução a ser adotada. Aqueles problemas que, devido sua natureza, exijam um pequeno tempo de resposta poderão ser implementados com a utilização de heurísticas especialmente adaptadas (de tempo polinomial) e cuja solução obtida seja a mais próxima possível de uma solução ótima do problema original. Imagine por exemplo um sistema que gerencie o tráfego de veículos em uma cidade, e que um operador deseje simular, em tempo real, situações distintas que o auxiliem na tomada de decisões. Problemas de planejamento estratégico em uma companhia que envolvam altos custos financeiros

de implantação, poderão ser resolvidos através de ferramentas mais robustas e de complexidade exponencial no tamanho do problema desde o tempo total de processamento esteja dentro de um horizonte aceitável pela empresa. Neste caso, o elevado tempo de processamento necessário para resolução do problema poderá ser perfeitamente aceitável em detrimento de uma maior qualidade na resposta apresentada. A escolha de uma estratégia de resolução, dependerá portanto, de uma combinação de fatores que incluem desde uma avaliação prévia do grau de dificuldade intrínseco ao problema até a elaboração de alternativas de resolução oriundas das características do problema e da qualidade desejada pela solução.

Com o objetivo de melhor classificar problemas computáveis independentemente dos algoritmos disponíveis em dado momento, utiliza-se o conceito de *algoritmo não-determinístico.* Intuitivamente falando, em um algoritmo não-determinístico, a solução do problema pode ser apresentada através de um "oráculo" (simulado através de um paralelismo ilimitado). O que se faz neste caso, será apenas verificar a natureza da solução apresentada. Este tipo de abordagem nos exime da talvez frustrante, apresentação de um algoritmo determinístico polinomial para o problema. Como observado anteriormente, o fato de não conseguirmos resolver um problema através de um algoritmo determinístico polinomial não nos dá um atestado da não existência de tal algoritmo (para maiores detalhes vide Cormem et. al. [1990], Garey&Jonhson[1979]).

I.9.1 - PROBLEMAS DE DECISÃO

Uma grande quantidade de problemas algorítmicos podem ser classificados, basicamente, em 3 categorias distintas a saber: *decisão, localização* e *otimização*. Esta classificação se refere, especialmente, a determinadas características observadas na solução do problema.

Nos *problemas de decisão*, deseja-se responder apenas *sim* ou *não* a uma determinada pergunta. Como exemplo, suponha que se deseje saber se existe ou não, em um grafo G qualquer, um circuito hamiltoniano com custo inferior a *k* (para *k* inteiro positivo). Já nos *problemas de localização*, deseja-se determinar uma certa estrutura safisfazendo as restrições do problema. Assim, no problema do circuito hamiltoniano, deve-se exibir explicitamente (se existir) um circuito hamiltoniano com custo inferior a *k*. Nos *problemas de otimização* espera-se encontrar uma determinada estrutura (solução viável) de forma a minimizar ou maximizar uma função objetivo associada. No caso do problema do caixeiro viajante, deve-se determinar, dentre todos os um circuitos hamiltonianos, aquele que tiver menor custo.

Observe que ao resolver um problema de otimização, resolve-se implicitamente os problemas de localização associados. Ou seja, estaremos localizando possíveis estruturas (soluções viáveis) que minimizem uma função objetivo. Obviamente neste caso, a resposta ao problema de decisão também será afirmativa. Pode-se afirmar portanto que um problema de otimização será pelo menos tão difícil quanto um problema de localização ou decisão. O surpreendente neste caso é que a resolução de um problema de decisão também será tão difícil quanto os problemas de localização ou otimização associados! Em outras palavras, pode-se dizer que um problema de decisão terá o mesmo grau de dificuldade que um problema de localização ou otimização! Dessa forma, a análise do grau de dificuldade inerente a cada problema poderá ser simplificada observando-se apenas os problemas de decisão! Assim, se um determinado problema de decisão não puder ser resolvido em tempo polinomial, o problema de otimização associado também não poderá ser resolvido em tempo polinomial e vice-versa.

O problema do Caixeiro Viajante (apresentado abaixo) ilustra bem esta interessante equivalência entre os problemas de decisão e otimização.

I - Caixeiro Viajante (Otimização):

Considere um grafo direcionado *G=(V,A)* onde para cada arco *(i,j)* $\in$ *A* tem-se um custo de transporte c_{ij} associado. Encontre um circuito hamiltoniano *W* t. q. o custo total de transporte *z** seja o mínimo possível.

II - Caixeiro Viajante (Decisão):

Considere um grafo direcionado G=(V,A) onde para cada arco *(i,j)* $\in$ *A* temos um custo de transporte c_{ij} associado. Além disso seja *k**, um inteiro positivo. Existe em *G* algum circuito hamiltoniano *W* tal que o custo total de transporte seja inferior ou igual a *k**?

