Complexidade de Algoritmos

Roberto S. Bigonha
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte – MG

2006

ABSTRACT

This paper presents the main and most popular techniques for algorithm complexity analysis and associated methodologies for their application to typical case-studies of a course on Algorithms and Data Structures.

RESUMO

Este artigo apresenta as principais técnicas mais populares de análise de complexidade de algoritmos e metodologias para suas aplicações em estudos de casos típicos de uma disciplina de Algoritmos e Estruturas de Dados.

Palavras-chave: função de custo, custo médio, custo no pior caso, complexidade de algoritmos

1 Análise de Complexidade

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*[1]
>
> Antoine de Saint-Exupéry (1900-1944)

A determinação dos custos dos algoritmos é indispensável quando se deseja compará-los conforme seus desempenhos, de forma a escolher corretamente a melhor solução.

Pode-se determinar o tempo de execução de um algoritmo medindo-se, por meio de um relógio, o tempo que ele gasta para vários conjuntos de dados de entrada em um dado computador.

Esse método, entretanto, apresenta vários pontos negativos, pois, em primeiro lugar, o algoritmo precisa ser implementado, e, ainda assim, pode não ser possível realizar as medidas de tempo para todas as suas possíveis entradas. Execuções experimentais só podem ser feitas para um número limitado de entradas, quando, para ser conclusivo, dever-se-ia considerar todas as entradas possíveis, inclusive as *intratáveis*.

Há também o fato de as medidas obtidas pela medição direta do tempo de execução serem dependentes do compilador usado para implementar o algoritmo, que pode gerar códigos distintos para construções equivalentes, e também por que os tempos medidos dependem do desempenho do *hardware* no qual o algoritmo é executado. Além disso, análise de algoritmos deveria ser realizada sem necessidade de implementá-los.

Isso sugere a necessidade de um método de análise que seja independente desses fatores, por exemplo, um que use modelos ma-

[1] Parece que a perfeição é obtida não quando nada há a acrescentar, mas quando nada pode ser removido.

temáticos apropriados, determinados a partir da estrutura dos algoritmos, sem necessidade de sua implementação nem de execuções experimentais.

2 Medidas via Modelo Matemático

O desempenho de muitos algoritmos depende do tamanho do problema a ser resolvido. Problema pequenos, como ordenar um arranjo contendo poucos números inteiros, são resolvidos com rapidez, por pior que seja o método usado. Por outro lado, a ordenação de um milhão de inteiros somente é eficiente com o uso de soluções apropriadas.

O estudo da complexidade e do custo de algoritmos está diretamente vinculado ao tamanho do problema a ser resolvido, que deve ser devidamente identificado em cada caso. Algumas vezes, o tamanho do problema, que é um inteiro, coincide com o tamanho dos dados de entrada. Outras vezes, o tamanho tem a ver com o esforço necessário para se chegar ao resultado desejado. Por exemplo, no caso da ordenação de inteiros, o tamanho do problema é o tamanho do arranjo a ser ordenado. O tamanho de um problema envolvendo grafos pode ser o número de nodos. Por outro lado, em um algoritmo para listar todos os divisores de um número inteiro, o tamanho do problema é o valor desse inteiro. Desta forma, a análise da complexidade de um algoritmo começa com a identificação da variável inteira que representa o tamanho do problema.

O segundo passo no estudo da complexidade é definição do tipo de custo que se deseja avaliar, pois há dois tipos de custos a considerar: tempo de execução e consumo de memória. Esses dois aspectos são muito importantes para a avaliação e comparação de soluções algorítmicas. Entretanto, como a questão do custo de tempo da

execução surge com mais frequência no estudo de algoritmos e de suas aplicações, ela recebe aqui mais atenção.

Define-se, assim, que **Função de Custo** ou **Função de Complexidade** $f(n)$ é a medida do **tempo** necessário para executar um algoritmo para um problema de tamanho n. Note que o default é o custo de tempo. Quando tratar-se de consumo de memória, deve-se explicitá-lo claramente.

O terceiro passo consiste na associação de custo a cada instrução do algoritmo e então contabilizar o custo de todas as operações que são usadas na solução de problemas.

3 Cálculo do Custo

Um exemplo singelo de cálculo de função de custo de tempo é o do algoritmo clássico de pesquisa sequencial em tabelas, o qual, ou localiza a chave na tabela ou informa que ela não está presente.

No algoritmo de pesquisa, definido na Fig. 1, `a` é um arranjo contendo n inteiros, indexado de 1 a n. A posição `a[0]` nunca é ocupada, sendo usada apenas como sentinela de pesquisa, para acelerar o método de busca.

	Algoritmo	melhor	pior
1	`int pesquise(int ch, int[] a, int n){`	melhor	pior
2	`  int i = n;`	c_2	c_2
3	`  a[0] = ch;`	c_3	c_3
4	`  while(!(a[i] == ch))`	c_4	$(n+1)c_4$
5	`  i--;`	0	nc_5
6	`  if (i != 0)`	c_6	c_6
7	`  return i;`	$\}c_7 = c_8$	$\}c_7 = c_8$
8	`  else return -1;`		
9	`}`		

Figura 1: Pesquisa em Tabela I

A função `pesquise` retorna um inteiro que, se positivo, é o índice da chave encontrada, senão retorna -1 para indicar que a chave não está na tabela. A pesquisa sequencial inicia-se na posição n da tabela, e, a seguir, testam-se os elementos nas posições inferiores até que o elemento procurado seja encontrado ou que toda a tabela tenha sido pesquisada. Assim, do ponto de vista de custo, há pelo menos três casos a considerar nessa análise:

1. O elemento procurado está na posição n da tabela, a primeira posição inspecionada, sendo encontrado imediatamente, com apenas uma comparação de chaves, constituindo uma situação de *melhor caso*.

2. O elemento procurado não se encontra na tabela, a qual então será percorrida em toda sua extensão, isto é, serão feito $n + 1$ comparações, o que configura o *pior caso*.

3. O elemento procurado está na posição k, para algum k no intervalo $[1, n - 1]$, sendo o custo da busca um valor entre o de melhor e o de pior casos, dito um custo médio. Nesse caso, alguma suposição deve ser feita a respeito da distribuição dos dados no arranjo `a` e das probabilidades de ocorrências das chaves procuradas. Ambas as distibuições não são facilmente determináveis, sendo assim o custo médio é de difícil cálculo.

No algoritmo da Fig. 1, c_i, para i no intervalo $[2, 8]$, denota o custo de execução da instrução associada i.

No algoritmo acima estão anotados em cada linha, para o melhor e o pior casos, os custos de execução das instruções associadas. Por exemplo, a instrução `i--;` da linha (5) tem custo 0 no melhor caso, e nc_5, no pior caso. Ressalta-se que as instruções (7) e (8) são executadas de forma excludente, mas ambas têm o mesmo

custo, i.e., $c_7 = c_8$, o que permite dizer que, nesse algoritmo, o custo total de ambas é c_7, conforme anotado.

O custo do algoritmo deve ser calculado iniciando-se pelo corpo do comando de repetição mais interno, que na Fig. 1 é a linha (5). Assim, no caso de a chave não estar na tabela, essa linha, de custo c_5, é executada n vezes, onde n é o número de entradas válidas na tabela, e a linha (4) é executada $n + 1$ vezes. Esse é o pior caso.

No melhor caso, linha (5) nunca será executada, sendo 0 seu custo, e a linha (4) é executada uma única vez. As demais operações de `pesquise` são executadas uma única vez, observada a ressalva acima em relação às linhas (7) e (8).

Em resumo, no melhor caso, o custo de execução do algoritmo `pesquise` é:

$$f(n) = c_2 + c_3 + c_4 + c_6 + c_7 \tag{1}$$

que pode ser re-escrita como

$$f(n) = K \tag{2}$$

onde $K = c_2 + c_3 + c_4 + c_6 + c_7$ é uma constante dependente da plataforma em que o algoritmo for executado.

E no pior caso, a condição do comando `while` da linha (4) tem custo $(n + 1)c_4$, e o do comando de repetição é $(n + 1)c_4 + nc_5$, e o custo do algoritmo completo, nesse caso, é

$$f(n) = c_2 + c_3 + (n + 1)c_4 + nc_5 + c_6 + c_7 \tag{3}$$

que reduz-se a

$$f(n) = K_1 n + K_2 \tag{4}$$

onde $K_1 = c_4 + c_5$ e $K_2 = c_2 + c_3 + c_4 + c_6 + c_7$

Os valores exatos de K, K_1 e K_2 dependem da plataforma em que uma implementação do algoritmo `pesquise` for executada, portanto, aparentemente, contraria a premissa de que o custo

do algoritmo deve ser avaliado independentemente da sua implementação. Entretanto, a influência de K da Eq. (2) e de K_1 e K_2 da Eq. (4) no custo final torna-se insignificante para valores elevados de n.

Por outro lado, como o custo de cada instrução não é mesmo conhecido, e a influência das constantes computadas pode ser insignificante para grandes n, pode-se simplicar o cálculo do custo, ignorando as operações em que o número de vezes em que são executadas não depende do tamanho do problema. Para isso, basta contabilizar as operações que sejam efetivamente relevantes para determinação do custo final, como exemplificado a seguir.

Instruções Executadas

O que realmente interessa, para fins de comparação de algoritmos, é exatamente a determinação do comportamento do custo em relação ao tamanho da entrada. O relevante para o custo é o número de vezes que cada instrução é executada em função do tamanho da entrada. Nesse contexto, a análise de custos fica bastante simplicada, como mostram os cálculos anotados na Fig. 2.

