Algumas Noções de Complexidade

Luís Lopes

DCC-FCUP

Estruturas de Dados

Algoritmo

- um método para resolver um problema
- especifica, passo a passo, as operações a realizar
- as funções que manipulam as estruturas de dados que definimos são também algoritmos
- e.g., inserir elemento no fim de uma lista, encontrar o índice de um elemento na lista, remover um elemento no início da lista
- outros algoritmos operam sobre estruturas de dados completas
- e.g., ordenar um array, multiplicar matrizes, inverter uma lista

Eficiência de um Algoritmo

- normalmente mede-se com base em 2 critérios em função do tamanho do input n
- taxa de crescimento do *tempo de execução* (complexidade temporal)
- taxa de crescimento do *espaço usado na memória* (complexidade espacial)
- vamos centrar a nossa atenção na primeira métrica

Análise da Eficiência

- descrição matemática do algoritmo em vez de implementação
- caracteriza o tempo de execução como uma função do tamanho do input
- tem em conta todos os possíveis inputs
- permite avaliar eficiência de forma independente do ambiente de hardware/software

Exemplo

Estimativa do tempo de execução em função de n

```
int findMax(int a[], int n) {
  int max = a[0];          // 2 operações
  int i   = 1;             // 1 operação
  while (i <= n - 1) {     // n operações
    // n-1 vezes ciclo
    if (a[i] > max)        // 2 operações
      max = a[i];          // 2 operações
    i = i + 1;             // 2 operações
  }
  return max;              // 1 operação
}
```

Pressupostos

- contabilizar número de instruções primitivas
- memória de acesso aleatório e ilimitada
- e.g., `max = a[0]` é composta por 2 operações primitivas
  - 1 leitura de `a[0]` + 1 atribuição a `max`

Exemplo (cont.)

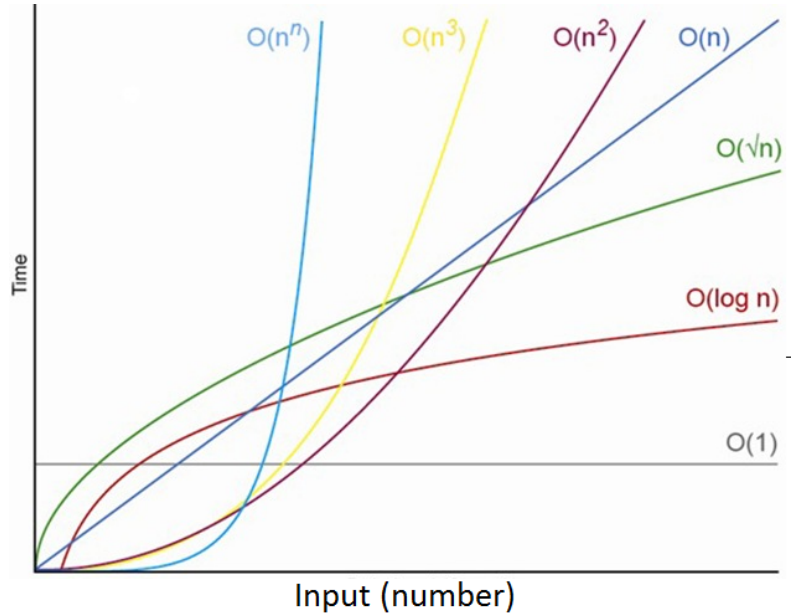
- caso mais favorável (`a[0]` é o maior elemento):
  $T(n) = 2 + 1 + n + 4(n-1) + 1 = 5n$ operações primitivas
- pior caso:
  $T(n) = 2 + 1 + n + 6(n-1) + 1 = 7n - 2$
- caso médio depende da distribuição do input
- o pior caso dá-nos um limite superior do tempo de execução
- neste caso $T(n) \propto n$

