Complexidade assintótica de programas

Técnicas de análise de algoritmos são consideradas partes integrantes do processo moderno de resolver problemas, permitindo escolher, de forma racional, um dentre vários algoritmos disponíveis para uma mesma aplicação. Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado para resolver o problema em questão.

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

Para medir o custo de execução de um algoritmo é comum definir uma **função de custo** ou **função de complexidade** f, onde $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n. Se $f(n)$ é uma medida da quantidade do tempo necessário para executar um algoritmo em um problema de tamanho n, então f é chamada **função de complexidade de tempo** do algoritmo. Se $f(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo de tamanho n, então f é chamada **função de complexidade de espaço** do algoritmo. A não ser que haja uma referência explícita, f denotará uma função de complexidade de tempo daqui para a frente. É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados. Por isso, é comum considerar-se o tempo de execução de um programa como uma função do tamanho da entrada. Entretanto, para alguns algoritmos, o custo de execução é uma função de entrada particular dos dados, não apenas do tamanho da entrada como para um algoritmo de ordenação, pois se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Temos então de distinguir três cenários: melhor caso, pior caso e caso médio. O **melhor caso** corresponde ao menor tempo de execução sobre todas as possíveis entradas de tamanho n. O **pior caso** corresponde ao maior tempo de execução sobre todas as entradas de tamanho n. Se f é uma função de complexidade, baseada na análise de pior caso, então o custo de aplicar o algoritmo nunca é maior do que $f(n)$.

O **caso médio** corresponde à média dos tempos de execução de todas as entradas de tamanho n. Na análise do caso médio, uma distribuição de probabilidades sobre o conjunto de tamanho n é suposta, e o custo médio é obtido com base nesta distribuição. Por essa razão, a análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso. É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Entretanto, na prática isso nem sempre é verdade.

Considere o problema de acessar os registros de um arquivo através do algoritmo de pesquisa seqüencial. Seja f uma função de complexidade tal que $f(n)$ é o número de vezes que a chave de consulta é comparada com a chave de cada registro. Os casos a considerar são:

$$\begin{array}{ll} \text{melhor caso} & : f(n) = 1 \\ \text{pior caso} & : f(n) = n \\ \text{caso médio} & : f(n) = (n+1)/2 \end{array}$$

O melhor caso ocorre quando o registro procurado é o primeiro consultado. O pior caso ocorre

quando o registro procurado é o último consultado, ou então não está presente no arquivo; para tal, é necessário realizar n comparações (considerando uma seqüência não ordenada). Para o caso médio, considere toda pesquisa com sucesso. Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro são necessárias i comparações, então $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \ldots + n \times p_n$.

Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i. Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n$, $1 \leq i \leq n$. Neste caso $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$. A análise do caso esperado para a situação descrita revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

A ordem de crescimento do tempo de execução de um algoritmo fornece uma caracterização simples da eficiência do algoritmo, e também nos permite comparar o desempenho relativo de algoritmos alternativos. Por exemplo, uma vez que o tamanho da entrada n se torna grande o suficiente, um algoritmo com tempo de execução de pior caso da ordem de $n \log n$, vence um algoritmo para o mesmo problema cujo tempo de execução no pior caso é da ordem de n^2. Embora às vezes seja possível determinar o tempo exato de execução de um algoritmo, a precisão extra, em geral, não vale o esforço de calculá-la. Para entradas grandes o bastante, as constantes multiplicativas e os termos de mais baixa ordem de um tempo de execução exato são dominados pelos efeitos do próprio tamanho da entrada.

Quando observamos tamanhos de entrada grandes o suficiente para tornar relevante apenas a ordem de crescimento do tempo de execução, estamos estudando a eficiência assintótica dos algoritmos. Ou seja, estamos preocupados com a maneira como o tempo de execução de um algoritmo aumenta com o tamanho da entrada no limite, à medida que o tamanho da entrada aumenta indefinidamente (sem limitação). Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.

