COMPLEXIDADE DE ALGORITMOS

Complexidade computacional e assintótica

O mesmo problema pode freqüentemente ser resolvido com algoritmos que diferem em eficiência. As diferenças entre os algoritmos podem ser irrelevantes para processar um pequeno número de itens de dados, mas cresce proporcionalmente com a quantidade de dados. Com a finalidade de comparar a eficiência de algoritmos, uma medida do grau de dificuldade de um algoritmo, chamada de "*complexidade computacional*" foi desenvolvida por Juris Hartmanis e Richard E. Stearns.

A complexidade computacional indica quanto esforço é necessário para se aplicar um algoritmo, ou quão custoso ele é.

É fundamental escrever programas corretos para algoritmos de nosso interesse, bem como demonstrar que eles calculam corretamente as funções desejadas. No entanto, não basta garantir que um algoritmo fornece as respostas corretas em um número finito de passos.

Muitas vezes um programa correto se torna inútil para uma entrada de dados moderadamente grande, pois o tempo de execução é inviável. Sendo assim, é importante considerar a eficiência de nossos algoritmos. Essas questões são tratadas pela *Teoria da Complexidade de Algoritmos*, área da computação que visa a determinar a complexidade (custo) de um algoritmo, o que torna possível:

- Comparar algoritmos
  Como existem muitos algoritmos que resolvem uma mesma classe de problemas, através dessa medida de complexidade podemos compará-los entre si.

- Determinar se um algoritmo é "*ótimo*"
  Existem problemas que têm um "limite inferior" para complexidade dos algoritmos que o resolvem. Esse limite inferior é determinado matematicamente, através da modelagem e estudo do problema. Um algoritmo é dito ótimo quando a sua complexidade é igual à "complexidade inferior limite" do problema.

  Desta forma, a teoria da complexidade classifica um algoritmo de acordo com seu grau de complexidade (custo). Complexidade neste contexto se refere à medida da quantidade de recursos consumidos durante a execução de um algoritmo. Quatro recursos básicos podem ser identificados:

  - número de operações básicas executadas pelo algoritmo;
  - tempo de execução gasto;
  - quantidade de memória necessária para execução do algoritmo;
  - recursos de hardware necessários.

A Teoria da Complexidade quantitativamente classifica os problemas em tratáveis baseando-se nas medidas de complexidade citadas. Admitindo-se que o número de operações seja tomado como medida de complexidade, os algoritmos são classificados de acordo com o número de operações executadas, como por exemplo uma função de valor absoluto, o incremento de uma variável.

A complexidade de um algoritmo pode ser analisada sob duas métricas: tempo e espaço. O tempo diz respeito ao tempo necessário à execução do algoritmo, e espaço, mostra o comportamento do algoritmo quanto à necessidade de memória principal. A métrica tempo é a mais utilizada.

O exemplo a seguir mostra como comparar algoritmos entre si, quanto ao seu custo de tempo. Por exemplo, suponha uma corda de tamanho T e que deve-se "organizá-la" de tal forma que ocupe o menor espaço possível. Vamos analisar três métodos:

1. Enrolar no braço
   Segurando uma das extremidades com a mão, e dando a volta sob o cotovelo, enrolamos a corda várias vezes, sempre mantendo o mesmo comprimento do laço;

2. Enrolar sobre si
   Usando algum apoio fixo (por exemplo, uma maçaneta), enrolamos a corda sobre esse ponto fixo, aumentando gradativamente o tamanho do laço;

3. Dobrar sucessivamente
   Dobre a corda ao meio, unindo as duas extremidades. Repita esse processo até que não seja mais possível dobrar.

Após um estudo matemático dos métodos, concluí-se que o método 3 é o mais eficiente, pois resolve o problema em menos passos, ou seja, o seu custo é menor.

Para avaliar a eficiência de um algoritmo, unidades de tempo real, tais como microssegundos e nanossegundos, não devem ser usadas. Ao contrário, unidades lógicas que expressam uma relação entre o tamanho *n* de um arquivo ou de uma matriz e, a quantidade de tempo *t* exigida para processar *s* dados precisam ser usadas.