Taxa de Crescimento de $T(n)$

- o tempo de execução, $T(n)$, pode ser afectado pela alteração do ambiente hardware/software
- tal não acontece se considerarmos a taxa de crescimento de $T(n)$: a variação de $T(n)$ quando se aumenta o valor de n.
- para a função `findMax()`, $T(n)$ é limitado por 2 funções lineares em n
- diz-se que o crescimento de $T(n)$ é linear

Taxas de crescimento

logarítmica	$\log n$
linear	n
n-logarítmica	$n \log n$
quadrática	n^2
cúbica	n^3
exponencial	$a^n (a > 1)$



Crescimento de algumas funções:

n	$\log_2 n$	$\sqrt{n}$	n	$n \log_2 n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
8	3	2.8	8	24	64	512	256
16	4	4.0	16	64	256	4096	65536
...	...	...	...	...	...	...	...
1024	10	32	1024	10240	$> 10^6$	$> 10^9$	$> 10^{308}$

Taxas de crescimento

Se assumirmos que cada operação pode ser executada em $1\mu s$ (micro-sec), qual será o maior problema (função de n) para um programa que execute em 1 seg., 1 min., 1 hora?

	Tamanho máximo problema (n)		
$T(n)$	*1 seg.*	*1 min.*	*1 hora*
$400n$	2500	150,000	9,000,000
$20n\lceil \log n \rceil$	4096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

No caso de crescimento exponencial, apenas conseguimos tratar problemas para dimensões muito pequenas de n.

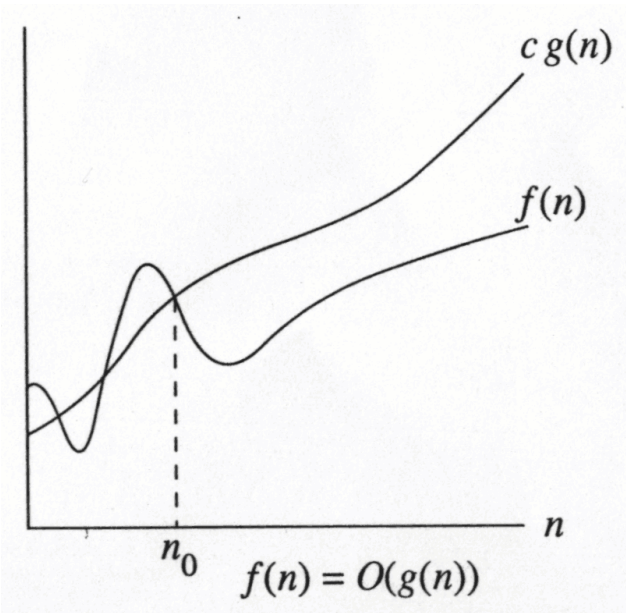
Ordem de Complexidade Assimptótica: $\mathcal{O}()$

Sejam $f(n)$ e $g(n)$ funções de $\mathbb{N}_0 \longmapsto \mathbb{R}$.

$f(n)$ é $\mathcal{O}(g(n))$ (ou seja, $f(n)$ é da ordem de $g(n)$)

se $\exists c > 0, \exists n_0 \geq 1 : f(n) \leq cg(n), \forall n \geq n_0$

Interpretação gráfica:



- $\mathcal{O}(g(n))$ corresponde ao limite superior da taxa de crescimento de $f(n)$.
- dizer que $f(n)$ é $\mathcal{O}(g(n))$, significa dizer que $f(n)$ não cresce mais do que $g(n)$.

Exemplo

- $2n + 10$ é $\mathcal{O}(n)$ porque:

$$2n+10 \leq cn \quad \Leftrightarrow \quad (c-2)n \geq 10 \quad \Leftrightarrow \quad n \geq \frac{10}{(c-2)}$$

para $c = 3$ e $n_0 = 10$ verifica-se sempre que $2n + 10 \leq cn$

- outras funções:

$$\begin{aligned} 20n^3 + 10n\log n + 5 &\quad — \quad \mathcal{O}(n^3) \\ a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 &\quad — \quad \mathcal{O}(n^k) \\ 3\log n + \log\log n &\quad — \quad \mathcal{O}(\log n) \\ 2^{100} &\quad — \quad \mathcal{O}(1) \\ \frac{5}{n} &\quad — \quad \mathcal{O}(\frac{1}{n}) \end{aligned}$$