A seguir, discutiremos a análise assintótica de algoritmos e, então, faremos veremos o comportamento de funções que surgem comumente na análise de algoritmos.

1. Comportamento Assintótico de Funções

Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os algoritmos ineficientes. Em outras palavras, a escolha do algoritmo não é um problema crítico para problemas de tamanho pequeno. Logo, a análise de algoritmos é realizada para valores grandes de n. Para tal, considera-se o comportamento assintótico das funções de custo. O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.

A análise de um algoritmo geralmente conta com apenas algumas operações elementares e, em muitos casos, somente uma operação elementar. A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada. A seguinte definição relaciona o comportamento assintótico de duas funções distintas.

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

O significado da definição acima pode ser expresso em termos gráficos, conforme ilustra a Figura 1.

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

Knuth sugeriu uma notação para dominação assintótica. Para expressar que $f(n)$ domina assintotica-

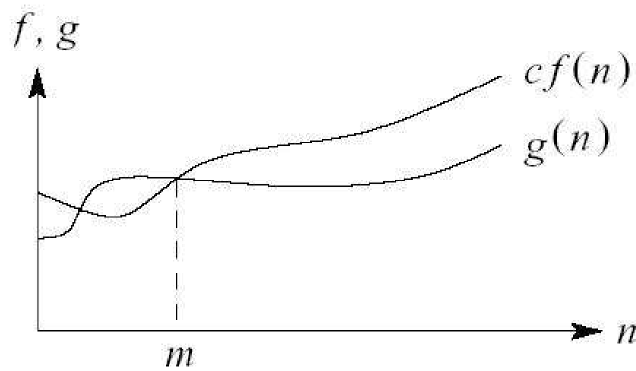


Figura 1. Dominação assintótica de $f(n)$ sobre $g(n)$

mente $g(n)$ escrevemos $g(n) = O(f(n))$, onde se lê $g(n)$ é da ordem no máximo $f(n)$. Por exemplo, quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, isto significa que existem constantes c e m tais que, para valores de n maiores ou iguais a m, $T(n) \leq cn^2$. A Figura 1 mostra um exemplo gráfico de dominação assintótica que ilustra a notação O. O valor da constante m mostrado é o menor valor possível, mas qualquer valor maior também é válido.

As funções de complexidade de tempo são definidas sobre os inteiros não negativos, ainda que possam ser não inteiros. A definição abaixo formaliza a notação de Knuth:

Definição notação O: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq cf(n)$, para todo $n \geq m$. **Exemplo:** Seja $g(n) = (n + 1)^2$. Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$. Isto porque $(n + 1)^2 \leq 4n^2$ para $n \geq 1$.

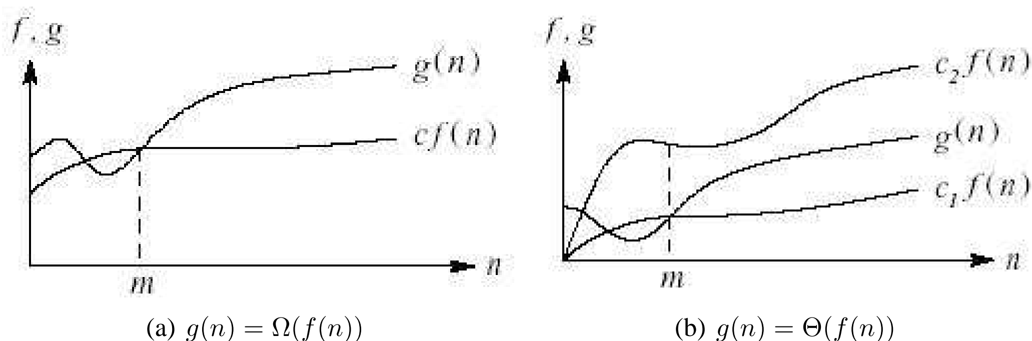
Exemplo: A regra da soma $O(f(n)) + O(g(n))$ pode ser usada para calcular o tempo de execução de uma sequência de trechos de programas. Suponha três trechos de programas cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$. O tempo de execução dos dois primeiros trechos é $O(\max(n, n^2))$, que é $O(n^2)$. O tempo de execução de todos os três trechos é então $O(\max(n^2, n \log n))$, que é $O(n^2)$.