Observe que se for encontrado um algoritmo polinomial para o problema do caixeiro viajante (apresentado em *(I)*) resolve-se também, em tempo polinomial, o problema de decisão associado. Neste caso, bastará comparar o valor da solução ótima z* do problema de otimização com o valor inteiro *k**. Logo, a resolução de (*I)* implica diretamente na resolução de (II). Pode-se mostrar facilmente que a recíproca também é verdadeira, ou seja, caso se tenha um algoritmo polinomial para o problema de decisão, será possível utilizá-lo na construção de um novo algoritmo, também polinomial para o problema de otimização. Para isto, determina-se inicialmente uma cota superior para o valor da solução ótima z* no problema do caixeiro viajante (otimização). É fácil ver que, se C, representa o custo da maior aresta em do grafo então *nC* será um limite superior para *z**. A idéia agora será fazer uma busca binária pelo elemento z* no intervalo *[0,nC]*. Primeiramente, resolve-se o problema de decisão fazendo *k*= nC/2*. Caso a resposta a esse problema de decisão seja *sim*, continua-se a busca de z* restrita ao intervalo *[0, nC/2],* caso contrário, busca-se *z** no intervalo *[nC/2, nC]*. O processo é repetido até que se tenha um intervalo contendo apenas o elemento *z** (valor da solução ótima no problema de otimização). Observe que a complexidade total do problema de otimização será de ordem *O(f(n).log(nC))* onde *f(n)* representa a complexidade do problema de decisão associado.

I.9.2 - ALGORITMOS NÃO-DETERMINÍSTICOS

Em um algoritmo não-determinístico (para problemas de decisão) pode-se encontrar, além dos comandos usuais de repetição, decisão, atribuição , as seguintes operações:

- *Escolha*(S) $\rightarrow$ Escolhe um elemento do conjunto S
- *FRACASSO* $\rightarrow$ Sinaliza término com fracasso
- *SUCESSO* $\rightarrow$ Sinaliza término com sucesso

A função X $\leftarrow$ *Escolha(S),* armazena em X qualquer um dos elementos presentes em S. Não existe uma regra que especifique como tal escolha é realizada. As funções *SUCESSO* e *FRACASSO* sinalizam o final de processamento. Sempre que existir um conjunto de escolhas que nos leve a um término com *SUCESSO*, ela será realizada. Falando informalmente, essa função pode ser encarada como um "oráculo" que sempre visualiza a melhor opção. Um algoritmo não-determinístico termina em *FRACASSO* se, e somente se, não existir nenhum conjunto de escolhas que nos levem a um término com *SUCESSO*. O tempo de processamento para as funções: *Escolha(S), SUCESSO* e *FRACASSO* será de *O(1)* passos.

Uma interpretação determinística de um algoritmo não-determinístico pode ser feita imaginando-se um paralelismo ilimitado. Cada vez que a função *Escolha(S)* for executada, o algoritmo será capaz de gerar cópias de si próprio, cada uma delas correspondendo a uma possível escolha. Assim, tem-se a execução de várias cópias simultâneamente. A primeira cópia que obtiver *SUCESSO* (caso isto venha a ocorrer) cessa imediatamente o processamento de todas as outras

cópias e retorna *sim* ao problema de decisão associado. Uma cópia que retorne *FRACASSO* interrompe o processamento apenas daquela cópia em funcionamento. Neste caso, a resposta ao problema de decisão associado será *não*, apenas se todos os demais processadores finalizarem com *FRACASSO*. Pode-se afirmar que, mesmo que a probabilidade de encontrar uma solução correta seja extremamente pequena (mas positiva), ela será determinada se o número de chutes ou tentativas tende a infinito. Como exemplo, considere um jogo de azar ou loteria. A probabilidade de uma pessoa em particular acertar toda uma sequência correta de valores e ganhar o tão almejado prêmio é sem dúvida bastante pequena. Apesar disso, a probabilidade de se encontrar um ganhador aumenta consideravelmente se milhões de apostadores jogarem simultâneamente! Embora estranho, este tipo de abordagem nos exime da apresentação imediata de um algoritmo determinísto para o problema considerado. O que se faz, é apenas reconhecer a solução apresentada. Informalmente falando, ao se imaginar um paralelismo ilimitado, simula-se, de certo modo, um "oráculo" ou "bola de cristal" que sempre nos retorna uma resposta correta! Desta forma, se a resposta ao problema de decisão considerado for *sim*, um dos infinitos processadores deverá ter finalizado com *SUCESSO!*

Mais adiante no Capítulo III, veremos que a implementação de um algoritmo não-determinístico poderá ser "realizada" através de um algoritmo probabilístico (Método de Las Vegas). Neste caso entretanto, o tempo de processamento poderá tender a infinito. Informalmente falando, um único processador irá cumprir o papel dos infinitos processadores no algoritmo determinístico. Esta abordagem será interessante na prática quando o tempo esperado de processamento for polinomial no tamanho do problema.

