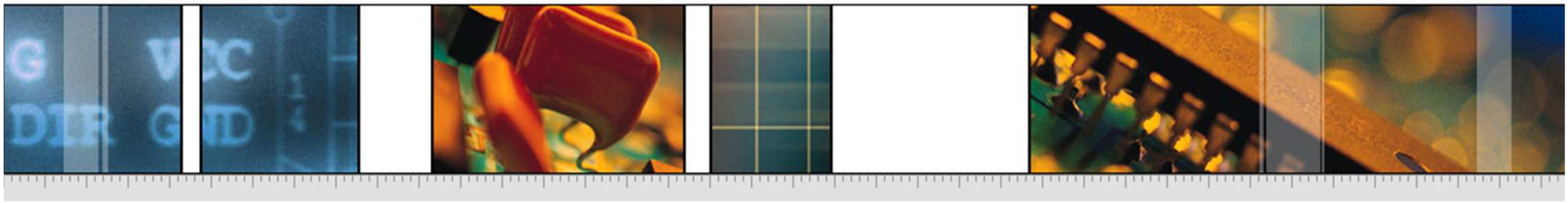


Análise de Complexidade de Algoritmos

mario alexandre gazziro

Definição

A complexidade de um algoritmo consiste na quantidade de "esforço computacional" necessária para sua execução.

Esse "esforço" é expresso em função de operações fundamentais, as quais variam de algoritmo para algoritmo, dependendo também do volume de dados de entrada.

Eficiência de algoritmos

Para um dado problema, considere dois algoritmos que o resolvem.

Seja *n* um parâmetro que caracteriza o tamanho da entrada do algoritmo.

Por exemplo, ordenar *n* números ou multiplicar duas matrizes quadradas *n* x *n* (cada uma com n^2 elementos).

Como podemos comparar os dois algoritmos para escolher o melhor ?

Complexidade de tempo ou de espaço

Precisamos definir uma medida que expresse a eficiência dos algoritmos. Podemos avaliar um algoritmo em termos de tempo de execução ou do espaço (memória) utilizado.

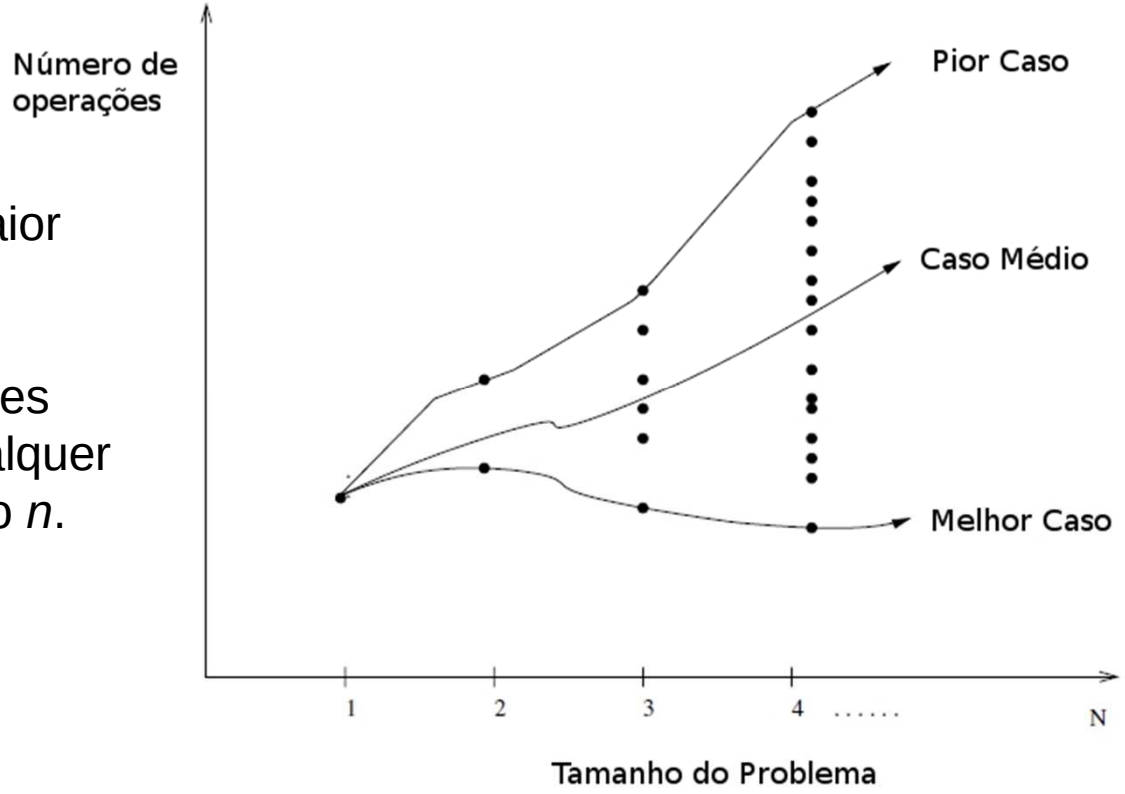
Para medir o tempo, poderiamos considerar o tempo absoluto (em segundos, minutos, horas, dias etc.). Porém, medir o tempo absoluto não é interessante por apresentar dependência com o *hardware* utilizado.