Dizer que $g(n)$ é $O(f(n))$ significa que $f(n)$ é um limite superior para a taxa de crescimento de $g(n)$. A definição abaixo especifica um limite inferior para $g(n)$.

Definição notação Ω: Uma função $g(n)$ é $\Omega(f(n))$ se existirem duas constantes c e m tais que $g(n) \geq cf(n)$, para todo $n \geq m$. **Exemplo:** Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(f(n))$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$. A Figura 2 (a) mostra intuitivamente o significado da notação Ω. Para todos os valores que estão à direita de m, o valor de $g(n)$ está sobre ou acima do valor de $cf(n)$.

Definição notação Θ: Uma função $g(n)$ é $\Theta(f(n))$ se existirem constantes positivas c_1, c_2 e m tais que $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$. A Figura 2 (b) mostra intuitivamente o significado da notação Θ. Dizemos que $g(n) = \Theta(f(n))$ se existirem constantes c_1, c_2 e m tais que, para todo $n \geq m$, o valor de $g(n)$ está acima de $c_1 f(n)$ e abaixo de $c_2 f(n)$. Em outras palavras, para todo $n \geq m$, a função $g(n)$ é igual a $f(n)$ a menos de uma constante. Neste caso, $f(n)$ é um **limite assintótico firme**.

Exemplo: Seja $g(n) = n^2/3 - 2n$. Vamos mostrar que $g(n) = \Theta(n^2)$. Para isso, temos de obter constantes c_1, c_2 e m tais que $c_1 n^2 \leq \frac{1}{3}n^2 - 2n \leq c_2 n^2$ para todo $n \geq m$. Dividindo por n^2 leva a $c_1 \leq \frac{1}{3} - \frac{2}{n} \leq c_2$. O lado direito da desigualdade será sempre válido para qualquer valor de $n \geq 1$ quando escolhemos $c_2 \geq 1/3$. Da mesma forma, escolhendo $c_1 \leq 1/21$, o lado esquerdo



(a) $g(n) = \Omega(f(n))$ (b) $g(n) = \Theta(f(n))$

Figura 2. Exemplo gráfico para as notações Ω e Θ

da desigualdade será válido para qualquer valor de $n \geq 7$. Logo, escolhendo $c_1 = 1/21$, $c_2 = 1/3$ e $m = 7$, é possível verificar que $n^2/3 - 2n = \Theta(n^2)$. Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

O limite assintótico superior definido pela notação O pode ser assintoticamente firme ou não. Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente firme, mas o limite $2n = O(n^2)$ não é assintoticamente firme. A notação o apresentada a seguir é usada para definir um limite superior que não é assintoticamente firme.

Notação o: O limite assintótico superior fornecido pela notação O pode ser ou não assintoticamente restrito. Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente restrito, mas o limite $2n = O(n^2)$ não é, portanto, $2n = o(n^2)$. Usamos a notação o para denotar um limite superior que não é assintoticamente restrito. Formalmente, uma função $g(n)$ é $o(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq g(n) < cf(n)$ para todo $n \geq m$.

Notação ω: Usamos a notação ω para denotar um limite inferior que não é assintoticamente restrito. Por exemplo, o limite $\frac{n^2}{2} = \Omega(n^2)$ é assintoticamente restrito, mas o limite $\frac{n^2}{2} = \Omega(n)$ não é, portanto, $\frac{n^2}{2} = \omega(n)$. Mais formalmente, uma função $g(n)$ é $\omega(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq cf(n) \leq g(n)$ para todo $n \geq m$.

Para fins de comparação de funções, muitas das propriedades relacionais de números reais também se aplicam a comparações assintóticas. No caso das propriedades seguintes, suponha que $f(n)$ e $g(n)$ sejam assintoticamente positivas.

