O que é um algoritmo?

Um **algoritmo** é um procedimento computacional bem definido que aceita um valor (ou conjunto de valores) como **input** e produz um valor (ou conjunto de valores) como **output**.

Definições alternativas:

- uma sequência de passos computacionais que transformam um input num output.

- uma ferramenta para resolver um problema computacional bem definido; problema esse que define a relação I/O pretendida.

Em geral diz-se que um determinado input é uma *instância* do problema resolvido pelo algoritmo.

Importância dos Algoritmos

Áreas típicas onde são utilizados algoritmos:

- Internet: routing, searching, etc
- Criptografia
- Investigação Operacional
- programas com mapas e.g. trajectos
- Matemática
- . . .

Exemplos de problemas complicados:

- dado um mapa em que estão assinaladas as distâncias entre diversos pontos, encontrar o trajecto de menor distância entre cada par de pontos.
- dado um conjunto de matrizes, possivelmente não quadradas e de diferentes dimensões, encontrar a forma mais eficiente de calcular o seu produto.

Importância dos Algoritmos (cont.)

Na procura de um algoritmo que resolva um determinado problema, interessa em geral encontrar um que seja *eficiente*.

Há, no entanto, problemas para os quais não se conhece uma solução eficiente. Esta classe de problemas denomina-se por NP.

Os problemas NP-completos, uma subclasse dos anteriores são especialmente interessantes porque:

- aparentemente são simples
- não se sabe se existe um algoritmo eficiente que os resolva
- aplicam-se a áreas muito importantes
- se um deles for resolúvel de forma eficiente, todos os outros o serão
- por vezes, ao resolver um problema NP-completo, contentamo-nos em encontrar uma solução que *aproxime* a solução ideal, em tempo útil.

Importância dos Algoritmos (cont.)

Por que razão é importante saber conceber e analisar algoritmos?

- Apesar de haver já um grande número de problemas para os quais se conhecem algoritmos eficiente, **nem todos estão ainda resolvidos** e documentados. Importa saber conceber novos algoritmos.

- Se a memória fosse gratuita e a velocidade ilimitada, qualquer solução correcta seria igualmente válida. Mas no mundo real, a eficiência de um algoritmo é determinante: **a utilização de recursos assume grande importância**.

  Algoritmos para um mesmo problema podem variar grandemente em termos de eficiência: podem ser diferenças muito mais importantes do que as devidas ao hardware ou ao sistema operativo.

Os **Algoritmos são uma tecnologia** que importa pois dominar, uma vez que as escolhas efectuadas a este nível na concepção de um sistema podem vir a determinar a sua validade.

Análise do Tempo de Execução – Modelo Computacional

Este tipo de análise implica decisões:

- Qual a tecnologia a utilizar?
- Qual o custo de utilização dos recursos a ela associados?

Assumiremos neste curso:

- Um modelo denominado Random Access Machine (RAM).
- Um computador genérico, uniprocessador, sem concorrência, ou seja, as instruções são executadas sequencialmente.
- Algoritmos implementados como programas de computador.

Modelo Computacional

Não especificaremos detalhadamente o modelo RAM; no entanto seremos *razoáveis* na sua utilização. . .

Um computador real não possui instruções muito poderosas, como *sort*, mas sim operações básicas como:

- aritmética
- manipulação de dados (carregamento, armazenamento, cópia)
- controlo (execução condicional, ciclos, subrotinas)

Todas executam em *tempo constante*.

Modelo Computacional

A propósito de razoabilidade. . .

$\Rightarrow$ Será a instrução *exponenciação* de tempo constante?

O modelo computacional não entra em conta com a hierarquia de memória presente nos computadores modernos (*cache*, *memória virtual*) e que pode ser relevante na análise.

O modelo RAM é no entanto quase sempre bem sucedido.

Análise do Tempo de Execução

Dimensão do input depende do problema:

- ordenação de um vector: *número de posições do vector*
- multiplicação de inteiros: *número total de bits usados na representação dos números*
- pode consistir em mais do que um item: caso de um *grafo* (V, E)

Tempo de execução: número de operações primitivas executadas

Operações definidas de forma *independente* de qualquer máquina

$\Rightarrow$ Será uma *chamada de sub-rotina* (ou função em C) uma operação primitiva?

Exemplo: Pesquisa linear num vector

		Custo	n. Vezes
1	`int procura(int *v, int a, int b, int k) {`		
2	`  int i;`		
3	`  i=a;`	c1	1
4	`  while ((i<=b) && (v[i]!=k))`	c2	m+1
5	`  i++;`	c3	m
6	`  if (i>b)`	c4	1
7	`  return -1;`	c5	1
8	`  else return i; }`	c5	1

Onde m é o número de vezes que a instrução na linha 5 é executada.

Este valor dependerá de quantas vezes a condição de guarda do ciclo é satisfeita: $0 \leq m \leq b - a + 1$.

Tempo Total de Execução

$$T(N) = c_1 + c_2(m+1) + c_3 m + c_4 + c_5$$

Para determinado tamanho fixo $N = b - a + 1$ da sequência a pesquisar – o *input* do algoritmo – o tempo total $T(N)$ pode variar com o conteúdo.