Dado esse fato, em análise de algoritmos, contamos o número de operações (consideradas relevantes) realizadas pelo algoritmo, expressando esse número como uma função do número de entradas *n*.

Melhor caso, caso médio e pior caso

A quantidade de operações realizadas por um algoritmo específico pode depender do tamanho da entrada do mesmo.

Nosso interesse maior reside no pior caso, definido pelo maior número de operações realizadas para qualquer entrada de tamanho *n*.



Extraído e adaptado de [1]

Análise do melhor caso

Definido pela letra grega Ω (Ômega).

Representa o menor tempo de execução de um algoritmo para uma dada entrada de tamanho *n*.

É raramente utilizado, por ter aplicação em poucos casos.

Exemplo:

O algoritmo de pesquisa sequêncial em um vetor tem complexidade f(*n*) = Ω(1).

A análise assume que o número procurado tenha sido o primeiro encontrado na lista.

Abordagem otimista!

Análise do caso médio

Representado pela letra grega Θ (Theta).

Deve-se obter a média dos tempos de execução de todas as entradas de tamanho *n*.

Outra forma é determinar a probabilidade de uma determinada condição ocorrer.

Exemplo:

O algoritmo de pesquisa sequêncial em um vetor tem complexidade f(*n*) = Θ(*n*/2).

Em média será necessário visitar *n*/2 elementos do vetor até encontrar o elemento procurado.

Muito dificil de determinar na maioria dos casos!

Análise do pior caso

Representado pela letra grega O (Ômicron).

Baseia-se no maior tempo de execução sobre todas as entradas de tamanho *n*.

É o metodo mais facil de se determinar.

Exemplo:

O algoritmo de pesquisa sequêncial em um vetor tem complexidade f(*n*) = *O*(*n*).

No pior caso será necessário visitar todos os *n* elementos do vetor até encontrar o elemento procurado.

Abordagem pessimista!

Notação *O*

Vamos analisar o seguinte exemplo: $\sum_{i=1}^{N} i^3$

```
int Soma(int N)
{
          int i, resultado;
(1)       resultado = 0;
(2)       for (i=1; i<=N;i++);
(3)            resultado += i*i*i;
(4)       return resultado;
}
```

Sendo que cada operação realizada consome 1 unidade de tempo do computador, e a declaração da função não conta tempo, temos:

- As linhas (1) e (4) realizam 1 operação cada, ou seja, 1 unidade de tempo cada: _____ 0N + 2

- A linha (3) possui 4 operações (duas multiplicações, uma soma e uma atribuição) e é executada N vezes, levando 4 unidades de tempo para ser executada: _____ 4N + 0

- A linha (2) possui uma inicialização de i, testes para i<=N e incremento de i, totalizando um custo de 1 + N+1 + N: _____ 2N + 2

6N + 4

Comportamento assintótico

Um caso particular interessante é quando n tem valor muito grande ($n \to \infty$), denominado comportamento assintótico.

