Complexidade
Part I

Ana Neri
ana.i.neri@inesctec.pt

DI,
University of Minho

February 20, 2023

Conteúdo

Análise de complexidade

O que é análise de complexidade?
Ramo dedicado ao estudo dos recursos necessários à execução de uma programa / algoritmo.

Estes recursos podem ser:

- tempo de execução
- memória usada
- energia consumida

Análise temporal

Fatores a ter em conta:

- O tamanho do input;
- A forma do input;
- E o hardware em que algoritmo é executado.

Vamos ver isto num **exemplo de procura num array desordenado**.

PROCURA NUM ARRAY DESORDENADO

A função `procura` procura `x` num vetor `v`, e devolve -1 se não encontrar ou o índice em que `x` se encontra.

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

TAMANHO DO INPUT

Os argumento são:

- um inteiro `x`
- e um array `v`

O tamanho do input é:

$$(N+1) * I \tag{1}$$

onde I é o número de bits usados para representar um inteiro.
Podemos analisar o tamanho do input em função do array argumento.

$$T :: N \to N \tag{2}$$

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

CONSIDERAÇÕES COM HARDWARE

No que diz respeito à maquina temos de identificar os componentes atómicos.

Componente Atómica:

Componentes com custo / complexidade são constante.

Operações		**Custo**
atribuições	(= e +=)	c_1
comparação	(<)	c_2
seleção em array	(`v[i]`)	c_3
teste de igualdade	(/= e ==)	c_4
comprimento	(`len(v)`)	c_5

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

Forma do input

O melhor caso, o pior caso e o caso médio

Se o vector usado já estiver ordenado, em principio é possível resolver o problema com uma complexidade é menor.

É costume analisarmos os 3 casos:

- **pior caso** estabelece um limite superior para o custo.
- **melhor caso** estabelece um limite inferior para o custo.
- **caso médio** estabelece um valor esperado do custo.

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

Atenção! Esta analise não depende no tamanho do array, `N`.

O MELHOR CASO

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

Operações		**Custo**
atribuições	(= e ++)	c_1
comparação	(<)	c_2
seleção em array	(`v[i]`)	c_3
teste de igualdade	(/= e ==)	c_4
comprimento	(`len(v)`)	c_5

Se o valor que procuramos está na primeira posição, então temos de contar com:

- c_1 do operação $i = 0$
- $c_5 + c_1$ da operação $N = len(v)$
- $c_2 + c_3 + c_4$ do teste `while`
- c_4 do teste `if`

Logo o custo é:

$$T(N) = 2 * c_1 + c_2 + c_3 + 2 * c_4 + c_5 \quad (3)$$

O custo é constante.
Não depende do número de elementos no array.

O PIOR CASO

Neste caso o elemento não existe. Logo a condição do ciclo (`v[i]!=x`) é sempre verdadeira.

$$T(N) \tag{4}$$

$$= \overbrace{c_1}^{i=0} + \overbrace{c_1 + c_5}^{\texttt{len(v)}} \overbrace{\sum_{i=0}^{N-1} (\underbrace{c_2 + c_3 + c_4}_{\text{teste while}} + \underbrace{c_1}_{i+=}) + \underbrace{c_2}_{i<N}}^{\text{ciclo while}} + \overbrace{c_4}^{\text{teste if}} \tag{5}$$

$$= 2 * c_1 + c_5 + (c_2 + c_3 + c_4) * \sum_{i=0}^{N-1} (1 + c_1 + c_2) + c_4 \tag{6}$$

$$= K_1 * N + K_2 \tag{7}$$

K_1 e K_2 são constantes.

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

Operações		**Custo**
atribuições	(= e ++)	c_1
comparação	(<)	c_2
seleção em array	(`v[i]`)	c_3
teste de igualdade	(/= e ==)	c_4
comprimento	(`len(v)`)	c_5

Logo a função de custo é um polinómio (de grau 1) sobre N.