	Algoritmo	Custos	
1	`int pesquise(int ch, int[] a, int n){`	`melhor`	`pior`
2	`  int i = n;`	1	1
3	`  a[0] = ch;`	1	1
4	`  while (!(a[i] == ch))`	1	$n+1$
5	`  i--;`	0	n
6	`  if (i != 0)`	1	1
7	`  return i;`	1	1
8	`  else return -1;`		
9	`}`		

Figura 2: Pesquisa em Tabela II

O custo, nesse caso, é o número total de vezes que as instruções são executadas. Note que, no melhor caso, a instrução (4) somente é executada uma vez, e a instrução (5), nenhuma vez. Entretanto, no pior caso, as frequências de execução dessas instruções são $n+1$ e n, respectivamente.

Assim, contabilizando a frequência de execução de todas as instruções de `pesquise`, chega-se rapidamente às funções finais de custo, que são:

$$\begin{aligned} f(n) &= 5 && \text{melhor caso} \\ f(n) &= 2n + 5 && \text{pior caso} \end{aligned}$$

É possível simplificar ainda mais o processo de cálculo do custo, sem perda de generalidade, observando-se que o número de vezes que as instruções (2), (3), (6), (7) e (8) são executadas independe de n, e, portanto, têm pouca influência no custo de problemas de grande porte, podendo ser ignoradas na análise de custo.

	Algoritmo	Custos	
1	`int pesquise(int ch, int[] a, int n){`	`melhor`	`pior`
2	`  int i = n;`		
3	`  a[0] = ch;`		
4	`  while (!(a[i] == ch))`	1	$n+1$
5	`  i--;`		
6	`  if (i != 0)`		
7	`  return i;`		
8	`  else return -1;`		
9	`}`		

Figura 3: Pesquisa em Tabela III

Além disso, as instruções (4) e (5) são executadas praticamente o mesmo número de vezes, i.e., $n + 1$ e n vezes, respectivamente, portanto, basta contar apenas uma deles, e.g, a linha (4), como mostra Fig. 3.

Essas simplificações permitem inferir imediatamente que:

- melhor caso: $f(n) = 1$, que ocorre quando o registro procurado é o primeiro consultado.

- pior caso: $f(n) = n + 1$, que ocorre quando o registro procurado não está presente na tabela.

Em resumo, no processo simplificado de determinação do custo do algoritmo em função do tamanho n do problema, ignora-se o custo das operações que têm pouca ou nenhuma influência no custo final para grandes n e consideram-se apenas as operações mais significativas.

Custo Médio

Mesmo com o método simplificado de análise de custos, o custo médio do `pesquise` pode ser ainda complexo, pois seu cálculo depende de outros fatores, como a distribuição probabilística das chaves procuradas.

Para ilustrar esse processo, suponha que toda pesquisa com o algoritmo da Fig. 3 recupere um registro e que não exista pesquisa sem sucesso.

Suponha também que p_i seja a probabilidade de o i-ésimo registro, para i no intervalo $[1, n]$, ser procurado, e considerando que para recuperar o i-ésimo registro são necessárias $n - i + 1$ comparações, então:

$$f(n) = \frac{1 \cdot p_n + 2 \cdot p_{n-1} + 3 \cdot p_{n-2} + \cdots + n \cdot p_1}{\sum_{i=1}^{n} p_i} \qquad (5)$$

Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n$, para $1 \leq i \leq n$ e $\sum_{i=1}^{n} p_i = 1$.

Nesse caso, obtém-se o custo médio $f(n)$:

$$f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2} \qquad (6)$$

Ou seja, a análise do caso esperado, nesse exemplo, com a suposição arbitrária de que todos os registros têm a mesma probabilidade de ser acessado, revela que uma pesquisa com sucesso examina, em média, aproximadamente metade dos registros. Entretanto, isso é somente válido no caso de as probabilidades de acesso aos elementos serem iguais. Em um caso real, pode não ser possível garantir isso.

4 Análise de Algoritmos Não-Recursivos

No cálculo dos custos de um programa que possui procedimentos, o tempo de execução de cada procedimento deve ser computado separadamente, um a um, iniciando-se com os procedimentos que não chamam outros procedimentos. A seguir, devem ser avaliados os procedimentos que chamam os procedimentos cujos tempos já foram avaliados. Esse processo deve ser repetido até se chegar ao programa principal.

Para cálculo de custo de cada um dos procedimentos deve-se:

- identificar a variável que representa o tamanho do problema;
- determinar as instruções consideradas relevantes para o cálculo do custo;
- contabilizar o número de vezes em que essas instruções são executadas em função do tamanho do problema.

Dorante, o custo de um algoritmo será visto como o número de vezes que suas instruções relevantes são executadas.

O processo de determinação do custo de cada instrução que contém operação destacada como relevante para o custo obedece as seguintes regras:

- As operações ditas não relevantes têm custo 0 e podem ser ignoradas no cálculo do custo.
- O custo de um comando de atribuição, de leitura ou de escrita que contém operação relevante normalmente pode ser considerado 1, exceto quando nele houver chamadas de funções, pois, nesse caso, seus custos de execução devem ser considerados.
- Se o comando de atribuição dito relevante envolver vetores de tamanho arbitrariamente grande, deve-se considerar os custos do processamento dos vetores.
- O tempo de execução de uma expressão relevante é normalmente 1, considerando a aplicação dos operadores envolvidos mais os tempos de avaliação de operandos que sejam constantes ou variáveis. Entretanto, caso operandos contenham chamadas de funções, seus custos devem ser considerados.
- O custo de execução de uma sequência de comandos é a soma dos custos dos comandos relevantes.
- O custo de execução de um comando de decisão é composto pelo custo de execução dos comandos relevantes executados dentro do comando condicional, mais o tempo para avaliar a condição, se ela contiver operação relevante para o custo.
- O custo de um comando de repetição é o somatório do custo de execução de cada iteração conforme o número de repetições definido, acrescido do custo do teste de término do laço. O custo de uma iteração é o custo da condição acrescida do custo do corpo do comando.

- O custo de uma chamada de procedimento é o custo desse procedimento para um problema cujo tamanho é o efinido por seus parâmetros implícitos ou explícitos.

Análise da Ordenação por Inserção

A seguir, o algoritmo de ordenação de um arranjo de inteiros, indexados de `1` a `n`, pelo método de inserção da Fig. 4 é usado para ilustrar a aplicação do método de cálculo de custo para o pior caso.

Nesse algoritmo, a instrução destacada como relevante é a de comparação de chaves `x < a[j]`, que ocorre na linha (5), e o tamanho do problema é n, o número de elementos no arranjo.

	Algoritmo	pior caso	
1	`void ordena(int[ ] a, int n){`	–	–
2	`  int i, j, x;`	–	–
3	`  for (int i = 2; i < n; i++) {`	i	$\sum_{i=2}^{n-1} i$
4	`  x = a[i]; j = i-1; a[0] = x;`		
5	`  while (x < a[j])`		
6	`  a[j+1] = a[j]; j--;`		
7	`  }`		
8	`  a[j+1] = x;`	–	–
9	`  }`		
10	`}`	–	–

Figura 4: Ordenação por Inserção

No comando `while` da linha (5) da Fig. 4, a instrução de comparação de chaves `x < a[j]`, no pior caso, é feita uma vez para cada `j` no intervalo $[0, i-1]$, portanto o custo dessa comparação de chaves é i, para $i \in [2, n-1]$.

O corpo do comando `for` da linha (3) tem o custo da única operação relevante nele contida, i.e., seu custo é i, pois o custo da demais operação não precisa ser considerado. Assim, o custo do comando `for` é $\sum_{i=2}^{n-1} i$.

Dado que $\sum_{i=1}^{n} i = \frac{n[n+1]}{2}$
tem-se que

$$\sum_{i=2}^{n-1} i = \sum_{k=1}^{n-2}(k+1) = \sum_{k=1}^{n-2} k + \sum_{k=1}^{n-2} 1 \tag{7}$$

ou

$$\sum_{i=2}^{n-1} i = \frac{(n-2)[(n-2)+1]}{2} + (n-2) \tag{8}$$

Portanto,

$$\sum_{i=2}^{n-1} i = \frac{n^2}{2} - \frac{n}{2} - 1 \tag{9}$$

Assim, o custo no pior caso do algoritmo de ordenação de n inteiros pelo método da inserção é:

$$f(n) = \frac{n^2}{2} - \frac{n}{2} - 1 \tag{10}$$

O melhor caso desse algoritmo ocorre quando o arranjo dado já estiver ordenado, pois a comparação `x < a[j]` somente será feita uma vez para cada valor de i, produzindo assim o custo:

$$f(n) = \sum_{i=2}^{n-1} 1 = n - 2 \tag{11}$$

Análise da Ordenação por Seleção

A título de ilustração, no algoritmo a seguir será considerado o número de comparações de elementos do arranjo `a` como custo relevante, e, a seguir, considerar-se-á o número de trocas de posição dos elementos do arranjo como a operação relevante determinante do custo.

Algoritmo	pior caso	
1 `void ordena(int[ ] a, int n){`	–	–
2 `  for (i = 1; i <= n-1; i++) {`	–	
3 `    min = i;`	–	
4 `    for (j = i+1; j <= n; j++) {`	$n-i$	$\sum_{i=1}^{n-1}(n-i)$
5 `      if (A[j] < A[min]) min = j;`		
6 `    }`		
7 `    x=A[min]; A[min]=A[i]; A[i]=x;`	–	
8 `  }`	–	
9 `}`	–	–

Figura 5: Ordenação por Seleção I

O custo do algoritmo de ordenação por seleção da Fig. 5, no pior caso, quando considera-se somente o custo da comparação de chaves de pesquisa com o elemento do arranjo, é:

$$f(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \tag{12}$$

Alternativamente, se número de trocas for considerado como a medida de custo relevante, tem-se a situação descrita na Fig. 6, que mostra que o programa ordenação por seleção realiza, tanto no melhor como no pior caso, exatamente $n-1$ trocas:

$$f(n) = \sum_{i=1}^{n-1} 1 = n-1 \tag{13}$$

Análise da Pesquisa Binária

Muitas vezes, a metodologia de contabilizar a operação eleita como relevante não é tão óbvia quanto a empregada nos exemplos apresentados, sendo mais apropriado o uso de abordagens específicas.