1. **Transitividade (válido também para O, Ω, o e ω):**
   $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implicam $f(n) = \Theta(h(n))$.
2. **Reflexividade (válido também para O e Ω):**
   $f(n) = \Theta(f(n))$
3. **Simetria:**
   $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$
4. **Simetria de transposição:**
   $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$
   $f(n) = o(g(n))$ se e somente se $g(n) = \omega(f(n))$

Regras de Simplificação

1. Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$, então $f_1(n)+f_2(n)$ está em $O(\max(g_1(n), g_2(n)))$.
2. Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$, então $f_1(n)f_2(n)$ está em $O(g_1(n)g_2(n))$.

A seguir, descreveremos e compararemos as funções de complexidade de algumas classes de problemas que surgem comumente na análise de algoritmos.

2. Classes de Comportamento Assintótico

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

Programas podem ser avaliados por meio da comparação de suas funções de complexidade, negligenciando as constantes de proporcionalidade. Um programa com tempo de execução $O(n)$, por exemplo, é melhor que um programa com tempo de execução $O(n^2)$. Entretanto, as constantes de proporcionalidade em cada caso podem alterar esta consideração. Por exemplo, é possível que um programa leve $100n$ unidades de tempo para ser executado enquanto outro leve $2n^2$ unidades de tempo? Qual dos dois programas é melhor?

A resposta a essa pergunta depende do tamanho do problema a ser executado. Para problemas de tamanho $n < 50$, o programa com tempo de execução $2n^2$ é melhor do que o programa com tempo de execução $100n$. Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$. Entretanto, quando n cresce, o programa com tempo $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

A maioria dos algoritmos possui um parâmetro que afeta o tempo de execução de forma mais significativa, usualmente o número de itens a ser processado. Este parâmetro pode ser o número de registros de um arquivo a ser ordenado, ou o número de nós de um grafo. As principais **classes de problemas** possuem as **funções de complexidade** descritas abaixo:

1. $f(n) = O(1)$. Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**. O uso do algoritmo independe do tamanho de n. Neste caso, as instruções do algoritmo são executadas um número fixo de vezes.
2. $f(n) = O(\log n)$. Um algoritmo de complexidade $O(\log n)$ é dito ter **complexidade logarítmica**. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como menor do que uma constante grande. Quando n é mil e a base do logaritmo é 2, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$. Para dobrar o valor de $\log n$ temos de considerar o quadrado de n. A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.
3. $f(n) = O(n)$. Um algoritmo de complexidade $O(n)$ é dito ter **complexidade linear**. Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem de processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho, o tempo de execução dobra.
4. $f(n) = O(n \log n)$. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independente-

mente e depois ajuntando as soluções (algoritmos com essa característica obedecem ao paradigma de "dividir para conquistar"). Quando n é 1 milhão e a base do logaritmo é 2, $n \log_2 n$ é cerca de 20 milhões. Quando n é 2 milhões, $n \log_2 n$ é cerca de 20 milhões. Quando n é 2 milhões, $n \log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

5. $f(n) = O(n^2)$. Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**. Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro de outro. Quando n é mil, o número de operações é da ordem de 1 milhão. Sempre que n dobra, o tempo de execução é multiplicado por 4. Algoritmos deste tipo são úteis para resolver problemas de tamanho relativamente pequenos.
6. $f(n) = O(n^3)$. Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**. Algoritmos desta ordem de complexidade são úteis apenas para resolver pequenos problemas. Quando n é 100, o número de operações é da ordem de 1 milhão. Sempre que n dobra, o tempo de execução fica multiplicado por 8.
7. $f(n) = O(2^n)$. Um algoritmo de complexidade $O(2^n)$ é dito ter **complexidade exponencial**. Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa a **força bruta** para resolvê-los. Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo de execução fica elevado ao quadrado.
8. $f(n) = O(n!)$. Um algoritmo de complexidade $O(n!)$ é também dito ter complexidade exponencial, apesar de que a **complexidade fatorial** $O(n!)$ tem comportamento muito pior do que a complexidade $O(2^n)$. Algoritmos desta ordem de complexidade geralmente ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los. Quando n é 20, $20! = 2432902008176640000$, um número com 19 dígitos.

