

Análise de Complexidade dos Algoritmos

Análise de algoritmos

Objetivos

- Provar que um algoritmo está correto
- Determinar os recursos exigidos pelo algoritmo (tempo, espaço, ...)
 - geralmente existem vários algoritmos para o mesmo problema
 - como escolher o “melhor”
 - comparar os recursos exigidos pelos diferentes algoritmos
 - um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema
- prever o crescimento dos recursos exigidos por um algoritmo, à medida que a quantidade de dados de entrada (tamanho de entrada) cresce (taxa de crescimento)

Tipos de análise

- Pormenorizada
 - mais exata e direta, mas em geral menos útil
 - expressa em segundos
 - resultado da avaliação da eficiência (por ex: tempo gasto):
 - único
 - dependente da velocidade e características do processador
- Através de ordens de grandeza
 - ignora as constantes multiplicativas
 - é uma análise **assintótica** (ignora os valores pequenos, considera os valores grandes)
 - expressa em ordens de grandeza (ordens assintóticas)
 - resultado da avaliação da eficiência:
 - paramétrico (uma função da quantidade de dados de entrada)
 - independente da velocidade e características do processador

Recursos

- A eficiência de um algoritmo é medida pela quantidade de recursos gastos durante a sua execução, como função da quantidade de dados de entrada
- O que são recursos?
 - tempo de execução do algoritmo (o mais usual)
 - espaço de memória necessário para guardar os dados

Tempo de execução de um algoritmo, $T(n)$

- A quantidade de dados de entrada depende do problema
 - normalmente é um só número
 - em algoritmos que envolvem arrays de 1D (ordenação, pesquisa, ...), a quantidade de dados é o número de elementos do array
 - mas podem ser mais
 - em algoritmos que envolvem grafos (caminho mais curto, ..), a quantidade de dados é geralmente expresso por duas quantidades: o número de nós e o número de arcos (temática a estudar mais tarde)

Tempo de execução de um algoritmo, $T(n)$

- Normalmente, o tempo exato não é importante
- Normalmente, o tempo é medido pelo número de operações primitivas do algoritmo que são executadas
 - operações primitivas: atribuições, comparações, ...
 - cada operação primitiva do algoritmo requer tempo constante
 - independente da quantidade de dados
 - operações não primitivas diferentes podem requerer tempos diferentes
 - chamadas de módulos (subprogramas/funções) também têm tempo constante
 - mas a execução do módulo (subprograma/função) poderá não ter

Ordem de crescimento

- O tempo de execução de um algoritmo é expresso como função da quantidade de dados de entrada do problema (em geral um número inteiro positivo)
- Para quantidades de dados elevadas, simplifica-se para se concentrar no essencial
 - eliminam-se os termos de ordem inferior
 - ignoram-se as constantes multiplicativas, como o coeficiente do termo de ordem superior
- Exemplo
 - tempo de execução: $T(n) = a \times n^2 + b \times n + c$
 - eliminam-se os termos de ordem inferior $\Rightarrow T(n) = a \times n^2$
 - ignora-se o coeficiente do termo de ordem superior $\Rightarrow T(n) = n^2$
- Aspeto importante
 - não se pode dizer que $T(n) = n^2$
 - mas pode-se dizer que $T(n)$ “cresce” de modo proporcional a n^2
- Diz-se que $T(n) = O(n^2)$ para capturar a noção de que a ordem de crescimento é proporcional a n^2 (chama-se a isto **análise assintótica**)

Análise assintótica

- Apenas a ordem de crescimento do tempo de execução é relevante
 - só se analisa o comportamento assintótico dos algoritmos:
 - se o algoritmo A1 é assintoticamente melhor que o algoritmo A2, então A1 será melhor escolha do que A2, exceto possivelmente para pequenas quantidades de dados
- Para valores enormes de **n**, as seguintes funções são da mesma ordem assintótica, por terem todas a mesma “taxa de crescimento” (são todas “equivalentes”)
 - n^2
 - $1000 \times n^2$
 - $1/1000 \times n^2$
 - $n^2 + 100 \times n$
 - ...