Uma função que expressa a relação entre *n* e *t* usualmente é muito mais complexa, e o cálculo dessa função é importante somente em relação a grandes quantidades de dados; quaisquer termos que não modifiquem substancialmente a grandeza da função devem ser eliminados da função. A função resultante fornece somente uma medida aproximada da eficiência da função original. No entanto essa aproximação é suficientemente próxima do original, especialmente para uma função que processa grandes quantidades de dados. Essa medida de eficiência é chamada de *complexidade assintótica* e é usada quando se desprezam certos tempos de uma função para expressar a eficiência de um algoritmo, ou quando calcular uma função é difícil ou impossível e somente aproximações podem ser encontradas.

Para ilustrar o primeiro caso considere o seguinte exemplo:

$$f(n) = n^2 + 100n + log_{10}n + 1000$$

Para pequenos valores de *n*, o último termo, *1000*, é o maior. Quando *n* se iguala a *10*, o segundo (*100n*) e o último (*1000*) termos estão em igual condição com os outros, dando uma pequena contribuição ao valor da função. Quando *n* atinge o valor de *100*, o primeiro e o segundo termos dão a mesma contribuição ao resultado. Mas quando *n* se torna maior que *100*, a contribuição do segundo termo se torna menos significativa. Por isso, para valores maiores de *n*, devido ao crescimento quadrático do primeiro termo (n^2), o valor da função *f* depende principalmente do valor de seu primeiro termo, como demonstra a Tabela 1. Os outros valores podem ser desprezados.

Tabela 1 - Taxa de crescimento de todos os termos da função $f(n) = n^2 + 100n + log_{10}n + 1000$.

n	$f(n)$	n^2	$100n$	$log_{10}n$	1000
1	1.101	1	100	0	1000
10	2.101	100	1.000	1	1000
100	21.002	10.000	10.000	2	1000
1.000	1.101.003	1.000.000	100.000	3	1000
10.000	101.001.004	100.000.000	1.000.000	4	1000
100.000	10.010.001.005	10.000.000.000	10.000.000	5	1000

Ordem de Grandeza (O)

A complexidade de um algoritmo não é medida em termos reais (medidas empíricas), ou seja, não analisamos algoritmos, por exemplo, em termos de tempo real de execução (tempo de execução pela CPU — Unidade Central de Processamento). Esse tipo de avaliação não é adequado, pois em geral existem muitos fatores influenciando o resultado, tais como eficiência do computador (*hardware*) e carga do sistema operacional no momento das medições, entre outros. O ideal é avaliar o algoritmo independente da plataforma e da tecnologia utilizada.

A complexidade de um algoritmo é medida segundo um modelo matemático (analítico). Esse modelo supõe que o algoritmo vai trabalhar sobre uma "entrada" (massa de dados) de tamanho *n*, ou seja, existem *n* informações distintas a serem tratadas.

A análise de um algoritmo é realizada considerando dois fatores:

- Tempo de Execução
  A complexidade mais estudada em algoritmos é a complexidade de tempo. O modelo para determinação desse custo é baseado no custo de cada operação realizada, em função do tamanho da entrada (*n*). Em geral, não são consideradas todas as operações, levando-se em consideração apenas as de maior custo. Por exemplo, algoritmos de ordenação têm a sua complexidade de tempo determinada pelo número de comparações realizadas, em função de *n*;

- Espaço em Memória
  Determina o espaço utilizado pelo algoritmo. Em geral, quanto menor a complexidade de tempo de um algoritmo, maior, a sua complexidade de espaço.

A complexidade não é uma medida exata, pois na verdade ela representa uma "ordem de grandeza".

A notação mais usada para especificar a complexidade assintótica, isto é, para estimar a taxa de crescimento da função, é a notação *O-Grande*, introduzida em 1894 por Paul Bachmann.

Define-se então a ordem de complexidade de um algoritmo como sendo um limite superior de sua complexidade para uma entrada suficientemente grande. Essa ordem de grandeza é simbolizada pela letra *O*. Por exemplo:

Tabela 2 – Ordem de grandeza.

x	***o(x)***
2*n + 5	n
n	n
127*n*n + 457*n	n*n
n*n*n + 5	n*n*n

Essa ordem de grandeza mostra a complexidade quando o tamanho da entrada tende ao infinito. A ordem de grandeza *O* permite avaliar o algoritmo independentemente do tamanho real da entrada.