Para ilustrar o uso de metodologias especiais em casos particulares, considere o cálculo da função de custo do clássico algoritmo

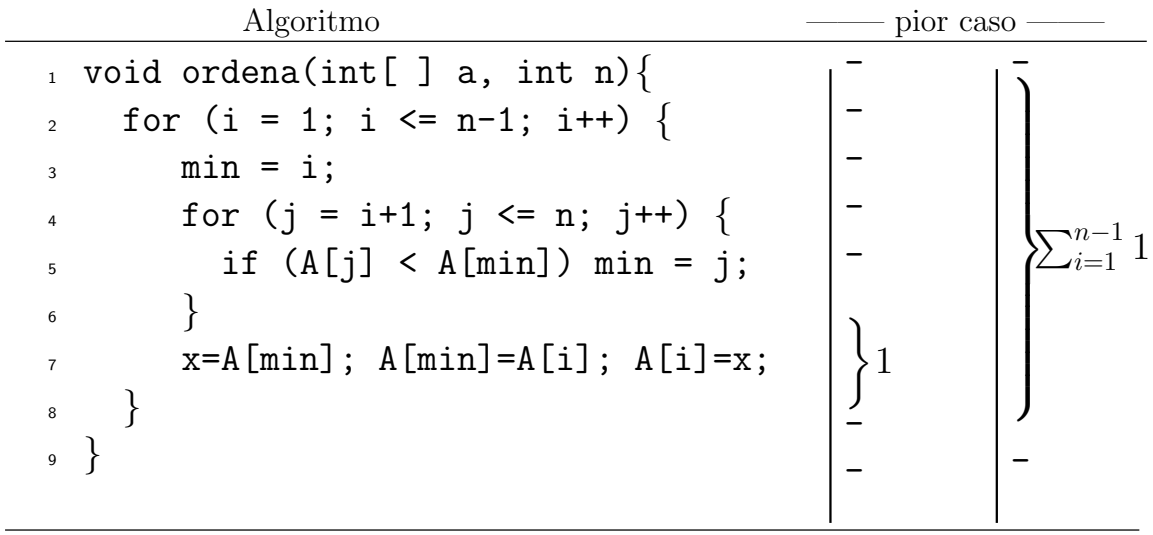

Figura 6: Ordenação por Seleção II

de pesquisa binária, que é uma técnica considerada eficiente para busca de chaves em tabelas ordenadas.

```
1  int pesquise(int ch, int[ ] a, int n) {
2     int i, esq, dir, cmp;
3     if (n <= 0) return 0;
4     esq = 1; dir = n;
5     do { i = (esq + dir)/2;
6          cmp = a[i].compareTo(ch);
7          if (cmp == 0) return i;
8          else if (cmp < 0) esq = i + 1;
9               else dir = i - 1;
10    while (esq <= dir);
11    return 0;
12 }
```

Figura 7: Pesquisa Binária

O algoritmo a ser analisado é o da Fig. 7, o qual, para fins de simplificação, pesquisa um inteiro `ch` em um arranjo `a` ordenado de n inteiros.

Para fins de cálculo do custo, suponha que n seja uma potência

de 2. Isso facilita os cálculos e não representa perda alguma de generalidade, pois o que interessa é que n seja um valor grande.

A função de custo do algoritmo é $f(n) = k$, onde k é o número de vezes que operação de comparação `cmp = a[i].compareTo(ch);` da linha (6) é executada no pior caso, que é quando o valor procurado não se encontra no arranjo.

A atividade central do algoritmo é o comando de repetição `do-while`, no qual, a cada iteração, o tamanho da parte da tabela a pesquisar é dividido ao meio.

Assim, inicialmente, quando a primeira comparação é realizada, o tamanho do arranjo a ser pesquisado é n. Na segunda iteração, esse tamanho torna-se $n/2$, e assim sucessivamente, até o elemento procurado ser encontrado ou o tamanho for reduzido a 1, como ilustra o quadro da Fig. 8.

#teste	Tamanho da Tabela
1	n
2	$n/2$
3	$n/4$
4	$n/8$
$\cdots$	$\cdots$
k	$n/2^{k-1} = 1$

Figura 8: Tamanho da Tabela em Análise

Note que, no pior caso, quando o elemento procurado não estiver no arranjo, a pesquisa somente para quando o tamanho do arranjo a ser pesquisado reduz-se a 1, isto é, quando $\frac{n}{2^{k-1}} = 1$.

Logo, o número de vezes k que o tamanho da parte do arranjo a ser pesquisada é dividido ao meio, no pior caso, é $\lg n + 1$.

Portanto, $f(n) = \lg n + 1$

5 Análise de Algoritmos Recursivos

No caso de **procedimento recursivo**, deve-se quebrar a circularidade associando ao procedimento uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos do procedimento, e assim gerar as equações de custo recorrentes, cujas soluções fornecem os custos desejados.

O custo da chamada recursiva é então definido em termos dessa função de complexidade desconhecida.

Assim, o problema da análise de custo de algoritmos recursivos compreende os seguintes passos:

1. determinar, a partir dos parâmetros do algoritmo, a variável n que denota o seu tamanho;
2. identificar a operação significativa a ser contada pela função de custo;
3. associar a cada procedimento recursivo uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos do procedimento;
4. postular que se a função de custo para o procedimento com parâmetros de tamanho n for $f(n)$, então o custo do mesmo procedimento com parâmetros de tamanho $g(n)$ é $f(g(n))$;
5. obter, a partir do algoritmo, uma equação de recorrência para $f(n)$;
6. resolver a equação de recorrência para obter a fórmula direta do custo.

Os principais procedimentos recursivos diretos, i.e., aqueles que chamam diretamente a si mesmo, podem ser estruturados de acordo

com um pequeno grupo de sete esquemas, cada qual gerando um tipo de equações de recorrência.

Nos esquemas apresentados a seguir, `void p(n)` representa a declaração do cabeçalho de um procedimento de nome `p` que recebe uma lista de parâmetros que caracterizam um problema de tamanho n, os quais não são explicitados, pois seu conhecimento é irrelevante para a análise de custo.

Em todos os esquemas apresentados a seguir, a operação relevante, determinante do custo, é `processa um elemento`, que tem custo 1, ou então `processo x elementos`, de custo x.

Algoritmos Recursivos Tipo I

A primeira classe de procedimentos recursivos são aqueles que terminam quando a entrada tem tamanho 0, e, com uma entrada com tamanho $n > 0$, processam um elemento e depois chamam-se recursivamente com parâmetros de tamanho $n - 1$, conforme mostra Fig. 9.

```
1 void p(n) {
2   if (n <= 0) termina
3   else {processa um elemento; p(n-1);}
4 }
```

Figura 9: Recursivo Tipo I

Seja $f(n)$ o custo postulado de `p(n)`, definido pelo número de elementos processados. A operação `processa um elemento` tem custo 1, a chamada recursiva `p(n-1)` tem custo $f(n-1)$, e o comportamento desse algoritmo é descrito pelo sistema de equações:

$$\begin{cases} f(n) = 1 + f(n-1), \text{ para } n > 0 \\ f(n) = 0, \text{ para } n \leq 0 \end{cases}$$

Para resolver esse sistema de equações, pode-se usar a própria recorrência

$$f(n) = 1 + f(n-1) \tag{14}$$

para calcular o termo $f(n-1)$ do lado direito, produzindo

$$f(n-1) = 1 + f(n-2)$$

E novamente usar Eq. (14) para calcular

$$f(n-2) = 1 + f(n-3)$$

E assim, concluir que se assim fosse feito sucessivamente, para um dado n, chegar-se-ia a

$$f(1) = 1 + f(0)$$

Agrupando essas equações, tem-se:

$$\left.\begin{array}{lcl} f(n) & = & 1 + f(n-1) \\ f(n-1) & = & 1 + f(n-2) \\ \cdots & = & \cdots \\ f(1) & = & 1 + f(0) \end{array}\right\} \text{n equações}$$

Adicionando as equações acima lado a lado:

$$\begin{array}{lcl} f(n) & = & 1 + \cancel{f(n-1)} \\ \cancel{f(n-1)} & = & 1 + \cancel{f(n-2)} \\ \cdots & = & \cdots \\ \cancel{f(1)} & = & 1 + f(0) \\ \hline f(n) & = & \displaystyle\sum_{i=1}^{n} 1 + f(0) \end{array}$$

Portanto, $f(n) = n$

Algoritmos Recursivos Tipo II

A segunda classe de procedimentos recursivos é formada por aqueles que, para um problema de tamanho 0, terminam imediatamente, e com problemas de tamanho $n > 0$, processam n elementos e chamam-se recursivamente com parâmetros de tamanho $n - 1$, conforme mostra Fig. 10.

```
1 void p(n) {
2   if (n <= 0) termina
3   else {processa n elementos; P(n - 1);}
4 }
```

Figura 10: Recursivo Tipo II

Supondo que `p(n)` tenha custo $f(n)$, o custo desse algoritmo é descrito pelas equações:

$$\begin{cases} f(n) = n + f(n-1), \text{ para } n > 0 \\ f(n) = 0, \text{ para } n \leq 0 \end{cases}$$