Análise assintótica

- Uma função f é **assintoticamente não-negativa** se
$$\exists m : f(n) \geq 0, \forall n > m \text{ (para todo o } n \text{ suficientemente grande)}$$
- Dadas duas funções assintoticamente não-negativas, T e f , diz-se que T está na ordem **Ó** de f e escreve-se $T(n) = O(f(n))$ se existem constantes $c > 1$ e $m > 0$ tais que $T(n) \leq c \times f(n), \forall n > m$

Complexidade dos Algoritmos

Trata da eficiência computacional dos algoritmos

- Complexidade temporal
 - exigências de tempo de processamento (tempo de execução)
- Complexidade espacial
 - exigências do espaço de armazenamento

Eficiência dos algoritmos

- A dimensão do problema será denotada por **n** (inteiro positivo)
- Na análise de um algoritmo
 - interessa o seu comportamento no caso geral, e não num caso específico
 - questão principal:

Qual o desempenho do algoritmo à medida que **n** cresce e se torna muito elevado?

Complexidade espacial de um algoritmo

- Espaço de memória necessário para executar até ao fim, **$S(n)$**
 - espaço de memória exigido em função da quantidade de dados de entrada **n**

Complexidade temporal de um algoritmo

- Tempo que demora a executar (tempo de processamento), **$T(n)$**
 - tempo de execução em função da quantidade de dados de entrada **n**

Complexidade vs. Eficiência

- À medida que a complexidade aumenta a eficiência diminui

Por vezes estima-se a complexidade para

- O “melhor caso” (pouco útil)
- O “pior caso” (mais útil)
- O “caso médio” (igualmente útil)

Complexidade temporal

Objetivo do estudo da complexidade

- Caracterizar o tempo de execução necessário para a resolução de um problema em função da quantidade de dados de entrada
 - interessa o caso mais difícil (a pior situação, o pior caso)
- Complexidade temporal, **$T(n)$** , significa que a computação de qualquer algoritmo de dimensão **n** pode ser completada em não mais de $T(n)$ operações

Complexidade assintótica

- Normalmente diz-se que um algoritmo é mais eficiente do que outro, se o seu tempo de execução **no pior caso** tiver uma menor ordem de crescimento (menor complexidade assintótica)

Crescimento de funções

- Ter um modo de descrever a **escalabilidade** de algoritmos
 - Quando se duplica a quantidade de dados de entrada de um problema, o que é que acontece ao tempo de execução?
- Na prática, é difícil (ou mesmo impossível) prever com rigor o tempo de execução de um algoritmo
- Pretende-se obter o tempo de execução a menos de
 - constantes multiplicativas (são tempos de execução de operações primitivas)
 - parcelas menos significativas para valores grandes de **n**
- Para isto
 - identificam-se as operações primitivas dominantes (mais frequentes ou mais demoradas)
 - determina-se o número de vezes que são executadas
(não o tempo de cada execução, que seria uma constante multiplicativa)
 - exprime-se o resultado com a notação de "O grande"

O tempo de execução em geral depende de um único parâmetro n

- Ordem de um polinómio
- Tamanho de um ficheiro a ser processado, ordenado, etc.
- Medida abstrata do tamanho do problema a considerar
 - normalmente relacionado com a quantidade de dados

Quando há mais do que um parâmetro

- Procura-se exprimir todos os parâmetros em função de um só
- Faz-se uma análise em separado para cada parâmetro

Notação de “O grande” (Big-O)

Definição

- **$T(n) = O(f(n))$** (lê-se $T(n)$ é de ordem $f(n)$)
 - o tempo de execução $T(n)$ de um algoritmo com **n** dados de entrada é de ordem $f(n)$ se existem constantes **$c > 1$** e **$m > 0$** tais que $T(n) \leq c \times f(n)$, $\forall n > m$
 - ou seja, se um algoritmo é $O(f(n))$, então há um ponto **m** a partir do qual o desempenho do algoritmo é limitado a um múltiplo de $f(n)$
 - $f(n)$ é assintoticamente um limite superior para $T(n)$, a menos de um fator constante
- Exemplos
 - 1) $3 \times n^2 = O(n^2)$
 - fazer $c = 3$ e $m = 1$: $3 \times n^2 \leq 3 \times n^2$, $\forall n > 1$
 - 2) $3 \times n^2 = O(n^3)$
 - fazer $c = 3$ e $m = 1$: $3 \times n^2 \leq 3 \times n^3$, $\forall n > 1$
 - 2) $10 \times n^2 + 10 \times n = O(n^2)$
 - fazer $c = 12$ e $m = 5$: $10 \times n^2 + 10 \times n \leq 12 \times n^2$, $\forall n > 5$

Esta notação é usada com três objetivos

- Limitar o erro que é feito ao ignorar os termos de ordens menores nas fórmulas matemáticas
- Limitar o erro que é feito na análise ao desprezar parte do algoritmo que contribui de forma mínima para o tempo de execução (complexidade) total
- Permitir classificar algoritmos de acordo com limites superiores do seu tempo de execução