Todo problema algorítmico possui um limite inferior de complexidade, ou seja, a quantidade mínima de recursos necessária para resolver um problema. O limite superior é a quantidade de recursos que é suficiente para resolver um problema. Seja *g* o limite inferior de um problema, um "algoritmo ótimo" é aquele que possui complexidade *O(g)*. Na prática, são raros os casos em que temos esta situação ideal.

Para entender o que vem a ser ordem de magnitude, suponha-se que num dado programa exista a seguinte linha de comando: `x++`.

Deseja-se estimar a ordem de magnitude das 3 situações a seguir:

```
// exemplo 1
int main (){
    int x;
    x++;
}
```

No **exemplo 1** a freqüência do comando é igual a 1.

```
// exemplo 2
int main (){
    int x, n=10;
    for (int i= 1; i<=n; i++)
       x++;
}
```

No segundo exemplo (**exemplo 2**) a freqüência do comando é igual a n.

```
// exemplo 3
int main (){
    int x, n=10;
    for (int i= 1; i<=n; i++)
        for (int j= 1; j<=n; j++)
            x++;
}
```

No **exemplo 3** a freqüência do comando é igual a n^2.

Observa-se nos três exemplos anteriores que a ordem de magnitude varia de um exemplo para outro. No **exemplo 1** *O(n) = 1*, no **exemplo 2** *O(n) = n* e no **exemplo 3** *O(n)* = n^2. *O(n)* indica que a ordem de magnitude é proporcional ao valor de *n*.

Comportamento

A complexidade pode ser qualificada quanto ao seu comportamento como:

- Polinomial
  O(n) é polinomial, pois à medida que *n* aumenta, o fator que estiver sendo analisado (tempo ou espaço) aumenta linearmente.

- Exponencial
  Qualquer comportamento não-polinomial é exponencial, pois à medida que *n* aumenta, o fator que estiver sendo analisado aumenta exponencialmente.

Outros exemplos de ordens de magnitude:

Ordem constante	$O(n) = 1$
Ordem logarítmica	$O(n) = log_2\ n$
Ordem cúbica	$O(n) = n^3$

Em cada uma das ordens de magnitude apresentadas, o parâmetro *n* provê uma medida do tamanho do problema no sentido de que o tempo necessário para executá-lo, ou o espaço de armazenamento necessário, ou ambos, serão incrementados conforme o valor de *n* aumenta.

Problemas tratáveis são aqueles que tem *n* como um fator polinomial em seus limites superior e inferior. Exemplos: $O(log\ n)$, $O(n)$, $O(n\ log\ n)$, $O(log\ n^k)$. Problemas tratáveis são ditos computáveis pois podem ser resolvidos computacionalmente.

Problemas intratáveis são aqueles que tem *n* como um fator exponencial em seus limites superior e inferior. Exemplos: $O(2^n)$, $O(n!)$, $O(n^n)$. No ponto de vista prático esses problemas são considerados insolúveis via computação, já que os recursos necessários para resolver esse problema têm um custo altíssimo.

Comparação de Programas

Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade. Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$. Porém, as constantes de proporcionalidade podem alterar esta consideração.

Exemplo: Um programa leva *100n* unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?

Depende do tamanho do problema. Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo *100n*. Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$. Entretanto, quando *n* cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Principais Classes de Problemas

f(n) = O(1)

- Algoritmos de complexidade *O(1)* são ditos de **complexidade constante**.
- Uso do algoritmo independe de *n*.
- As instruções do algoritmo são executadas um número fixo de vezes.

f(n) = O(log n)

- Um algoritmo de complexidade *O(log n)* é dito ter **complexidade logarítmica**.
- Típico em algoritmos que transformam um problema em outros menores.
- Pode-se considerar o tempo de execução como menor que uma constante grande.