Consideremos como exemplo os dois algoritmos a seguir:

Algoritmo I: $f_1(n) = 5n^2 + 7n$ operações
Algoritmo II: $f_2(n) = 600n + 3000$ operações

Os termos inferiores as constantes multiplicativas contribuem pouco na comparação e podem ser descartados.

O importante é verificar que $f_1(n)$ cresce com n^2, ao passo que $f_2(n)$ cresce com n. Um crescimento quadrático é considerado pior que um crescimento linear.

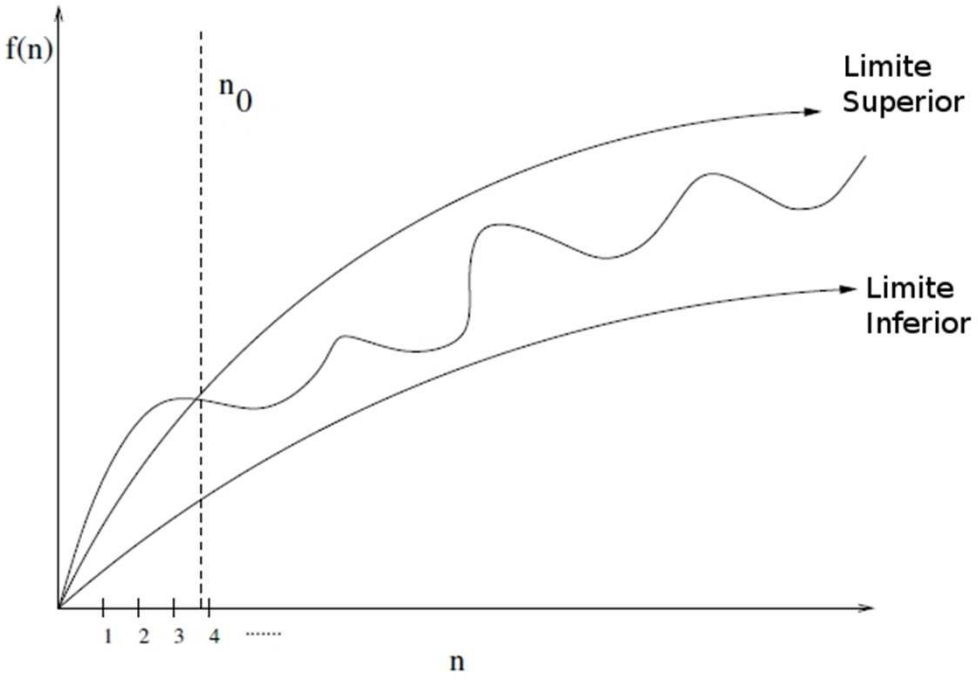
Dessa forma, devemos optar pelo algortimo II ao invés do algoritmo I.

Limite superior e inferior

Um algoritmo como busca binária normalmente é executado um pouco mais rápido em vetores de tamanho exato $n=2^k-1$ (onde k é inteiro).

Trata-se de um pequeno detalhe, porém cria oscilações no tempo de execução do mesmo.

Essas oscilações são representadas dentro de um limite superior e de um limite inferior, como mostra a figura.



Extraído e adaptado de [1]

Mudança de limite superior

Dado o problema de multiplicação de duas matrizes quadradas n x n.

Sabemos que um algoritmo pode resolver esse problema pelo método trivial, com complexidade $O(n^3)$. Logo, o limite superior desse problema é $O(n^3)$.

Porém, o limite superior de um problema pode mudar, se alguem descobrir um algoritmo melhor.

Para esse mesmo problema, o algotirmo de Strassen reduziu a complexidade Para $O(n^{\log 7})$, alterando assim o limite superior.

Coopersmith e Winograd melhoraram ainda mais para $O(n^{2,376})$.