Ordens mais comuns

- **1**: tempo de execução constante
 - muitas operações são executadas uma só vez ou poucas vezes
 - se isto acontece para todo o algoritmo, então o tempo de execução é constante
- **log n**: tempo de execução logarítmico
 - cresce ligeiramente à medida que **n** cresce:
 - quando **n** duplica, **log n** aumenta mas muito pouco
- **n**: tempo de execução linear
 - típico quando algum processamento é feito para cada dado de entrada
 - situação ótima quando é necessário
 - processar **n** dados de entrada, ou
 - produzir **n** dados na saída

Ordens mais comuns

- $n \log n$:

- típico quando
 - se reduz um problema em subproblemas
 - se resolvem estes separadamente
 - se combinam as soluções

- n^2 : tempo de execução quadrático

- típico quando é necessário processar todos os pares de dados de entrada
- prático apenas em pequenos problemas (ex: produto de vetores)

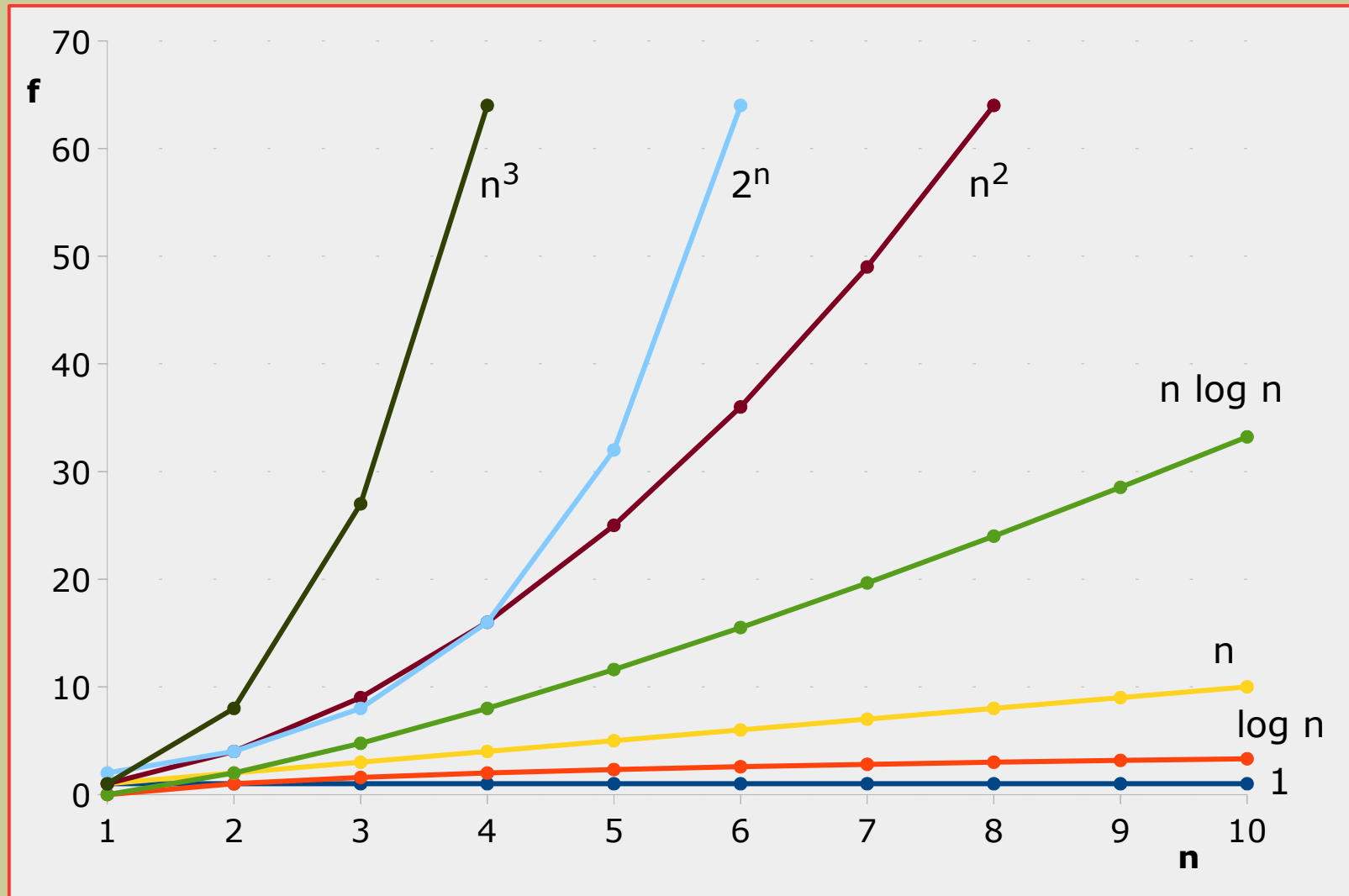
- 2^n : tempo de execução exponencial

- provavelmente de pouca aplicação prática

- n^3 : tempo de execução cúbico

- para $n = 100$, $n^3 = 1.000.000$ (ex: produto de matrizes)