Um algoritmo cuja função de complexidade é $O(c^n)$, $c > 1$, é chamado de **algoritmo exponencial** no tempo de execução. Um algoritmo cuja função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio, é chamado de **algoritmo polinomial** no tempo de execução. A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema cresce. Esta é a razão pela qual algoritmos polinomiais são muito mais úteis na prática do que algoritmos exponenciais.

Os algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva, enquanto algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema. Um problema é considerado **intratável** se ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo, enquanto um problema é considerado bem resolvido quando existe um algoritmo polinomial para resolvê-lo.

Entretanto, a distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções. Por exemplo, um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20. Da mesma forma, existem algoritmos exponenciais que são muito úteis na prática. Por exemplo, o algoritmo Simplex, frequentemente utilizado para programação linear, possui complexidade de tempo exponencial para o pior caso.

Infelizmente, porém, exemplos como o algoritmo Simplex não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis. Considere, como exemplo, o seguinte problema: um **caixeiro viajante** deseja visitar n cidade de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez. Supondo que sempre exista uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota que o caixeiro viajante possa utilizar na sua viagem. Um algoritmo simples para esse problema seria verificar todas

as rotas e escolher a menor delas. Como existem $(n-1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, então o número total de adições é $n!$. Considerando um computador capaz de executar 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^45 séculos somente para executar as adições.

A seguir, faremos a análise assintótica de dois algoritmos para resolver o problema da ordenação: ordenação por intercalação (*Mergesort*) e ordenação por inserção. Então compararemos assintoticamente os dois algoritmos.

3. Análise de Algoritmos Recursivos

Muitos algoritmos úteis são recursivos em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Em geral, esses algoritmos seguem uma abordagem de *dividir e conquistar* que envolve três passos em cada nível da recursão:

1. **Dividir** o problema em um determinado número de subproblemas;
2. **Conquistar** os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.
3. **Combinar** as soluções dos subproblemas para formar a solução para o problema original.

Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução frequentemente pode ser descrito por uma *equação de recorrência* ou *recorrência*, que descreve o tempo de execução global sobre um problema de tamanho n em termos do tempo de execução sobre entradas menores. Então, podemos usar ferramentas matemáticas para resolver a recorrência e estabelecer limites sobre o desempenho do algoritmo.

Uma recorrência para o tempo de execução de um algoritmo de dividir e conquistar se baseia nos três passos do paradigma básico. Consideramos $T(n)$ o tempo de execução sobre um problema de tamanho n. Se o tamanho do problema for pequeno o bastante, digamos $n \leq c$ para alguma constante c, a solução direta demorará um tempo constante, que consideraremos $\Theta(1)$. Vamos supor que o problema seja dividido em a subproblemas, cada um dos quais com $1/b$ do tamanho do problema original. Se levarmos o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções dadas aos subproblemas na solução para o problema original, obteremos a recorrência

$$T(n) = \begin{cases} \Theta(1), \text{se } n \leq c, \\ aT(n/b) + D(n) + C(n), \text{caso contrário.} \end{cases} \tag{1}$$

Por exemplo, considere o problema de ordenação que consiste em rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. Um dos algoritmos que resolvem o problema de ordenação é o algoritmo de ordenação por intercalação (*Mergesort*). O *Mergesort* utiliza o paradigma de divisão e conquista como segue: dado um vetor de tamanho n, divida recursivamente o vetor a ser ordenado em dois vetores até obter n vetores de um único elemento. Então, intercale dois vetores de um elemento, formando um vetor ordenado de dois elementos. Repita este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado. Veja um exemplo na Figura 3. e o pseudocódigo na Figura 4.