Algumas Propriedades Úteis

- Se $h(n) \sim \mathcal{O}(f(n))$, então $\alpha h(n)$ (com $\alpha > 0$) $\sim \mathcal{O}(f(n))$
- Se $h_1(n) \sim \mathcal{O}(f(n))$ e $h_2(n) \sim \mathcal{O}(g(n))$, então $h_1(n) + h_2(n) \sim \mathcal{O}(f(n) + g(n))$
- Se $h_1(n) \sim \mathcal{O}(f(n))$ e $h_2(n) \sim \mathcal{O}(g(n))$, então $h_1(n)h_2(n) \sim \mathcal{O}(f(n)g(n))$
- Se $h(n) \sim \mathcal{O}(f(n))$ e $f(n) \sim \mathcal{O}(g(n))$, então $h(n) \sim \mathcal{O}(g(n))$

Exemplo: Método da Bolha (Bubble-Sort)

```
void bubbleSort(int a[], int n) {
  int i, j, tmp;
  for(int i = 0; i < n; i++)          // n iterações
    for(int j = 0; j < n-1-i; j++) // n-i iterações
      if (a[j] > a[j+1]) {
        tmp    = a[j+1];
        a[j+1] = a[j];
        a[j]   = tmp;
      }
}
```

Efectivamente, o número de iterações total é a soma das iterações que o ciclo `j` faz para cada valor de `i`, i.e.

$$T(n) \sim \sum_{i=1}^{n-1}(n-i) \quad = (n-1)+(n-2)+\ldots+2+1 =$$
$$= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$
$$\Rightarrow T(n) \text{ é } \mathcal{O}(n^2)$$

Exemplo: Factorial

```
int factorial(int n) {
  if(n == 0 || n == 1)          // O(1)
    return 1;                   // O(1)
  else
    return n * factorial(n-1);  // O(1) + T(n-1)
}
```

$$\therefore T(n) = \begin{cases} \alpha + T(n-1), & \text{se } n > 1 \\ \beta, & \text{se } n \leq 1 \end{cases}$$

assumindo $n \geq 2$:

$T(n-1) = \alpha + T(n-2) \Rightarrow T(n) = 2\alpha + T(n-2)$

em geral: $T(n) = i\alpha + T(n-i), n > i$

se $i = n-1$, então $T(n) = \alpha(n-1) + T(1) = \alpha(n-1) + \beta$

donde podemos concluir que $T(n)$ é $\mathcal{O}(n)$

Exemplo: Pesquisa Binária

```
int binarySearch(int[] values, int val, int low, int high) {
  if( high < low )
    return -1;
  else {
    int half = low + ( high - low ) / 2;
    if ( val == values[half] ) )
      return half;
    else if ( val < values[half] ) )
      return binarySearch(values, val, low, half - 1);
    else
      return binarySearch(values, val, half + 1, high);
  }
}
```

- em cada passo do ciclo, o intervalo reduz-se a metade.
- para um n dado, o algoritmo pára quando atingir um intervalo de tamanho ≤ 1

Exemplo: Pesquisa Binária

- qual dos valores $n/2, n/4, n/8, \ldots, n/2^k \ldots$ será o primeiro a ser ≤ 1?
- temos de resolver $n/2^k \leq 1$, o que dá um número de passos $k \geq \log_2 n$
- tomando o mais pequeno inteiro k que satisfaz $k \geq \log_2 n$ ($k = \lceil \log_2 n \rceil$) sabemos que ao fim de um máximo de k iterações encontramos o valor no array ou podemos concluir que o valor que procuramos não existe
- deduzimos assim que $T(n) = \lceil \log_2 n \rceil$ ou seja $T(n)$ é $\mathcal{O}(\log n)$