Ordens mais comuns



Consequências da definição

- $O(n^3) + O(n^2) = O(n^3)$
- $O(n^2) + O(n) = O(n^2)$
- $O(\sum_{k=1,\dots,p} a_k n^k) = O(n^p)$
- $O(1)$ indica um trecho de algoritmo com tempo constante

Exemplos genéricos

$$c_k n^k + c_{k-1} n^{k-1} + \dots + c_0 = O(n^k)$$

(c_i - constantes)

$$\log_2 n = O(\log n)$$

(não se indica a base porque mudar de base é multiplicar por uma constante)

(mudança de base: $\log_a x = \log_b x / \log_b a$)

$$4 = O(1)$$

(usa-se 1 para ordem constante)

Metodologia para determinar a ordem de complexidade

Exemplo 1

- Considere-se a seguinte função em C (calcula a soma dos elementos dum array 1D)

```
int somaArray (int *A, int N)
{
    int k, soma;
    soma = 0;
    k = 0;
    while (k < N) {
        soma = soma + A[k];
        k = k + 1;
    }
    return soma;
}
```

- Pior caso
 - não existe, pois todos os elementos do array têm de ser tratados (analísados)

Exemplo 1

- Contabilização do número de vezes que as operações primitivas são executadas

k	soma = 0	k = 0	k < N	soma = soma + A[k]	k = k + 1
0	1	1	1 (V)	1	1
1	0	0	1 (V)	1	1
2	0	0	1 (V)	1	1
...	...	...	...	...	...
N - 2	0	0	1 (V)	1	1
N - 1	0	0	1 (V)	1	1
N	0	0	1 (F)	0	0
	1	1	N + 1	N	N

- $T(N): 1 + 1 + (N + 1) + N + N = 3 \times N + 3$

$T(N) = O(p(N))$, em que $p(N)$ é um polinómio de grau 1 em ordem a N

- Ordem de complexidade: **$O(N)$**

Exemplo 1

- Operação primitiva dominante (mais vezes executada): **$k < N$**
- Contabilização do número de vezes que a operação **dominante** é executada

k	$k < N$
0 ($k = 0$)	1 (V)
1 ($k = k+1$)	1 (V)
2 ($k = k+1$)	1 (V)
...	...
$N - 1$ ($k = k+1$)	1 (V)
N ($k = k+1$)	1 (F)
	$N + 1$

- $T(N): N + 1$

$T(N) = O(p(N))$, em que $p(N)$ é um polinómio de grau 1 em ordem a N

- Ordem de complexidade: **$O(N)$**

Exemplo 1

- Ordens de complexidade determinadas
 - contabilizando todas as operações: $O(N)$
 - contabilizando a operação dominante: $O(N)$
- Conclusão
 - o número de vezes que as operações que estão dentro do ciclo são realizadas não tem impacto no cálculo da ordem de complexidade
 - basta considerar a operação dominante, caso esta seja fácil de determinar

Exemplo 2

- Considere-se a seguinte função em C (determina o maior elemento dum array 1D)

```
int maiorElementoArray (int *A, int N)
{
    int k, maior;
    maior = A[0];
    k = 1;
    while (k < N) {
        if (A[k] > maior)
            maior = A[k];
        k = k + 1;
    }
    return maior;
}
```

Exemplo 2

- Contabilização do número de vezes que as operações primitivas são executadas
 - Pior caso:
 - para a operação "maior = A[k]" ser executada o máximo de vezes, o array deve estar ordenado por ordem crescente
 - as restantes operação são sempre executadas o máximo de vezes

k	maior = A[0]	k = 1	k < N	A[k] > maior	maior = A[k]	k = k + 1
1	1	1	1 (V)	1 (V)	1	1
2	0	0	1 (V)	1 (V)	1	1
3	0	0	1 (V)	1 (V)	1	1
...	...	...	...	...	...	...
N - 1	0	0	1 (V)	1 (V)	1	1
N	0	0	1 (F)	0 (F)	0	0
	1	1	N	N - 1	N - 1	N - 1

Exemplo 2

- Contabilização do número de vezes que as operações primitivas são executadas
 - $T(N): 1 + 1 + N + (N - 1) + (N - 1) + (N - 1) = 4 \times N - 1$
 $T(N) = O(p(N))$, em que $p(N)$ é um polinómio de grau 1 em ordem a N
- Ordem de complexidade: **$O(N)$**
- Questão:
 - Como se define o melhor caso e o caso médio?