Extraído e adaptado de [1]

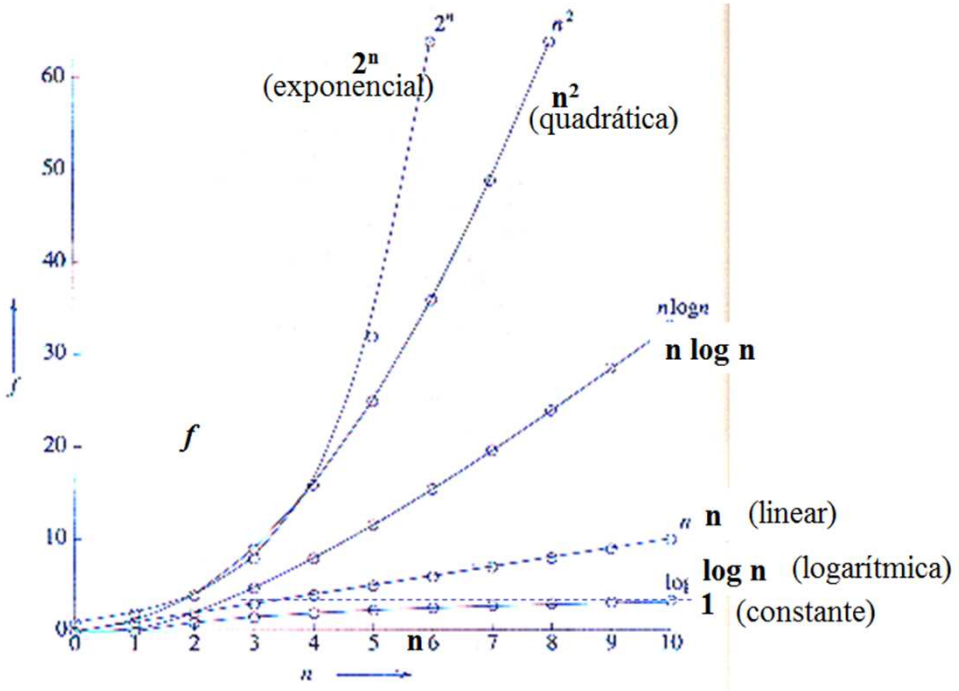
Relações de dominância na notação *O*

O uso da notação *O* agrupa as funções em um conjunto de classes, sendo que todas as funçoes equivalentes pertencem a mesma classe.

Por exemplo, f(*n*)=0,37*n* e g(*n*)=234,664 pertencem a mesma classe.

Listamos, a seguir, essas classes:

Funções constantes: **f(*n*) = 1**
Funções logaritmicas: **f(*n*) = log *n***
Funções lineares: **f(*n*) = *n***
Funções superlineares: **f(*n*) = *n* log *n***
Funções quadráticas: **f(*n*) = n^2**
Funções cúbicas: **f(*n*) = n^3**
Funções exponenciais: **f(*n*) = c^n**
Funções fatoriais: **f(*n*) = *n*!**



Complexidade constante

Representação: *O*(1)

Descrição: São algoritmos onde a complexidade independe do tamanho *n* de entradas.

É o único em que as instruções dos algoritmos são executadas um número fixo de vezes.

```
if (condição == true) then
{
          ; realiza operação em tempo constante
}
else
{
          ; realiza operação em tempo constante
}
```

Complexidade linear

Representação: *O*(*n*)

Descrição: Uma operação é realizada em cada elemento de entrada.

Como exemplo, a pesquisa de elementos em uma lista.

```
for (i=0; i<N; i++)
{
     if (condição == true) then
     {
               ; realiza operação em tempo constante
     }
     else
     {
               ; realiza operação em tempo constante
     }
}
```

Complexidade logarítimica

Representação: $O(\log n)$

Descrição: Ocorre tipicamente em algoritmos que dividem o problema em partes menores.

```
int PesquisaBinaria ( int array[], int chave , int N)
{
   int inf = 0;
   int sup = N - 1;
   int meio;
   while (inf <= sup)
   {
     meio = (inf+sup)/2;
     if (chave == array[meio]) return meio;
     else
         if (chave < array[meio]) sup = meio-1;
         else inf = meio+1;
   }
   return -1;   // não encontrado
}
```