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.
- É a melhor situação possível para um algoritmo que tem de processar/produzir *n* elementos de entrada/saída.
- Cada vez que *n* dobra de tamanho, o tempo de execução dobra.

f(n) = O(n log n)

- Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.
- Quando *n* é 1 milhão, *nlog2n* é cerca de 20 milhões.
- Quando *n* é 2 milhões, *nlog2n* é cerca de 42 milhões, pouco mais do que o dobro.

f(n) = O(n^2)

- Um algoritmo de complexidade *O(n^2)* é dito ter **complexidade quadrática**.
- Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
- Quando *n* é mil, o número de operações é da ordem de 1 milhão.
- Sempre que *n* dobra, o tempo de execução é multiplicado por 4.
- Úteis para resolver problemas de tamanhos relativamente pequenos.

f(n) = O(n^3)

- Um algoritmo de complexidade *O(n^3)* é dito ter **complexidade cúbica**.
- Úteis apenas para resolver pequenos problemas.
- Quando *n* é 100, o número de operações é da ordem de 1 milhão.
- Sempre que *n* dobra, o tempo de execução fica multiplicado por 8.

f(n) = O(2^n)

- Um algoritmo de complexidade *O(2^n)* é dito ter **complexidade exponencial**.
- Geralmente não são úteis sob o ponto de vista prático.
- Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
- Quando *n* é 20, o tempo de execução é cerca de 1 milhão. Quando *n* dobra, o tempo fica elevado ao quadrado.

f(n) = O(n!)

- Um algoritmo de complexidade *O(n!)* é dito ter complexidade exponencial, apesar de *O(n!)* ter comportamento muito pior do que *O(2^n)*.
- Geralmente ocorrem quando se usa **força bruta** para na solução do problema.
- *n = 20* -> 20! = 2432902008176640000, um número com 19 dígitos.
- **n = 40** -> um número com 48 dígitos.

Exemplos de Complexidade

Os algoritmos podem ser classificados por suas complexidades de tempo e espaço. Sob esse aspecto, diversas classes de algoritmos podem ser distinguidas. Seus crescimentos estão também exibidos na Figura 1.

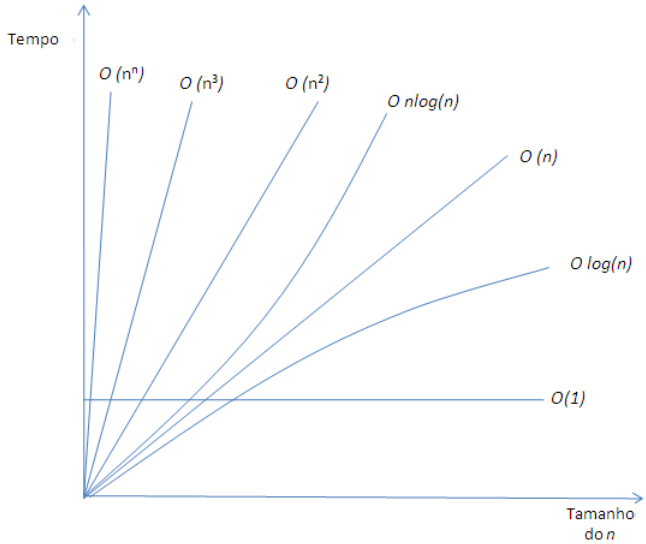


Figura 1 - Comportamento da ordem de complexidade em função do tamanho da entrada e do tempo.

Por exemplo, um algoritmo é chamado de *constante* se seu tempo de execução permanece o mesmo para qualquer número de elementos; é chamado de *quadrático* se seu tempo de execução é $O(n^2)$.

Tabela 3 - Comportamento do tempo em função do tamanho da entrada e da ordem de complexidade.