Na verdade, máquinas não-determinísticas, como discutidas acima, não são interessantes do ponto de vista prático (de implementação). Elas nos ajudarão apenas na compreensão e classificação dos problemas algorítmicos. Assim, quando determinado problema não puder ser resolvido "eficientemente" através dos mecanismos conhecidos, utiliza-se as funções não-determinísticas detendo-se apenas na natureza da solução apresentada. Esta etapa do algoritmo será chamada *reconhecimento* da solução.

Os algoritmos não-determinísticos são fundamentais na definição da classe *NP* dos problemas de decisão. Para verificar se um dado problema de decisão π, pertence à classe *NP*, duas etapas deverão ser executadas: *exibição* e *reconhecimento*. Na etapa de *exibição*, uma justificativa à resposta *sim* é apresentada. O *reconhecimento* desta solução deverá ser realizada em tempo determinístico polinomial no tamanho da entrada. Vale ressaltar que, se o problema não tem reconhecimento polinomial ele poderá ou não pertencer a *NP*.

Observe que um algoritmo determinístico pode ser visto como um caso particular de um algoritmo não-determinístico, em outras palavras $P \subseteq NP$. Será entretanto que *P* está contido propriamente em *NP*? Ou seja, será possível afirmar que $P \neq NP$? Na verdade, se uma subclasse de *NP*, denominada *NP-Completo*, admitir um único problema que possa ser resolvido por um algoritmo polinomial então todos os problemas em *NP* admitirão também um algoritmo polinomial em sua resolução. Neste caso, tem-se $P=NP$! Existe uma forte tendência em se acreditar que não existem algoritmos polinomiais para problemas NP-Completos. Apesar disso, ainda não foi obtida nenhuma prova conclusiva e o problema continua em aberto. Maiores detalhes sobre este assunto podem ser encontrados em Garey&Jonhson[1979].

O exemplo seguinte mostra que o problema do Caixeiro Viajante (decisão) pertence a NP. Toda vez que um algoritmo não determinísticos encontrar *SUCESSO* ou *FRACASSO* durante o processamento ele para imediatamente.

Exemplo I.12: (Ciclo Hamiltoniano)

Suponha $G(V,E)$ um grafo completo não-orientado (onde $|V|=n$ e $|E|=m$) e k um inteiro positivo. Deseja-se determinar se G admite ou não um ciclo hamiltoniano de tamanho menor ou iqual a k Suponha que os custos entre cada para de vértices i,j sejam armazenadas em uma matriz C quadrada nxn. Caso não exista a conexão entre os vértices i e j o custo associado será *infinito*. O

ciclo hamiltoniano definindo a sequência de visitas realizadas será armazenado em um vetor S de *n* elementos. No algoritmo não-determinístico seguinte são apresentadas as etapas de exibição e reconhecimento. Suponha que o vértice *1* do grafo seja o vértice de partida e chegada:

Procedimento I.5: Ciclo Hamiltoniano não-determinístico – Grafo Completo;
Início
S ← ∅;
S[1] ← *1;*
Para *i:=2* **até** *n* **faça** {exibição}
S[i]← *Escolha{2,..,n};*
Para *i:=1* **até** *n-1* **faça** {começa reconhecimento}
para *j:=i+1* **até** *n* **faça**
se *S[i]=S[j]* **então** *FRACASSO*;
fim para;
custo_total ← *0;*
Para *i:=1* **até** *n-1* **faça**
custo_total ← *custo_total + C[S[i],S[i+1]];*
custo_total ← *custo_total + C[S[n],S[1]];*
Se *custo_total > k* **então** *FRACASSO;*
SUCESSO;
fim.

Figura I.16: Circuito Hamiltoniano Não-determinístico

A função *Escolha(.)* acima, retorna passo a passo os véstices presentes no ciclo hamiltoniano. Informalmente falando, tem-se infinitos processadores executando uma cópia deste procedimento. O primeiro processador a resolver o problema, passa essa informação a todos os demais. Observe que a etapa de reconhecimento é realizada por um algoritmo polinomial de ordem $O(n^2)$. Logo, o problema do ciclo hamiltoniano pertencerá à classe *NP* dos problemas de decisão.