Complexidade log linear (superlinear)

Representação: $O(n \log n)$

Descrição: Ocorre tipicamente em algoritmos que dividem o problema em partes menores, porém, juntando posteriormente a solução dos problemas menores.

```
void merge(int inicio, int fim)
{
   if (inicio < fim)
  {
          int meio = (inicio + fim) / 2;
          merge(inicio, meio);
          merge(meio + 1, fim);
          mesclar(inicio, meio, fim);
   }
}
```

Complexidade quadrática

Representação: $O(n^2)$

Descrição: Itens são processados aos pares, geralmente com um *loop* dentro do outro (*loops* aninhados).

```
void bubbleSort(int[] a)
{
   for (int i = 0; i < a.length-1; i++)
  {
          for (int j = 0; j < a.length-1; j++)
          {
                    if (a[j] > a[j+1])
                    {
                              swap(a, j, j+1);
                    }
          }
   }
}
```

Complexidade cúbica

Representação: $O(n^3)$

Descrição: Itens são processados três a três, geralmente com um *loop* dentro de outros dois.

```
int dist[N][N];
int i, j, k;

for ( k = 0; k < N; k++ )
   for ( i = 0; i < N; i++ )
     for ( j = 0; j < N; j++ )
               dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
```

Complexidade exponencial e fatorial

Representação: $O(2n)$ e $O(n!)$

Descrição: Essas classes configuram a utilização de força bruta para obtenção da solução de um problema.

A classe exponencial geralmente é implementada com base diretamente no enunciado do problema e nas definições dos conceitos envolvidos. Como exemplo dessa classe, temos a busca por senhas numéricas de n digitos, representando um espaço de 10^n senhas.

A classe fatorial consiste em testar todas as possíveis permutações existentes na solução, a procura da solução ótima para o problema. Como exemplo dessa classe, temos o problema do Caixeiro Viajante. Tal problema envolve encontrar a rota mínima para visitar várias cidades sem repetir nenhuma.

Taxas de crescimento

Dado um computador que consome 1 nanosegundo para executar uma operação, a tabela abaixo mostra o tempo de execução aproximado de um algoritmo com diferentes ordens de complexidade, para 13 tamanhos de entrada:

n f(n)*	lg n	n	n lg n	n^2	2^n	n!
10	0,003 µs	0,01 µs	0,033 µs	0,1 µs	1 µs	3,63 ms
20	0,004 µs	0,02 µs	0,086 µs	0,4 µs	1 ms	77,1 anos
30	0,005 µs	0,03 µs	0,147 µs	0,9 µs	1 s	$8,4x10^{15}$ anos
40	0,005 µs	0,04 µs	0,213 µs	1,6 µs	18,3 min	
50	0,006 µs	0,05 µs	0,282 µs	2,5 µs	13 dias	
100	0,007 µs	0,1 µs	0,644 µs	10 µs	$4x10^{13}$ anos	
1.000	0,010 µs	1,00 µs	9,966 µs	1 ms		
10.000	0,013 µs	10 µs	130 µs	100 ms		
100.000	0,017 µs	0,10 ms	1,67 ms	10 s		
1.000.000	0,020 µs	1 ms	19,93 ms	16,7 min		
10.000.000	0,023 µs	0,01 s	0,23 s	1,16 dias		
100.000.000	0,027 µs	0,10 s	2,66 s	115,7 dias		
1.000.000.000	0,030 µs	1 s	29,90 s	31,7 anos		

Extraído e adaptado de [1]

Problemas intratáveis e aproximações

Existe ainda a classe de problemas para os quais não existe solução algoritmica eficiente.

São os chamados “problemas intratáveis”. Em geral, são necessárias utilização de heurísticas para aproximar a solução ótima.

Referências

[1] Skiena, S. S. The Algorithm Design Manual, Second Edition, Springer, ISBN: 978-1-84800-069-8, Springer-Verlag 2008.