Para a análise do *Mergesort*, considere a comparação de chaves a operação relevante para determinar o tempo de execução do algoritmo. Observe que para o *Mergesort*, temos $a = b = 2$ em 1 pois cada instância é dividida em dois subproblemas cujo tamanho é a metade do problema original. Para

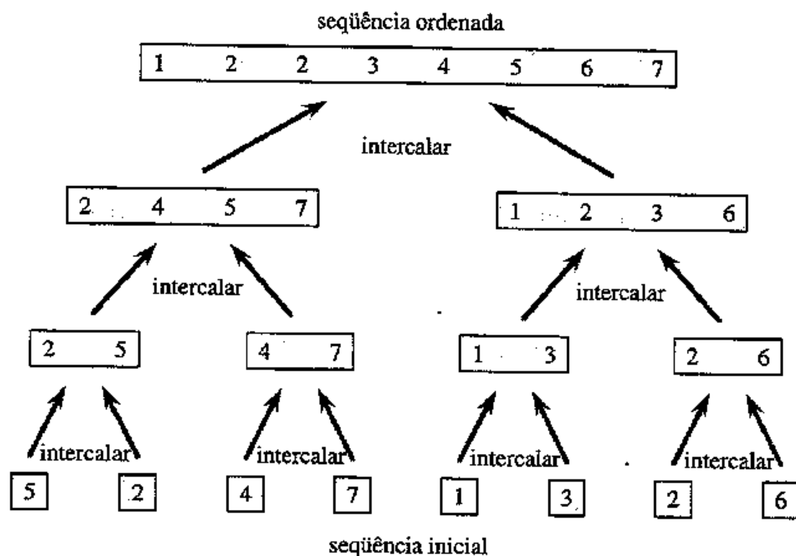


Figura 3. O *Mergesort* aplicado sobre o arranjo $A = (5, 2, 4, 7, 1, 3, 2, 6)$. Os comprimentos das sequências ordenadas que estão sendo intercaladas aumentam com a progressão do algoritmo da parte inferior até a parte superior.

```
1: procedure MergeSort(var A: array[1..n] of integer; i, j: integer);
2:     se i < j então
3:         m := (i + j − 1)/2;
4:         MergeSort(A,i,m);
5:         MergeSort(A,m + 1,j);
6:         Merge(A,i,m,j);
```

Figura 4. Algoritmo de ordenação por intercalação.

facilitar a análise, vamos supor que o tamanho do problema é uma potência de dois (o que não afeta a ordem de crescimento da solução para a recorrência). Podemos desmembrar o tempo de execução como a seguir:

- **Dividir:** simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portanto, $D(n) = \Theta(1)$.
- **Conquistar:** resolvemos recursivamente dois subproblemas; cada um tem o tamanho $n/2$ e contribui com $T(n/2)$ para o tempo da execução.
- **Combinar:** o procedimento *MERGE* em um subarranjo de n elementos leva o tempo $\Theta(n)$ para intercalar os elementos. Assim, $C(n) = \Theta(n)$.

Desta forma, a recorrência para o tempo de execução $T(n)$ do pior caso do *Mergesort* é:

$$T(n) = \begin{cases} \Theta(1), \text{se } n = 1, \\ 2T(n/2) + \Theta(n) \text{ se } n > 1. \end{cases} \quad (2)$$

O Teorema Mestre, uma das técnicas de resolução de recorrências, resolve recorrências no formato $T(n) = aT(n/b) + f(n)$. Este formato descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas, cada um do tamanho n/b, onde a e b são constantes positivas. Os a subproblemas são resolvidos recursivamente, cada um no tempo $T(n/b)$. O custo de dividir o problema e combinar os resultados dos subproblemas é descrito pela função $f(n)$.

A recorrência que surge do Mergesort tem $a = 2$, $b = 2$ e $f(n) = \Theta(n)$. De acordo com o método, se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$. Como $f(n) = \Theta(n)$, a resolução desta recorrência é $\Theta(n \log_2 n)$ o que faz do *Mergesort* um algoritmo ótimo já que foi provado que o limite inferior do problema de ordenação por comparação é $\Omega(n \log n)$. Tecnicamente, na realidade a recorrência do Mergesort não está bem definida, porque n/b poderia não ser um inteiro. Porém, a substituição de cada um dos a termos $T(n/b)$ por $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$ não afeta o comportamento assintótico da recorrência. Por esta razão omitimos as funções piso e teto.