O CASO MÉDIO

Temos de identificar todas as execuções r, e para cada uma delas determinar o custo c_r e a probabilidade p_r

$$\overline{T}(N) = \sum_r p_r * c_r \quad (8)$$

Podemos dividir as execuções em 2 grupos:

- casos de sucesso - a função retorna um índice
- ou casos de insucesso

Digamos que o primeiro caso tem probabilidade:

$$\overline{T}(N) = p_{suc} * \overline{T}_{suc}(N) + p_{ins} * \overline{T}_{ins}(n) \quad (9)$$

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

O CASO MÉDIO

No caso $\overline{T}_{ins}(N)$ o ciclo só termina quando $i < N$ for falsa:

$$\overline{T}_{ins}(N) = N \tag{10}$$

No caso $\overline{T}_{suc}(N)$ o elemento a procurar pode ocorrer em qualquer probabilidade em qualquer das N posições do array.

$$\overline{T}_{suc}(N) = \sum_{i=0}^{N-1} \overbrace{\frac{1}{N}}^{\text{prob(x==v[i])}} * \overbrace{(i+1)}^{\text{custo}} = \frac{N+1}{2} \tag{11}$$

Agora só falta sabermos as probabilidade de sucesso e insucesso.

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

$$\overline{T}(N) = p_{suc} * \overline{T}_{suc}(N) + p_{ins} * \overline{T}_{ins}(n)$$

O CASO MÉDIO

Trata-se da procura de uma procura de inteiro num array de inteiros.
Se estes valores forem completamente aleatórios então a probabilidade de sucesso é quase nula $\frac{1}{2^b}$, onde b é o número de bits usados para representar o elemento.

$$p_{ins} = (1 - \frac{1}{2^b}) \tag{12}$$

Logo:

$$\overline{T}(n) \approx 0 * \frac{N+1}{2} + 1 * N \approx N \tag{13}$$

```
def procura (x, v):
  i=0
  N =len(v)
  while ((i<N) and v[i] != x):
    i+=1
  if (i==N): return -1
  else: return i
```

$$\overline{T}(N) = p_{suc} * \overline{T}_{suc}(N) + p_{ins} * \overline{T}_{ins}(n)$$

$$\overline{T}_{ins}(N) = N \qquad \overline{T}_{suc}(N) = \frac{N+1}{2}$$

Procura num array ordenado

Considerando um array ordenado por ordem crecente.

```
proc_lin (x,v):
    i=0
    N= len(v)
    while ((i<N) && (v[i])<x):
        i+=1
    if ((i==N) || (v[i] !=x)):
        return (-1)
    else:
        return i
```

Nesta caso só temos de analisar o numero de acessos ao array, isto é, depende apenas do número de iterações do ciclo.

Análise da eficiência

```
proc_lin (x,v):
    i=0
    N= len(v)
    while ((i<N) && (v[i])<x):
        i+=1
    if ((i==N) || (v[i] !=x)):
        return (-1)
    else:
        return i
```

Qual o custo do melhor caso?

$$T(N) = 2 \tag{14}$$

Qual é o custo do pior caso?

$$T(N) = N + 1 \tag{15}$$

Através da análise do melhor caso e do pior caso não é possível verificar que este caso tem uma melhoria clara.

Análise da eficiência

Caso médio

```
proc_lin (x,v):
    i=0
    N= len(v)
    while ((i<N) && (v[i])<x):
        i+=1
    if ((i==N) || (v[i] !=x)):
        return (-1)
    else:
        return i
```

A melhoria será evidente na análise do caso médio.
Considere que temos um array com valores uniformemente distribuídos e um valor de procura aleatório.
Há igual probabilidade do ciclo fazer 0 até $N-1$ iterações.
Com k iterações o número de acessos ao array é $k+2$.
Logo, temos N comportamentos cada um com probabilidade $\frac{1}{N}$.

$$\overline{T}(N) = \sum_{i=0}^{N-1} \overbrace{\frac{1}{N}}^{\text{prob}} * \overbrace{(i+2)}^{\text{custo}} = \frac{N+3}{2} \tag{16}$$

Em média acedemos a metade das posições do array.

Mais exemplos

Operações sobre inteiros

```
def prod(x,y){
    r=0
    while(x>0):
        r = r+y
        x = x-1
    return r
}
```

Qual é o custo do melhor caso?

Qual é o custo do pior caso?