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

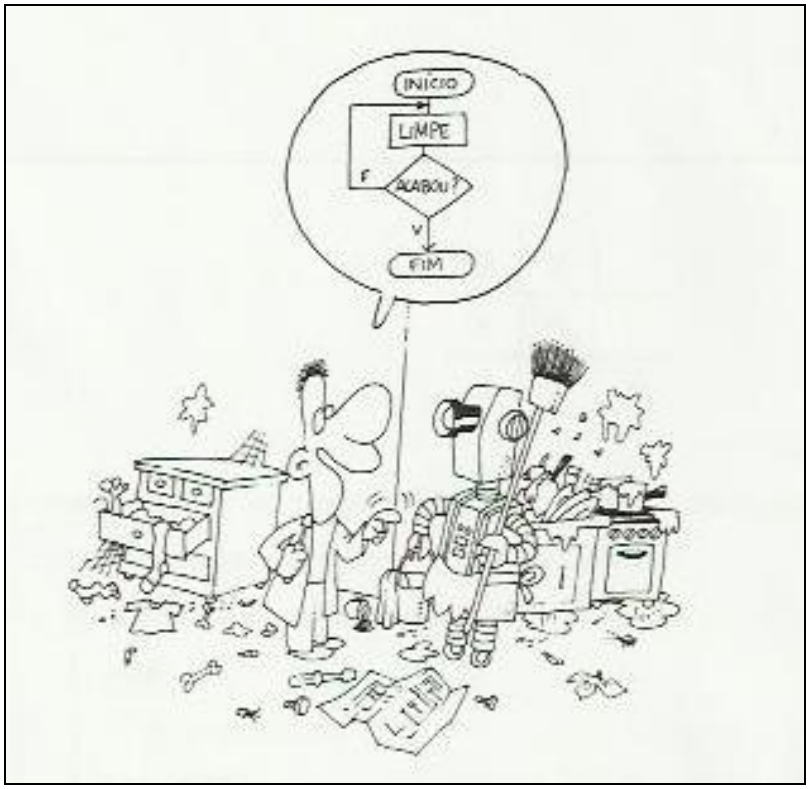
Alguns algoritmos mal projetados, ou algoritmos cuja complexidade não pode ser melhorada, não têm aplicações práticas nos computadores disponíveis. Para processar um milhão de itens com um algoritmo quadrático, mais de 11 dias são necessários, e, para um algoritmo cúbico, milhares de anos. Mesmo se um computador pudesse realizar uma operação por nanossegundos (um bilhão de operações por segundo), o algoritmo quadrático terminaria em 16,7 segundos, mas o algoritmo cúbico exigiria mais de 31 anos. Mesmo uma melhoria de 1.000 vezes na velocidade de execução teria muito pouco sentido prático para esse algoritmo. A Tabela 3 mostra o comportamento do tempo em função da dimensão da entrada e ordem de complexidade. A velocidade impressionante dos computadores é de

utilidade limitada se os programas que neles rodam usam algoritmos ineficientes.

Conclusão

A aplicação da análise de complexidade de algoritmos permite que muitas soluções ao serem propostas, para um determinado problema, sejam avaliadas em termos de convergência, tempo e espaço de memória. Todas estas métricas correspondem ao "custo" deste algoritmo.

Para problemas de pequena ordem, muitas vezes o tempo gasto na avaliação de complexidade não justifica o beneficio. Mas, em problemas de grandezas superiores, esta avaliação permite o uso de melhores soluções, quando não, da solução ótima. Por exemplo: métodos de criptografia e algoritmos de compressão de voz e imagem.



Exemplo**: Sequencia de Fibonacci**

Para projetar um algoritmo eficiente, é fundamental preocupar-se com a sua complexidade. Considere a seqüência de Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

A seqüência pode ser definida recursivamente:

$$F_n = \begin{cases} 0, \text{ se } n = 0 \\ 1, \text{ se } n = 1 \\ F_{n-1} + F_{n-2}, \text{ se } n > 1 \end{cases}$$

Dado o valor de n, deseja-se obter o n-ésimo elemento da seqüência de Fibonacci. Observe os dois algoritmos e analise sua complexidade.

Algoritmo 1

```
int fibo1(int n) {
  if (n == 0)
    return 0;
  else
    if (n == 1)
        return 1;
    else
      return fibo1(n - 1) + fibo1(n - 2);
}
```

Experimente rodar este algoritmo para n = 100. A complexidade é $O(2^n)$.
(Mesmo se uma operação levasse um picosegundo,
2100 operações levariam 3*1013 anos = 30.000.000.000.000 anos.)

Algoritmo 2

```
int fibo2 (int n) {
  int i, penultimo, ultimo, atual;
  if (n == 0)
    return 0;
  else
    if (n == 1)
      return 1;
    else {
      penultimo = 0;
      ultimo = 1;
      for (i = 2; i <= n; i++){
        atual = penultimo + ultimo;
        penultimo = ultimo;
        ultimo = atual;
      }
      return atual;
    }
}
```

A complexidade agora passou de $O(2^n)$ para O(n).