Considere agora outro algoritmo para o problema de ordenação conhecido como *ordenação por inserção*: tendo ordenado o subarranjo $A[1..j-1]$, insira o elemento isolado $A[j]$ em seu lugar apropriado, formando o subarranjo ordenado $A[1..j]$ (veja o pseudo-código na Figura 5).

```
1: procedure Insertion-Sort(A);
2:     for j := 2 to comprimento(A) do
3:         chave := A[j];
4:         Comentário: o código abaixo insere A[j] na sequência ordenada A[1..j − 1]
5:         i := j − 1;
6:         while i > 0 and A[i] > chave do
7:             A[i + 1] := A[i];
8:             i := i − 1;
9:         A[i + 1] := chave;
```

Figura 5. Algoritmo de ordenação por inserção.

Seja $T(n)$ a função que conta o número de comparações de chaves e n o número de chaves a serem ordenadas. No melhor caso, quando o arranjo já está ordenado, o laço da linha 6 é executado uma única vez quando então o algoritmo descobre que $A[i] \leq chave$. Como o laço da linha 2 executa $n-1$ vezes, a complexidade do algoritmo no melhor caso é $T(n) = n - 1 = \Theta(n)$.

Considere agora o pior caso do algoritmo que acontece quando o arranjo está ordenado na ordem inversa. Neste caso, devemos então comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1..j-1]$ e, assim, o número de vezes que o laço da linha 6 é executado é $2 + 3 + \ldots + n$ vezes que pode ser expressa por uma progressão aritmética:

$$2 + 3 + \ldots + n = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} = \Theta(n^2).$$

Portanto, o tempo de execução no pior caso da ordenação por inserção é uma função quadrática de n. Logo, o *Mergesort* é assintoticamente mais eficiente que a ordenação por inserção. Entretanto, vale ressaltar que a ordenação por inserção é mais interessante para arquivos com menos de 20 elementos, podendo ser mesmo mais eficiente do que o *Mergesort*. Além disso, para arquivos já ordenados, o tempo de execução é linear.

4. Conclusão

Parte fundamental da complexidade de algoritmos, a análise de algoritmos é fundamental para ajudar no projeto de algoritmos eficientes, seja na escolha de um algoritmo entre outros já existentes, seja no desenvolvimento de um algoritmo para um problema ainda não resolvido. Por isso, nós discutimos, ainda que brevemente, a notação assintótica e alguns recursos úteis frequentemente empregados na análise de algoritmos.

Existem ainda outros recursos utilizados na análise da complexidade de um algoritmo. Por exemplo, se o algoritmo é recursivo, frequentemente, pode ser expresso como uma recorrência que pode ser resolvida por técnicas de análise de algoritmos tais como o Método Mestre, a Árvore de Recursão ou o Método por Substituição no qual supomos um limite e utilizamos indução matemática para descobrir se o limite está correto.

Enquanto a teoria de análise de algoritmos estuda a análise da complexidade de algoritmos, a teoria da complexidade estuda a classificação de problemas com base na complexidade dos algoritmos que os resolvam. Neste contexto, alguns problemas classificados como NP-difíceis - para os quais não se conhece qualquer algoritmo de tempo polinomial - necessitam que outras alternativas de projeto de algoritmos sejam empregadas tais como algoritmos heurísticos, algoritmos randomizados, algoritmos aproximativos, dentre outras opções.

5. Bibliografia Utilizada

Esse texto é um resumo sobre o assunto tirados dos livros:

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.
SZWARCFITER, J. L. e MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994.
ZIVIANI, N. Projeto de Algoritmos: com implementações em Pascal e C, 2ª edição, Cengage Learning, 2009.