Usando o método apresentado anteriormente, pode-se gerar as equações, que são adicionadas lado a lado:

$$\left.\begin{array}{rcl} f(n) & = & n + \cancel{f(n-1)} \\ \cancel{f(n-1)} & = & n - 1 + \cancel{f(n-2)} \\ \cancel{f(n-2)} & = & n - 2 + \cancel{(n-3)} \\ \cancel{(n-3)} & = & n - 3 + \cancel{f(n-4)} \\ \cdots & = & \cdots \\ \cancel{f(n-k)} & = & n - k + f(\overbrace{n-k-1}^{0}) \end{array}\right\} k+1 \text{ equações}$$

$$f(n) = \sum_{i=0}^{k}(n-i) + f(0)$$

Para obter a função de custo desejada, basta computar o somatório presente na definição de $f(n)$, isto é:

$$f(n) = \sum_{i=0}^{k}(n-i) + f(0) \quad \text{e} \quad n-k-1=0$$

$$f(n) = \sum_{i=0}^{k} n + \sum_{i=0}^{k} i + 0$$

$$f(n) = (k+1)n + \frac{k(k+1)}{2} \quad \text{(veja Seção 10)}$$

$$f(n) = (\frac{n+n-k}{2})(k+1) + 0, \quad \text{onde } k = n-1$$

$$f(n) = (\frac{n+n-n+1}{2})(n-1+1)$$

Assim, tem-se a expressão final de custo: $f(n) = \frac{n^2+n}{2}$

Algoritmos Recursivos Tipo III

A terceira classe de procedimentos recursivos é formada pelos que, com uma entrada de tamanho 0, termina, e, com entrada de tamanho $n > 0$, processam um elemento e depois chamam-se recursivamente com parâmetros de tamanho $n/2$, conforme Fig. 11.

```
1 void p(n) {
2   if (n <= 0) termina
3   else {processa um elemento; p(n/2);}
4 }
```

Figura 11: Recursivo Tipo III

Esses algoritmos têm natureza logarítmica, pois dividem o espaço de trabalho em duas metades e processam recursivamente um elemento em cada metade até que o espaço tenha tamanho 1.

Para o cálculo de sua complexidade, supõe-se que n seja um inteiro potência de 2, que `p(n)` tenha custo $f(n)$, e, assim, o comportamento desse algoritmo é descrito pelas *equações de recorrência*:

$$\begin{cases} f(n) &= 1 + f(n/2), \text{ para } n > 1 \\ f(1) &= 1 \\ f(n) &= 0, \text{ para } n \leq 0 \end{cases}$$

Repetindo o processo realizado com o esquema Tipo I, usa-se a equação de recorrência

$$f(n) = 1 + f(n/2) \tag{15}$$

para calcular o termo $f(n/2)$ do lado direito

$$f(n/2) = 1 + f(n/2^2) \tag{16}$$

E assim, sucessivamente, até que se possa deduzir a última equação, onde supõe-se que n seja uma potência de 2:

$$f(n/2^{k-1}) = 1 + f(n/2^k) \tag{17}$$

onde $n/2^k = 1$ e, consequentemente, $k = \lg n$

Agrupando essas equações, tem-se:

$$\left.\begin{array}{lcl} f(n) & = & 1 + f(n/2) \\ f(n/2) & = & 1 + f(n/2^2) \\ f(n/2^2) & = & 1 + f(n/2^3) \\ f(n/2^3) & = & 1 + f(n/2^4) \\ \cdots & = & \cdots \\ f(n/2^{k-1}) & = & 1 + f(n/2^k) \end{array}\right\} k \text{ equações}$$

Adicionando-as lado a lado:

$$
\begin{array}{lcl}
f(n) & = & 1 + \cancel{f(n/2)} \\
\cancel{f(n/2)} & = & 1 + \cancel{f(n/2^2)} \\
\cancel{f(n/2^2)} & = & 1 + \cancel{f(n/2^3)} \\
\cancel{f(n/2^3)} & = & 1 + \cancel{f(n/2^4)} \\
\cdots & = & \cdots \\
\cancel{f(n/2^{k-1})} & = & 1 + f(n/2^k) \\
\hline
f(n) & = & \sum_{i=1}^{k} 1 + f(\underbrace{n/2^k}_{1}) \\
f(n) & = & k + f(1), \text{ onde } k = \lg n \text{ e } f(1) = 1
\end{array}
$$

Assim, obtém-se a expressão final de custo: $f(n) = \lg n + 1$

Algoritmos Recursivos Tipo IV

A quarta classe de procedimentos recursivos são aqueles que, com uma entrada de tamanho 0 terminam imediatamente, e, com entrada com tamanho $n > 0$, processam n elementos e depois chamam-se recursivamente com parâmetros de tamanho $n/2$.

```
1 void p(n) {
2   if (n <= 0) termina
3      else {processa n elementos; P(n/2);}
4 }
```

Figura 12: Recursivo Tipo IV

Assim, supondo que n seja um inteiro potência de 2, o custo $f(n)$ desse algoritmo no pior caso é descrito pelas *equações de recorrência*:

$$
\begin{cases}
f(n) = n + f(n/2), \text{ para } n > 1 \\
f(1) = 1 \\
f(n) = 0, \text{ para } n \leq 0
\end{cases}
$$

A solução desse sistema é obtida via uma conveniente mudança de variável: no lugar de $f(n) = n + f(n/2)$, usa-se $f(2^k) = 2^k + f(2^{k-1})$, trocando-se n por 2^k.

A solução da nova equação de recorrência fica assim:

$$\begin{array}{rcl}
f(2^k) & = & 2^k + \cancel{f(2^{k-1})} \\
\cancel{f(2^{k-1})} & = & 2^{k-1} + \cancel{f(2^{k-2})} \\
\cancel{f(2^{k-2})} & = & 2^{k-2} + \cancel{f(2^{k-3})} \\
\cdots & = & \cdots \\
\cancel{f(2^1)} & = & 2^1 + f(2^0) \\
\hline
f(2^k) & = & \displaystyle\sum_{i=1}^{k} 2^i + f(1)
\end{array}$$

$$\begin{array}{rcl}
f(2^k) & = & 2^{k+1} - 2 + 1 \\
f(2^k) & = & 2.2^k - 1
\end{array}$$

Portanto, $f(n) = 2n - 1$

Algoritmos Recursivos Tipo V

A quinta classe de procedimentos recursivos é formada por aqueles que, com uma entrada de tamanho 0, terminam imediatamente, e, com entrada com tamanho $n > 0$, processam um elemento e depois chamam-se recursivamente duas vezes com parâmetros de tamanho $n/2$, conforme apresentado na Fig. 13.

```
1 void p(n) {
2   if (n <= 0) termina;
3       else {processa 1 elemento; P(n/2); P(n/2);}
4 }
```

Figura 13: Recursivo Tipo V

Supondo que `p(n)` tenha custo $f(n)$, o comportamento desse algoritmo é descrito pelas equações:

$$\begin{cases} f(n) &= 1 + 2f(n/2), \text{ para } n > 1 \\ f(1) &= 1 \\ f(n) &= 0, \text{ para } n \leq 0 \end{cases}$$

Supondo que n seja uma potência inteira de 2, a expansão dos termos da equação de recorrência e subsequente adição desses termos lado a lado produzem:

$$\begin{array}{ll} f(n) & = 1 + \underline{2\cancel{f(n/2)}} \\ \underline{2\cancel{f(n/2)}} & = 2 + \underline{4\cancel{f(n/4)}} \\ \underline{4\cancel{f(n/4)}} & = 4 + \underline{8\cancel{f(n/8)}} \\ \underline{8\cancel{f(n/8)}} & = 8 + \underline{16\cancel{f(n/16)}} \\ \cdots & = \cdots \\ \underline{2^{k-1}\cancel{f(n/2^{k-1})}} & = 2^{k-1} + 2^k f(n/2^k) \\ \hline f(n) & = \sum_{i=0}^{k-1} 2^i + 2^k f(\underbrace{n/2^k}_{1}) \end{array}$$

Assim,

$$\begin{aligned} f(n) &= \sum_{i=0}^{k-1} 2^i + 2^k \\ f(n) &= 2^k - 1 + 2^k, \text{onde } 2^k = n \\ f(n) &= n - 1 + n \end{aligned}$$

Portanto, $f(n) = 2n - 1$

Algoritmos Recursivos Tipo VI

A sexta classe de procedimentos recursivos é a dos que, com uma entrada de tamanho 0, terminam, e, com entrada com tamanho

$n > 0$, processam n elementos e depois chamam-se recursivamente duas vezes com parâmetros de tamanho $n/2$, conforme Fig. 14.

```
1 void p(n) {
2   if (n <= 0) termina
3      else {processa n elementos; P(n/2); P(n/2);}
4 }
```

Figura 14: Recursivo Tipo VI

O custo desse algoritmo é descrito pelas equações:

$$\begin{cases} f(n) &= n + 2f(n/2), \text{ para } n > 1 \\ f(1) &= 1 \\ f(n) &= 0, \text{ para } n \leq 0 \end{cases}$$

Cuja solução é obtida via uma conveniente mudança de variável: no lugar de $f(n) = n + 2f(n/2)$, usa-se $f(2^k) = 2^k + 2f(2^{k-1})$, onde n foi trocado por 2^k.