Mais exemplos

Procura binária

```
def bsearch(x, v):
    i, s = 0, len(v) - 1
    while i < s:
        m = (i + s) // 2
        if v[m] == x:
            i = s = m
        elif v[m] > x:
            s = m - 1
        else:
            i = m + 1
    if (i > s) or (v[i] != x):
        return -1
    else:
        return i
```

Qual é o custo do melhor caso?

Qual é o custo do pior caso?

Qual é o custo do caso médio?

Análise Assimptótica

Para compararmos programas com base na sua eficiência, vamos agrupar os programas que para valores elevados do tamanho de input têm performances comparáveis.
Ou seja, vamos fazer o estudo do **crescimento assimptótico das funções**.

O que é isto do crescimento assimptótico?

Análise Assomptótica

Considere as funções

$$f(x) = (x+2)^2 \qquad g(x) = x^2 + 4 * x + 4 \qquad h(x) = (x+4)^2 \tag{17}$$

Se compararmos as funções com igualdade extensional:

$$f = g \quad \text{sse} \quad \forall_x f(x) = g(x) \tag{18}$$

f e g são iguais e f e h são diferentes.
Mas se usarmos uma definição menos restritiva:

$$f \sim g \quad \text{sse} \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1 \tag{19}$$

Este é a definição de corresponde à comparação assimptótica de funções, aqui $f \sim g \sim h$.
Este é um tipo de funções que pelo menos a partir de um certo valor têm taxas de crescimento iguais.
Vamos ver definições para caracterizar os limites superiores do crescimento de uma função.

Para uma função g, a classe (conjunto) de funções $o(g(x))$ defini-se por:

$$f \in o(g(x)) \quad \text{sse} \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{20}$$

Esta definição é equivalente a :

$$\begin{aligned} f \in o(g) \text{sse} \forall_{C>0} \exists_{n_0} \forall_{n\geq n_0} |\frac{f(x)}{g(x)}| \geq C \\ \text{sse} \forall_{C>0} \exists_{n_0} \forall_{n\geq n_0} |f(x)| \geq C * |g(x)| \end{aligned} \tag{21}$$

Onde a variável C representa a diferença da taxa de crescimento das duas funções.

Notação Grande-$\mathcal{O}$

Uma noção mais comum é:

$$f \in \mathcal{O}(g) \quad \text{sse} \quad \exists_{C>0}\exists_{n_0}\forall_{n\geq n_0}|f(n)| \geq C * |g(n)| \tag{22}$$

É normal escrever-se $f = \mathcal{O}(g)$ em vez de $f \in \mathcal{O}(g)$.

A diferença entre $o(g)$ e $\mathcal{O}(g)$ é que na primeira as taxas de crescimento era arbitrariamente pequena, enquanto na última a taxa de crescimento surge quantificada existencialmente.

- a função é reflexiva ($f = \mathcal{O}(f)$)
- e transitiva (se $f = \mathcal{O}(g)$ e $g = \mathcal{O}(f)$ então $f = \mathcal{O}(h)$)

$$f \in \Theta(g) \quad \text{sse} \quad f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) \tag{23}$$

TABELAS IMPORTANTES

Classe	**Nome**
$\mathcal{O}(1)$	Constante
$\mathcal{O}(\log N)$	Logarítmico
$\mathcal{O}(N)$	Linear
$\mathcal{O}(N * \log N)$	Quasi-linear
$\mathcal{O}(N^2)$	Quadrático
$\mathcal{O}(N^c)$	Polinomial
$\mathcal{O}(c^N)$	Exponencial

Table. Algumas classes de complexidade comuns

N	$\frac{1}{N} = \mathcal{O}(1)$	$\log_2(N)$	$N * \log_2 N$	N^2	N^5	2^N
1	1	0	0	1	1	2
10	0.1	3.01	30.10	100	100000	1024
100	0.01	6.02	602.06	10^4	10^10	$1.2 * 10^30$
1000	0.001	9.03	9030.89	10^6	10^{15}	$10.7 * 10^{300}$

Table. Valores que evidenciam as diferenças nas suas taxas de crescimento.

Próximo episódio

- Definições recursivas
- Análise amortizada