Exemplo 2

- Operação primitiva dominante (mais vezes executada): **$k < N$**
- Contabilização do número de vezes que a operação dominante é executada
 - Pior caso: não existe, pois esta operação é sempre executada o máximo de vezes

k	$k < N$
1 ($k = 1$)	1 (V)
2 ($k = k+1$)	1 (V)
3 ($k = k+1$)	1 (V)
...	...
N - 1 ($k = k+1$)	1 (V)
N ($k = k+1$)	1 (F)
	N

- $T(N)$: N

$T(N) = O(p(N))$, em que $p(N)$ é um polinómio de grau 1 em ordem a N

- Ordem de complexidade: **$O(N)$**

Exemplo 3

- Considere-se a seguinte função em C (calcula a soma dos elementos dum array 2D)

```
int somaMatriz (int X[][], int M, int N)
{
    int k, soma;
    soma = 0;
    for (k = 0; k < M; k++) {
        for (j = 0; j < N; j++)
            soma = soma + V[k][j];
    }
    return soma;
}
```

- Pior caso
 - não existe, pois todos os elementos do array têm que ser tratados (ou analisados)

Exemplo 3

- Operação primitiva dominante (mais vezes executada): $j < N$
- Contabilização do número de vezes que a operação dominante é executada

k	k < M	j	j < N
0 (k = 0)	1 (V)	0 (j = 0) 1 (j++) ... N - 1 (j++) N (j++)	1 (V) 1 (V) ... 1 (V) 1 (F)
1 + 0	1	1 + N	N + 1
1 (k++)	1 (V)	0 (j = 0) 1 (j++) ... N - 1 (j++) N (j++)	1 (V) 1 (V) ... 1 (V) 1 (F)
0 + 1	1	1 + N	N + 1
...	...	...	...

Exemplo 3

- Contabilização do número de vezes que a operação dominante é executada

k	k < M	j	j < N
...	...	...	...
M - 1 (k = k+1)	1 (V)	0 (j = 0) 1 (j++) ... N - 1 (j++) N (j++)	1 (V) 1 (V) ... 1 (V) 1 (F)
0 + 1	1	1 + N	N + 1
M (k = k+1)	1 (F)	-	-
0 + 1	1	0	0
1 + M	M + 1	M + M × N	total

Exemplo 3

- Contabilização do número de vezes que a operação dominante é executada
 - resultados da simulação:
 - $k = 0 \Rightarrow N + 1$
 - $k = 1 \Rightarrow N + 1$
 - ...
 - $k = M-1 \Rightarrow N + 1$
 - $k = M \Rightarrow 0$
 - **total:** $M \times (N + 1) = M \times N + M$
 - fazendo **$M = a \times N$** , em que **a** é uma constante
 - $T(N)$: $M \times N + M = a \times N \times N + a \times N = a \times N^2 + a \times N$
 $T(N) = O(p_2(N))$, em que $p_2(N)$ é um polinómio de grau 2 em ordem a N
- Ordem de complexidade: **$O(N^2)$**

Exemplo 4

- Considere-se a seguinte função em C (pesquisa de um elemento num array 1D)

```
int pesquisaExaustiva (int E, int *A, int N)
{
    int k;
    k = 0;
    while (k < N && A[k] != E)
        k = k + 1;
    if (k == N)
        return -1;
    else
        return k;
}
```

- Operação dominante: ?
- Pior caso, melhor caso e caso médio: ?

Exemplo 4

- Operação primitiva dominante (mais vezes executada): **$k < N \ \&\& \ A[k] \neq E$**
- Contabilização do número de vezes que a operação dominante é executada
- Pior caso: o elemento E não está no array A

k	$k < N \ \&\& \ A[k] \neq E$
0 (k = 0)	1 (V)
1 (k = k+1)	1 (V)
2 (k = k+1)	1 (V)
...	...
N - 1 (k = k+1)	1 (V)
N (k = k+1)	1 (F)
	N + 1

- $T(N)$: $N + 1$; $T(N) = O(p(N))$, em que $p(N)$ é um polinómio de grau 1 em ordem a N
- Ordem de complexidade: **$O(N)$**