A solução da nova equação de recorrência fica assim:

$$\left.\begin{array}{rcl} f(2^k) & = & 2^k + \cancel{2f(2^{k-1})} \\ \cancel{2f(2^{k-1})} & = & 2^k + \cancel{4f(2^{k-2})} \\ \cancel{4f(2^{k-2})} & = & 2^k + \cancel{8f(2^{k-3})} \\ \cdots & = & \cdots \\ \cancel{2^{k-1}f(2^1)} & = & 2^k + 2^k f(2^0) \end{array}\right\} k \text{ equações}$$

$$f(2^k) = \sum_{i=1}^{k} 2^k + 2^k f(1)$$

Donde, deduz-se que:

$$\begin{aligned} f(2^k) &= k2^k - 2^k \\ f(2^k) &= (k-1).2^k, \text{onde } k = \lg n \\ f(n) &= (\lg n - 1)2^{\lg n} \end{aligned}$$

Portanto, $f(n) = n \lg n - n$

Algoritmos Recursivos Tipo VII

A sétima classe de procedimentos recursivos compreende aqueles que, com uma entrada de tamanho 0, terminam imediatamente, e, com entrada com tamanho $n > 0$, processam um elemento e depois chamam-se recursivamente duas vezes com parâmetros de tamanho $n-1$, conforme Fig. 15.

```
1 void p(n) {
2   if (n <= 0) termina;
3      else {processa 1 elemento; P(n-1); P(n-1);}
4 }
```

Figura 15: Recursivo Tipo VII

O custo $f(n)$ do algoritmo pode ser descrito pelas equações:

$$\begin{cases} f(n) &= 1 + 2f(n-1) \\ f(0) &= 0 \end{cases}$$

Cuja solução advém dos cálculos:

$$\begin{array}{ll} f(n) & = 1 + \cancel{2f(n-1)} \\ \cancel{2f(n-1)} & = 2 + \cancel{4f(n-2)} \\ \cancel{4f(n-2)} & = 4 + \cancel{8f(n-3)} \\ \cancel{8f(n-3)} & = 8 + \cancel{16f(n-4)} \\ \cdots & = \cdots \\ \cancel{2^{k-1}f(n-(k-1))} & = 2^{k-1} + 2^k f(n-k) \\ \hline f(n) & = \sum_{i=0}^{k-1} 2^i + 2^k f(\underbrace{n-k}_{0}) \end{array}$$

Assim, $f(n) = \sum_{i=0}^{n-1} 2^i$ e, portanto, $f(n) = 2^n - 1$

Conclusão

Os algoritmos recursivos mais comuns têm um dos sete formatos apresentados ou alguma variação deles. Evidentemente, outras estruturas são perfeitamente possíveis, e a metodologia utilizada deve ser adaptada para tratar esses casos não previstos.

6 Complexidade Assintótica

As complexidades de tempo e de espaço são medidas como função do tamanho do problema ou da entrada do algoritmo. As técnicas descritas permitem definir o comportamento dos algoritmos à medida que a entrada cresce. Entretanto, toda formulação apresentada é bastante imprecisa, porque muitas aproximações são feitas durante a análise dos algoritmos, tais como:

- os cálculos são feitos sem que o algoritmo seja executado em qualquer máquina e, portanto, os custos reais das operação são ignorados;
- supõe-se que toda instrução tenha o mesmo custo, contabilizando apenas sua frequência de execução;
- o custo da maioria das instruções dos algoritmos analisados é ignorada, sendo contabilizado somente os das operações identificadas como relevantes;
- instruções relevantes são supostas serem executadas em toda iteração dos comandos de repetição que as contêm, mesmo quando ocorrem em comandos condicionais;
- a análise de custo médio é geralmente muito difícil, prevalecendo quase sempre a do pior caso.

Quando trabalha-se com problemas de pequeno porte, as decisões de conduta de análise listadas acima podem invalidar totalmente o custo apurado. Entretanto, quando o tamanho do problema assume altas proporções, os comportamentos dos algoritmos tendem a se diferenciarem, separando-se em classes de complexidade.

O comportamento limite da complexidade quando a entrada cresce é chamado de complexidade assintótica de tempo ou de espaço, ou simplesmente, complexidade assintótica, a qual define o limite de tamanho dos problemas que um dado algoritmo pode resolver.

Ressalta-se que independentemente do aumento da velocidade de cálculo dos computadores, o que realmente determina o limite do tamanho dos problemas que podem ser resolvidos com um dado algoritmo é a sua complexidade.

Para facilitar o tratamento das aproximações listadas e permitir a eliminação de detalhes que são irrelevantes para a análise do custo de algoritmos, usa-se a notação $O(g(n))$ (leia-se *ordem de* $g(n)$ *ou ordem de no máximo* $g(n)$), a qual denota uma quantidade que não é explicitamente conhecida, mas que define com clareza seu comportamento.

Isto é, a notação $O(g(n))$ propõe substituir a real expressão da função de custo, que frequentemente não é conhecida, por uma regra de comportamento bem definida e mais simples, que permite fazer comparações dos custos de algoritmos.

Diz-se que um algoritmo tem custo $O(g(n))$, mas deve-se estar ciente de que as técnicas de análise de custo de algoritmos apresentadas permitem calcular a função $g(n)$, a qual serve para inferir o comportamento da real e desconhecida função de custo $f(n)$, e nunca, sua exata expressão.

Dominação Assintótica

Sejam $f(n)$ a real função de custo desconhecida de um algoritmo, e $g(n)$, sua função de custo computada, considerando n o tamanho do problema.

Definição 1 *$O(g(n))$ denota um conjunto de funções $f(n)$ tais que $|f(n)| \leq c|g(n)|$, para c constante.*

A notação $O(g(n))$ permite definir o chamado comportamento assintótico de outras funções, como a função de custo $f(n)$, pois, representa o *comportamento do custo* quando tamanho n cresce. Trata-se de definir a **taxa de crescimento** ou **ordem de crescimento** da função de custo.

Definição 2 *$f(n) = O(g(n))$ ou $f(n) \in O(g(n))$, se $\exists$ constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é menor ou igual a $cg(n)$.*

Note que a definição 2 estabelece que $O(g(n))$ é um conjunto de funções, e a notação $f(n) = O(g(n))$ é apenas uma nova forma de dizer que $f(n) \in O(g(n))$, portanto, o sinal "=", nesse contexto, tem significado distinto do da álgebra.

De acordo com essa definição, o tempo T de execução do algoritmo de custo $f(n) = O(g(n))$, isto é, *$f(n)$ da ordem de $g(n)$*, em uma máquina M e n suficientemente grande, é limitado superiormente por um valor proporcional a $g(n)$, isto é, $T \leq K.g(n)$, onde K é uma constante que depende das características de M.

Definição 3 *Uma função $g(n)$ domina assintoticamente outra função $f(n)$, se existem duas constantes n_0 e c, tais que para $n \geq n_0$, temos $|f(n)| \leq c.|g(n)|$.*

No gráfico da Fig. 16, a curva em vermelho seria a função de custo real e desconhecida, mostrando que $g(n)$ é um limite assintótico superior para $f(n)$.

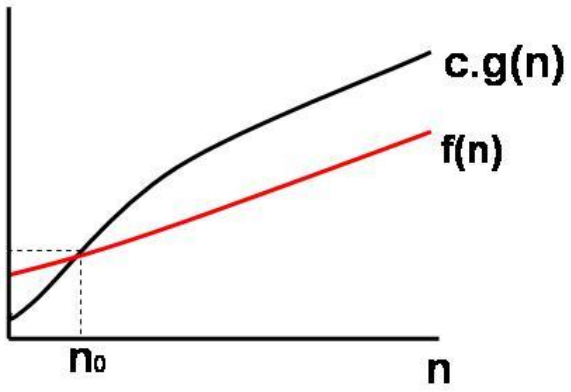

Figura 16: Dominação Assintótica

Para uma melhor compreensão do conceito de dominação assintótica, considere as funções:

$$\begin{cases} f(n) = n - 2 \\ g(n) = n^2 - 4 \end{cases}$$

Note que $|n - 2| \leq |c(n^2 - 4)|$ para $n \geq 2$ e $c \geq 1/4$, portanto, $g(n)$ domina assintoticamente $f(n)$, i.e., $f(n) = O(g(n))$.

Por outro lado, $f(n)$ não domina $g(n)$, porque $\nexists\, c > 0$ tal que $|n^2 - 4| \leq c|n - 2|$, para todo $n > 2$, i.e., $g(n) \notin O(f(n))$

A dominação assintótica pode ser uma relação unidirecional, como no exemplo acima, mas pode também ser bidirecional, como mostra o seguinte exemplo:

$$\begin{cases} f(n) = 10n^2 \\ g(n) = 20n^2 \end{cases}$$

no qual, para todo $n > 1$, tem-se que $|10n^2| \leq |20n^2|$, portanto, $g(n)$ domina assintoticamente $f(n)$, e que $|20n^2| \leq 2|10n^2|$, significando que $f(n)$ também domina $g(n)$.

Se as funções de custo f e g dominam assintoticamente uma a outra, então os algoritmos associados têm custos assintoticamente equivalentes. Nesses casos, o comportamento assintótico não serve para compará-los.

Obtenção da Ordem

A vantagem do uso de $O(f(n))$ é que ordem de crescimento permite uma caracterização mais simples da função de custo. A simplificação, sem perda de precisão, da expressão de custo $O(f(n))$ é amparada pelas seguintes regras:

- se $f(n)$ for uma soma de vários fatores, pode-se descartar todos exceto o que tiver maior taxa de crescimento.
- se o único termo resultante da simplificação acima contiver coeficientes constantes, pode-se trocá-los pela constante 1.

Para exemplificar a aplicação dessas regras, considere a expressão $O(5n^4 - 10n^3 + 4n^2 + n + 1000)$, na qual claramente o termo $5n^4$ é o que tem a maior taxa de crescimento. Assim, pelas regras acima, essa expressão reduz-se a $O(n^4)$.

Por essas mesmas regras, o algoritmo de ordenação pelo método de inserção, cujo custo no pior caso é $f(n) = n^2/2 - n/2 - 1$, tem uma ordem de crescimento de sua complexidade assintótica definida por $O(n^2)$.

Note que seria um abuso dizer que o tempo de execução do algoritmo de ordenação por inserção é $O(n^2)$, haja vista que tudo que se sabe é que o tempo real de execução é apenas limitado superiormente por cn^2, para algum c. O que se quer dizer quando se fala que *o tempo de execução* é $O(n^2)$ é que no pior caso o tempo de execução é proporcional a n^2, para n suficientemente grande.

Ressalta-se, contudo, que para n pequeno, outros fatores têm maior influência na comparação de desempenho de algoritmos, nesses casos, outros modelos de apuração de custo devem ser usados.

Expressões de Complexidade

No processo de análise da complexidade de um algoritmo, custos de trechos do algoritmo em análise devem ser combinados para compor o seu custo final. Nesse processo, expressões com a notação O devem ser combinadas e avaliadas, segundo as regras próprias, que estão listadas abaixo, onde, c é uma constante, n denota o tamanho do problema, $f(n)$, uma função de custo, $h(n)$, outra função de custo, e $t(n, i)$, uma função intermediária de custo que, além de n, depende do valor de uma variável i do algoritmo.

As regras básicas de operação com a notação O são:

- $f(n) = O(f(n))$:
  um trecho do algoritmo de custo $f(n)$ pode ser considerado de custo assintótico $O(f(n))$.

- $f(n) = O(1)$:
  o custo de operações que não dependem do tamanho n do problema, i.e., quando $f(n) = c$, tem custo $O(1)$.

- $f(n) = O(c + g(n))$ então $f(n) = O(g(n))$:
  a combinação de dois trechos de algoritmo de custos c e $O(g(n))$, respectivamente, tem custo $O(g(n))$.

- $f(n) = O(g(n))+O(h(n))$, então $f(n) = O(max(g(n), h(n)))$:
  a combinação de dois trechos de algoritmo de custos $O(g(n))$ e $O(h(n))$, respectivamente, tem custo $O(max(g(n), h(n)))$, onde max é uma função que retorna o termo de maior custo conforme a ordem de complexidade definida na Fig. 20.

- $f(n) = O(g(n)) + O(g(n))$, então $f(n) = O(g(n))$:
  a combinação de dois trechos de algoritmo de custos individuais $O(g(n))$ tem o custo combinado $O(g(n)) + O(g(n))$, que é o mesmo que $O(g(n))$.

- $f(n) = c.O(g(n))$, então $f(n) = O(g(n))$:
  a execução c vezes de um trecho de algoritmo de custo $O(g(n))$ tem custo final $c.O(cg(n))$, que reduz-se à $O(g(n))$.

- $f(n) = \sum_{i=a}^{b}(O(t(n,i))$, então $f(n) = O(\sum_{i=a}^{b} t(n,i))$:
  a execução repetida i vezes, para i no intervalo $[a,b]$, de um trecho de algoritmo de custo $O(t(n,i))$ tem custo $O(\sum_{i=a}^{b} t(n,i))$.

- $f(n) = O(g(n)).O(h(n))$, então $f(n) = O(g(n).h(n))$:
  a execução $O(g(n))$ vezes de um trecho de custo $O(h(n))$ tem custo final $O(g(n).h(n))$.

- $f(n) = g(n).O(h(n))$, então $f(n) = O(g(n).h(n))$:
  a execução $g(n)$ vezes de um trecho de custo $O(h(n))$ tem custo $O(g(n).h(n))$.

Essas transformações são muito úteis quando se combinam os custos de trechos de algoritmo, e deseja-se manter as expressões de custo o mais simples possível.

Cálculo da Complexidade Assintótica

Para ilustrar as operações com a notação $O(n)$, considere o cálculo do custo do algoritmo de ordenação de um arranjo de n inteiros pela método da seleção, mostrado na Fig. 17.

A análise usada contabiliza todas as instruções que são executadas, e supõe-se que n seja o tamanho da entrada de dados e que toda instrução tenha custo 1.

```
1  void ordena(int[ ] a, int n){
2    for (i = 1; i <= n-1; i++) {
3       min = i;
4       for (j = i+1; j <= n; j++) {
5         if (A[j] < A[min])
6             min = j;
7       }
8       x=A[min]; A[min]=A[i]; A[i]=x;
9    }
10 }
```

Figura 17: Ordenação por Seleção I

Inicia-se a análise pelo comando de repetição mais interno, destacado na Fig. 18, que contém um comando de decisão que contém apenas um comando de atribuição, que leva um tempo constante para ser executado, assim como a avaliação da sua condição.

```
4  for (j = i+1; j <= n; j++) {
5    if (A[j] < A[min])
6       min = j;
7  }
```

Figura 18: Loop Interno de Seleção I

Como não se sabe se o corpo do comando de decisão será executado ou não, então deve-se considerar o pior caso, ou seja, assumir que a linha (6) será sempre executada. Assume-se também que o tempo para incrementar a variável de controle da repetição e avaliar sua condição de terminação seja 1. Assim, o tempo combinado para executar uma vez o corpo da repetição composta pelas linhas (5) e (6) é 3, portanto $O(3)$, que é o mesmo que $O(1)$, o qual simplifica-se para 1.

Como o número de iterações da repetição é $n - i$, o tempo total gasto nesse comando interno de repetição é $(n - i) \times 1 + 1$, considerando o teste final de saída. Portanto, o custo do comando `for` interno acima é $1+(n-i)$, que pode ser simplicado para $(n-i)$, pois $O(1 + (n - i))$ é mesmo que $O(n - i)$, e $(n - i) \in O(n - i)$.

```
2         for (i = 1; i <= n-1; i++) {
3             min = i;
4,5,6,7       -----------------> custo n-i
8             x = A[min]; A[min] = A[i]; A[i] = x;
9         }
```

Figura 19: Loop Externo de Seleção I

O corpo do `for` mais externo, destacado na Fig. 19, contém o `for` interno e os comandos de atribuição nas linhas (3) e (8). Logo, o tempo de execução das linhas (3) a (9) é:

$$1 + (n - i) + 1 + 1 + 1 = (n - i) + 4$$

que pode ser simplificado para $(n - i)$, pois, $O((n - i) + 4)$ é o mesmo que $O(n - i)$, e $(n - i) \in O(n - i)$.

A linha (2), que tem a condição do `for` mais externo, é executada $n-1$ vezes, e cada iteração do `for` tem custo $n-i$, calculado acima. Assim, o tempo total $f(n)$ para executar o programa é:

$$f(n) = \sum_{i=1}^{n-1}(n - i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Portanto,

$$f(n) \;=\; O(\frac{n^2}{2} - \frac{n}{2})$$

$$f(n) \;=\; O(n^2)$$

7 Ordens de Complexidade

As classes de comportamento assintóticos comumente associadas a algoritmos são classificadas, conforme apresentado na Fig. 20, pela taxa de crescimento de seus custos. As funções de crescimento mais lento são classificadas como de menor ordem do que as de crescimento mais rápido. Essa ordenação das expressões de complexidade provê uma dimensão para comparar algoritmos.

A complexidade de menor ordem é a constante $O(1)$, que não depende do tamanho n do problema, e a maior delas é a exponencial $O(c^n)$, que define o comportamento de problemas intratáveis, de crescimento muito rápido.

Ordem	Complexidade	Nome
1	$O(1)$	complexidade constante
2	$O(\lg n)$	complexidade logarítmica
3	$O(n)$	complexidade linear
4	$O(n \lg n)$	complexidade linear logarítmica
5	$O(n^2)$	complexidade quadrática
6	$O(n^3)$	complexidade cúbica
7	$O(2^n)$	complexidade exponencial
8	$O(c^n)$	complexidade exponencial ($c > 2$)

Figura 20: Ordem de Custos das Classes de Complexidade

Complexidade Constante: $f(n) = O(1)$

Algoritmos de complexidade $O(1)$ são ditos de complexidade constante. Seu custo de execução independe do tamanho de n, e, assim, seu tempo de execução $f(n)$ é limitado por uma constante K, cujo valor é fixado a partir da velocidade do processador, pois as instruções do algoritmo são executadas um número fixo de vezes.

Complexidade Logarítmica: $f(n) = O(\lg n)$

Um algoritmo de complexidade $O(\lg n)$ é dito ter complexidade logarítmica, sendo seu tempo de execução $f(n) \leq K \lg n$, onde K é uma constante de tempo definido para cada processador. Esse tempo de execução ocorre em algoritmos que resolvem um problema pela sua divisão iterativa em duas partes e buscando a solução em uma das partes.

Esses algoritmos são muito eficientes, pois é pequena a influência do tamanho da entrada no custo final. Observe o efeito no custo do crescimento do tamanho n:

- quando n é 1.000, $\lg n \approx 10$
- quando n é 32.000, $\lg n \approx 16$
- quando n é um milhão, $\lg n \approx 20$

Complexidade Linear: $f(n) = O(n)$

Um algoritmo de complexidade $O(n)$ é dito ter complexidade linear. Seu tempo de execução $f(n) \leq K.n$, onde K é a constante de tempo do processador. Em geral, nessa classe de algoritmos, todos elementos da entrada são processados uma única vez. Note que cada vez que n dobra de tamanho, o tempo de execução apenas dobra.

Complexidade Linear Logarítmica $f(n) = O(n \lg n)$

Algoritmos de complexidade $O(n \lg n)$ são ditos de complexidade linear logarítmica. Tempos de execução dessa classe ocorrem em algoritmos que abordam um problema quebrando-o, recursivamente, em problemas menores, os quais são resolvidos de forma independente, e depois as soluções dos subproblemas são combinadas. Seu

tempo de execução $f(n) \leq Kn \lg n$, onde K é a constante do processador. Observe que quando n é um milhão, $n \lg n$ é cerca de 20 milhões. e quando n é dois milhões, $n \lg n$ é cerca de 42 milhões, apenas um pouco mais do que o dobro.

Complexidade Quadrática: $f(n) = O(n^2)$

Um algoritmo de complexidade $O(n^2)$ é dito ter complexidade quadrática. Seu tempo de execução $f(n) \leq Kn^2$, onde K é a constante do processador. Algoritmos dessa complexidade executam as operações mais significativas para o custo dentro de comandos de repetição aninhados em dois níveis. Observe que nesse caso, um custo cresce com o quadrado do tamanho, o que limita algoritmos desse tipo à solução de problemas de tamanhos relativamente pequenos.

Complexidade Cúbica: $f(n) = O(n^3)$

Um algoritmo de complexidade $O(n^3)$ é dito ter complexidade cúbica. Seu tempo de execução $f(n) \leq Kn^3$, onde K é a constante do processador. Os custos de algoritmos dessa ordem de complexidade crescem muito rapidamente, sendo, assim, úteis apenas para resolver problemas pequenos. Observe que sempre que n é multiplicado por um fator k, o tempo de execução fica multiplicado por k^3, atingido valores muito altos rapidamente.

Complexidade Polinomial: $f(n) = O(n^d)$

Um algoritmo cuja função de complexidade é $O(p(n))$, tal que $p(n) = \sum_{i=0}^{d} a_i n^i$, onde a_0, a_1, a_2, $\cdots$, a_d são coeficientes e $a_d > 0$, é chamado de algoritmo polinomial no tempo de execução. Como

o termo de maior crescimento $a_d n^d$ sempre prevalece sobre os demais, no lugar de $O(p(n))$ escreve-se apenas $O(n^d)$, para $d \geq 0$. Essa classe de complexidade inclui as ordens de complexidade para polinômios de grau zero $[O(1)]$, grau um $[O(n)]$, grau dois $[O(n^2)]$ e grau três $[O(n^3)]$.

Os algoritmos polinomiais geralmente resultam de um entendimento mais profundo da estrutura do problema, que evita o uso de *força bruta* na busca de sua solução. De um ponto de vista teórico, um problema é considerado *intratável* se para ele não existe um algoritmo polinomial capaz de resolvê-lo.

Dito de outra forma, um problema é considerado bem resolvido quando existe um algoritmo polinomial para resolvê-lo. Ou então que um algoritmo só pode se considerado eficiente se ele executa em tempo polinomial.

Complexidade Exponencial: $f(n) = O(c^n)$

Um algoritmo de complexidade $O(c^n)$, para $c \geq 2$ é dito ter complexidade exponencial. Seu tempo de execução $f(n) \leq K.c^n$, onde K é a constante do processador. Algoritmos dessa ordem de complexidade geralmente não são úteis sob o ponto de vista prático, a não ser para valores de n muito pequenos. Eles ocorrem na solução de problemas na qual se usa **força bruta** ou **busca exaustiva** para resolvê-los. Observe que para $c = 2$, quando n é vinte, o tempo de execução é cerca de um milhão, e, quando n dobra, o tempo de execução fica elevado ao quadrado.

8 Complexidade X Velocidade do Processador

Na solução de problemas de grande porte, isto é, problemas cujos tamanhos sejam consideráveis, a classe de complexidade do

algoritmo usado tem maior impacto no tamanho máximo dos problemas que podem ser resolvidos em um dado tempo de execução do que a velocidade do processor utilizado.

Considere, a título de argumento, o efeito no tamanho do problema que pode ser resolvido em um dado tempo por algumas classes de algoritmos quando se aumenta a velocidade do processador.

Para esse efeito, sejam K_A o fator de custo do processador A, $K_B = K_A/16$, o fator de custo de um processador B, que seja 16 vezes mais rápido, e sejam t, o tempo máximo admitido no execução algoritmo, n_A, o tamanho do problema que um algoritmo consegue resolver no tempo máximo t admitido, quando executado na máquina A, n_B, o tamanho máximo do mesmo problema na máquina B e $f(n)$, a função de complexidade de custo do algoritmo.

Caso Logaritmo: Algoritmo A1 – $O(\lg n)$

Se $f(n) = O(\lg n)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.\lg n_A \\ t = K_B.\lg n_B \end{cases}$$

Portanto,

$$\begin{aligned} K_A.\lg n_A &= K_B.\lg n_B \\ K_A.\lg n_A &= \tfrac{K_A}{16}.\lg n_B \\ \lg n_B &= 16.\lg n_A \end{aligned}$$

Donde, tem-se que $\boxed{n_B = n_A^{16}}$

Caso Linear: Algoritmo A2 – $O(n)$

Se $f(n) = O(n)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.n_A \\ t = K_B.n_B \end{cases}$$

Portanto,

$$\begin{aligned} K_A.n_A &= K_B.n_B \\ K_A.n_A &= \tfrac{K_A}{16}.n_B \end{aligned}$$

Donde, tem-se que $\boxed{n_B = 16n_A}$

Caso Linear Logarítmica: Algoritmo A3 – $O(n \lg n)$

Se $f(n) = O(n \lg n)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.n_A.\lg n_A \\ t = K_B.n_B.\lg n_B \end{cases}$$

Portanto,

$$\begin{aligned} K_A.n_A.\lg n_A &= K_B.n_B.\lg n_B \\ K_A.n_A.\lg n_A &= \tfrac{K_A}{16}.n_B.\lg n_B \\ n_B.\lg n_B &= (16 \lg n_A).n_A \\ n_B &= 16(\frac{\lg n_A}{\lg n_B}).n_A \end{aligned}$$

Considerando que n_B é ligeiramente maior que n_A, haja vista os resultados de algoritmos com $f(n) = O(n)$, e que a curva dos logaritmos é de lento crescimento, pode-se concluir que $\boxed{n_B \approx 16n_A}$.

Caso Quadrática: Algoritmo A4 – $O(n^2)$

Se $f(n) = O(n^2)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.n_A^2 \\ t = K_B.n_B^2 \end{cases}$$

Portanto,

$$\begin{aligned} K_A.n_A^2 &= K_B.n_B^2 \\ K_A.n_A^2 &= \tfrac{K_A}{16}.n_B^2 \\ n_B^2 &= 16.n_A^2 \end{aligned}$$

Donde, tem-se que $\boxed{n_B = 4n_A}$

Caso Cúbica: Algoritmo A5 – $O(n^3)$

Se $f(n) = O(n^3)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.n_A^3 \\ t = K_B.n_B^3 \end{cases}$$

Portanto,

$$\begin{aligned} K_A.n_A^3 &= K_B.n_B^3 \\ K_A.n_A^3 &= \tfrac{K_A}{16}.n_B^3 \\ n_B^3 &= 16.n_A^3 \end{aligned}$$

Donde tem-se que $\boxed{n_B = 2,51n_A}$

Caso Exponencial: Algoritmo A6 – $O(2^n)$

Se $f(n) = O(2^n)$, então, no pior caso, têm-se

$$\begin{cases} t = K_A.2^{n_A} \\ t = K_B.2^{n_B} \end{cases}$$

Portanto,

$$\begin{aligned} K_A.2^{n_A} &= K_B.2^{n_B} \\ K_A.2^{n_A} &= \tfrac{K_A}{16}.2^{n_B} \\ 2^{n_B} &= 16.2^{n_A} \\ 2^{n_B} &= 2^{4+n_A} \end{aligned}$$

Donde tem-se que $\boxed{n_B = n_A + 4}$

Resumo

Tabela 21 resume a influência do velocidade processador no tamanho do problema que um dado algoritmo pode resolver, quando fixado o tempo máximo de execução. Claramente, vê-se a complexidade é o fator determinante do tamanho máximo do problema que se pode resolver.

Algoritmo	Complexidade de Tempo	Tamanho máximo para uma dada Velocidade	Tamanho máximo para velocidade 16 vezes maior
A1	$\lg n$	N	N^{16}
A2	n	N	$16N$
A3	$n \lg n$	N	$\approx 16N$
A4	n^2	N	$4N$
A5	n^3	N	$2,51N$
A6	2^n	N	$N + 4$

Figura 21: Influência da Velocidade do Processador

9 Outros Limites Assintóticos

Além da popular notação-O, há outras que provêem informações adicionais sobre o comportamento assintótico de algoritmos. As principais são Θ, Ω, o e ω.

Notação Θ

Diz-se que $f(n) = \Theta(g(n))$, que se lê $f(n)$ *é da ordem exata de* $g(n)$, se existirem constantes positivas $c1, c2$ e n_0 tais que, para $n \geq n_0$, o valor de $f(n)$ estiver sempre entre $c_1 g(n)$ e $c_2 g(n)$.

Se $f(n) = \Theta(g(n))$, a função $g(n)$ é um limite assintótico firme ou restrito (*asymptotically tight bound*) para $f(n)$, pois, $g(n)$ limita $f(n)$ acima e abaixo.

Assim, $f(n) = \Theta(g(n))$ significa que $f(n)$ cresce assintoticamente tão rapidamente quanto $g(n)$, como ilustra a Fig. 22.

Enquanto a notação-O trata do limite superior de funções de custo no pior caso, a notação-Θ está ligada aos limites inferior e superior do custo médio.

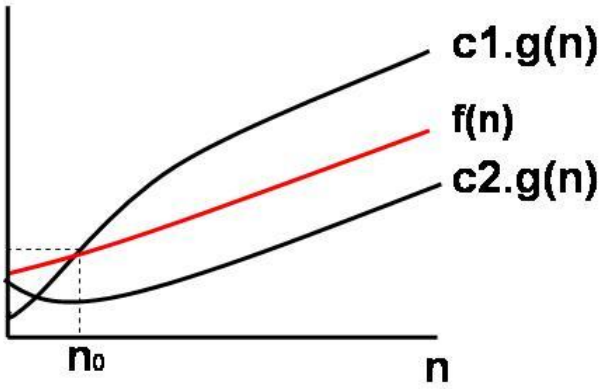

Figura 22: Limite Assintótico Firme

Formalmente, define-se a notação Θ como:

Definição 4 $f(n) = \Theta(g(n))$, $\exists\ c_1 > 0, c_2 > 0 \quad e \quad n_0 \geq 0$, *tais que* $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall\ n \geq n_0$.

Notação o

A notação o é definida como:

Definição 5 $f(n) = o(g(n))$, $\forall\ c > 0 \quad e \quad n_0 \geq 0$, *tais que* $0 \leq f(n) < cg(n),\ \forall\ n \geq n_0$.

Intuitivamente, pela Definição 5, a função $f(n)$ tem um crescimento muito menor que g(n) quando n tende para infinito. Isto pode ser expresso da seguinte forma:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

que pode ser visto como uma definição alternativa da notação o.

Embora as definições das notações O (O grande) e o (o pequeno) sejam similares, há uma diferença no fato de a expressão $0 \leq f(n) \leq cg(n)$ ser válida para **todas** as constantes $c > 0$ no caso de $f(n) = o(g(n))$, e, no caso de O, requer-se apenas que **exista** uma constante c que garanta a validade dessa expressão. As notações O e o definem limites assintóticos que não são necessariamente firmes, pois ambas não especificam limites inferiores.

Notação Ω

Diz-se que $f(n) = \Omega(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é maior ou igual a $cg(n)$. Portanto, $g(n)$ é um limite assintótico inferior (*asymptotically lower bound*) para $f(n)$, como sugere a Fig. 23. Formalmente, tem-se:

Definição 6 *$f(n) = \Omega(g(n))$, se $\exists$ constantes $c > 0$ e $n_0 > 0$, tais que $0 \leq cg(n) \leq f(n)$, $\forall\ n \geq n_0$.*

Observe que quando a notação Ω é usada para expressar o tempo de execução de um algoritmo no melhor caso, define-se também o limite inferior do tempo de execução para *todas* as entradas.

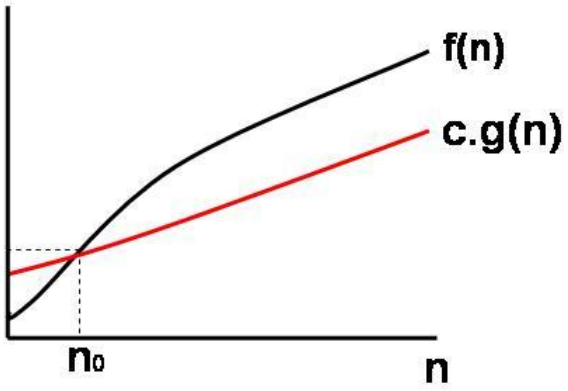

Figura 23: Limite Assintótico Inferior

Quando se fala que *o tempo de execução* é $\Omega(g(n))$ quer-se dizer que o tempo de execução desse algoritmo é de ordem de no

mínimo uma constante vezes $g(n)$ para grandes valores de n. Por exemplo, o algoritmo de ordenação por inserção é $\Omega(n)$ no melhor caso, então, seu tempo de execução é no mínimo $\Omega(n)$.

Notação ω

Por analogia, a notação ω está relacionada com a notação Ω da mesma forma que a notação o está relacionada com a notação O. Formalmente, a notação ω é definida da seguinte forma:

Definição 7 $f(n) = \omega(g(n))$, *se* $\forall$ *constante* $c > 0$, $\exists$ $n_0 > 0$ *tal que* $0 \leq cg(n) < f(n)$, $\forall$ $n \geq n_0$

Embora as definições das notações Ω e ω sejam similares, há uma diferença importante que é que, em $f(n) = \omega(g(n))$, a expressão $0 \leq cg(n) < f(n)$ é válida para **todas** as constantes $c > 0$.

A relação $f(n) = \omega(g(n))$ implica em

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty,$$

se o limite existir.

Relação Entre O, o, Ω, Θ, ω

Com base nas definições apresentadas, pode-se dizer que:

- $f(n) = O(g(n))$ significa que $f(n)$ cresce assintoticamente mais lentamente que $c.g(n)$, para alguma constante c
- $f(n) = o(g(n))$ significa que $f(n)$ cresce assintoticamente mais lentamente que $c.g(n)$, para toda constante c
- $f(n) = \Theta(g(n))$ significa que $f(n)$ cresce assintoticamente tão rapidamente quanto $g(n)$

- $f(n) = \Omega(g(n))$ significa que $f(n)$ cresce assintoticamente mais rapidamente que $c.g(n)$, para alguma constante c
- $f(n) = \omega(g(n))$ significa que $f(n)$ cresce assintoticamente mais rapidamente que $c.g(n)$, para toda constante c

Os comportamentos descritos pelas notações Θ, Ω e $O(n)$ estão relacionados conforme o Teorema 1.

Teorema 1 *$f(n) = \Theta(g(n))$ se e somente se, para duas funções quaisquer $f(n)$ e $g(n)$, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.*

A aplicação das notações de complexidade nas expressões de custo dos algoritmos de ordenação analisados permite afirmar que:

- para o algoritmo de ordenação por inserção, o melhor caso tem custo $\Omega(n)$, o pior caso $O(n^2)$, e o custo do caso médio é de difícil cálculo, e, com certeza, não é $\Theta(n^2)$.
- para o algoritmo de ordenação por seleção, o melhor caso tem custo $\Omega(n^2)$, o pior caso, $O(n^2)$, e o caso médio, $\Theta(n^2)$.

Operações com Aproximações

Durante o processo de determinação do custos de algoritmos, as notações de complexidade assintótica ajudam a simplificar os cálculos, mas é preciso cuidado ao operar as expressões de custos obtidas, haja vista que o símbolo "=", que ocorrem em $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = o(g(n))$ e $f(n) = \omega(g(n))$, tem um significado especial de pertinência a conjunto, que não é seu significado usual na álgebra tradicional. Por exemplo, se dado que $f(n) = O(g(n))$, não faz sentido escrever a comutação $O(g(n)) = f(n)$.

Entretanto, algumas das operações tradicionais da *igualdade*, como transitividade, reflexidade e simetria, são preservadas.

Transitividade

1. $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
2. $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
3. $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
4. $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
5. $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Reflexividade

1. $f(n) = \Theta(f(n))$
2. $f(n) = O(f(n))$
3. $f(n) = \Omega(f(n))$

Simetria

1. $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$

Simetria de Transposição

1. $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$
2. $f(n) = o(g(n))$ se e somente se $g(n) = \omega(f(n))$

10 Fórmulas Importantes

Os seguintes somatórios e produtos são frequentemente encontrados no processo de cálculo da função de custo de comandos de repetição de muitos algoritmos.

Para facilitar sua resolução, apresentamos suas fórmulas de avaliação, nas quais n, a, b, c, i, a_j $\forall j \geq 1$, k, x e q são valores inteiros

positivos, x, y, z são números reais positivos, e lg, logaritmo na base 2.

Logaritmos

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b(x^y) = y \log_b x$
- $\log_b x = (\log_c x)(\log_b c)$
- $y^{\log_c x} = x^{\log_c y}$
- $b^{\log_b x} = x$

Produtos

- Produto dos n termos $a_1, a_2, \cdots, a_n$ de uma Progressão Geométrica de razão q:
  $P_n = \sqrt{a_1^n \times a_n^n}$ ou $P_n = a_1^{n+1} \times q^{\frac{n+1}{2} \times n}$

Somatórios

- Soma dos n termos $a_1, a_2, \cdots, a_n$ de uma Progressão Aritmética:
  $$\sum_{i=1}^{n} a_i = \frac{a_1 + a_n}{2} \times n$$

- Soma dos n termos $a_1, a_2, \cdots, a_n$ de uma Progressão Geométrica de razão q:
  $$\sum_{i=1}^{n} a_i = \frac{a_1(q^n - 1)}{q - 1} \quad (q > 1)$$
- $$\sum_{i=1}^{n-1}(n - i) = \frac{n(n-1)}{2}$$
- $$\sum_{i=1}^{n}(n - i) = \frac{n(n-1)}{2}$$
- $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
- $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
- $$\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k+1} + \text{termo de menor ordem}$$
- $$\sum_{i=0}^{n} ax^i = a \times \frac{1 - x^{n+1}}{1 - x} \qquad (x > 1)$$
- $$\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$$
- $$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \qquad (a > 1)$$
- $$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

- $\sum_{i=0}^{n-1} \frac{i}{2^i} = 2 - \frac{n+1}{2^{n-1}}$
- $\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{2n-1}{2^{n-1}}$
- $\sum_{i=0}^{n-1} i2^i = 2 + (n-2)2^n$
- $\sum_{i=0}^{n} ix^i = \frac{x - nx^n + (n-1)x^{n+1}}{(1-x)^2} \qquad (x > 1)$
- $\sum_{i=1}^{n} \lg i = \lg n!$

11 Notas Bibliográficas

A tradicional referência bibliográfica sobre complexidade de algoritmos é a série clássica de livros de Donald Knuth, *The Art of Computer Programming* [3, 4], que organizou didaticamente o conhecimento dessa área e formalizou a notação que é usada desde então.

O clássico *The Design and Analysis of Computer Algorithms* [1] de autoria de Alfred Aho, John Hopcroft e Jeffrey D. Ullman é uma obra prima que aborda a análise de algoritmos de forma bastante completa e profunda.

O livro *Algoritmos* [8] de Thomas H. Cormen et al. apresenta uma abordagem moderna e bastante extensa, sendo de leitura obrigatória de especialistas.

Os livros *Projeto de Algoritmos com Implementações em Pascal, C, Java e C++* [5, 6] de Nivio Ziviani oferecem uma

introdução do tema de fácil leitura.

Referências

[1] Alfred V, Aho, John E. Hopcroft e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] Alfred V. Aho, J.E. Hopcroft e J.D. Ullman, J.D. *Data Structure and Algorithms*. Addison-Wesley, 1983.

[3] Donald Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Second Edition, 1973.

[4] Donald Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, Second Edition, 1973.

[5] Nivio Ziviani. *Projeto de Algoritmos com Implementação em Pascal e C*. Editora Thompson, 2004. (Capítulos 1,3 e 5).

[6] Nivio Ziviani. *Projeto de Algoritmos com Implementações em Java e C++*. Editora Thompson, 2007.

[7] Roberto S. Bigonha e Mariza A. S. Bigonha. *Estruturas de Dados Fundamentais*. Notas de Aula.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Algoritmos: Teoria e Prática*. Editora Campus, 2002.