Melhor Caso: valor encontrado na primeira posição do vector

$T(N) = (c_1 + c_2 + c_4 + c_5)$, donde $T(N)$ é constante.

Pior Caso: valor não encontrado

$$T(N) = c_1 + c_2(N+1) + c_3(N) + c_4 + c_5 = (c_2 + c_3)N + (c_1 + c_2 + c_4 + c_5)$$

logo $T(N)$ é função *linear* de N

Outro exemplo: Pesquisa de duplicados num vector

		Custo	n. Vezes
1	`void dup(int *v,int a,int b) {`		
2	`  int i,j;`		
3	`  for (i=a;i<b;i++)`	c1	N
4	`  for (j=i+1;j<=b;j++)`	c2	S1
5	`  if (v[i]==v[j])`	c3	S2
6	`  printf("%d igual a %d\n",i,j);`	c4	S2
7	`}`		

onde

$$N = b - a + 1 \text{ (dimensão do input)}; \quad S_1 = \sum_{i=a}^{b-1}(n_i + 1); \quad S_2 = \sum_{i=a}^{b-1} n_i$$

e $n_i = b - i$ é o número de vezes que o ciclo interior é executado, para cada i.

Tempo Total de Execução

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2 + c_4 S_2$$

Neste algoritmo, o melhor caso e o pior caso são iguais: para qualquer vector de entrada de tamanho N, os ciclos são executados o mesmo número de vezes.

Simplifiquemos, considerando $a = 1, b = N$. Então

$$S_2 = \sum_{i=1}^{N-1} N - i = (N-1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$S_1 = \sum_{i=1}^{N-1} (N - i + 1) = (N-1)(N+1) - \frac{(N-1)N}{2} = \frac{1}{2}N^2 + \frac{1}{2}N - 1$$

$T(N)$ é pois uma função *quadrática* do tamanho do input:

$$T(N) = k_2 N^2 + K_1 N + K_0$$

Algumas Considerações

Simplificação da Análise:

- utilizámos *custos abstractos* c_i em vez de tempos concretos

E simplificaremos ainda mais: veremos que para N suficientemente grande o termo quadrático anula os restantes, logo o tempo de execução de *dup* pode ser aproximado por:

$$T(N) = kN^2$$

E de facto nem a constante é relevante face ao termo N^2. Diremos simplesmente que o algoritmo tem um tempo de execução de

$$\Theta(N^2)$$

Análise de Pior Caso e Caso Médio

("worst case" / "average case")

Análise de pior caso é útil:

- limite superior para *qualquer input* – uma garantia!
- pior caso ocorre frequentemente (e.g. pesquisa de informação não existente)
- muitas vezes caso médio é próximo do pior caso! (veremos exemplos)

Análise de caso médio ou esperado: idealmente involveria estudo probabilístico sobre a dimensão e natureza do *input* para um determinado problema (e.g. vectores com elevado grau de ordenação? Com que tamanho médio?)

Em geral assumiremos que para determinada dimensão todos os inputs ocorrem com igual probabilidade (na ordenação de um vector todos os graus de ordenação prévia são igualmente prováveis)

Notação Assimptótica – Comportamento de Funções

Tempo de execução de um algoritmo expresso como função do tamanho do input (em geral um número em $\mathbf{N} = \{0, 1, 2, \ldots\}$):

$$T(N) = k_2 N^2 + k_1 N + k_0$$

Normalmente o tempo *exacto* não é importante: para dados de entrada de elevada dimensão as constantes multiplicativas e os termos de menor grau são anulados (basta uma pequena fracção do termo de maior grau).

Então apenas a *ordem de crescimento* do tempo de execução é relevante e estudamos o *comportamento assimptótico dos algoritmos*.

Se o algoritmo A_1 é assimptoticamente melhor do que A_2, A_1 será melhor escolha do que A_2 excepto para inputs muito pequenos.

Consideraremos apenas funções *assimptoticamente não-negativas*.

Notação Θ

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $\Theta(g(n))$ como o seguinte *conjunto de funções*:

$$\Theta(g(n)) = \{f(n) \mid \text{existem } c_1, c_2, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Abuso de linguagem: $f(n) = \Theta(g(n))$ em vez de $f(n) \in \Theta(g(n))$.

Para $n \geq n_0$, $f(n)$ é igual a $g(n)$ a menos de um factor constante

Determinação da classe Θ de funções polinomiais

Basta ignorar os termos de menor grau e o coeficiente do termo de maior grau:

$$7n^2 - 2n = \Theta(n^2)$$

De facto terá de ser para $\forall n \geq n_0$:

$$c_1 n^2 \leq 7n^2 - 2n \leq c_2 n^2$$

$$c_1 \leq 7 - \frac{2}{n} \leq c_2$$

Basta escolher, por exemplo, $n \geq 10$, $c_1 \leq 6$, $c_2 \geq 8$

(constantes dependem da função)

Determinação da classe Θ

Pode-se demonstrar por redução ao absurdo que $6n^3 \neq \Theta(n^2)$.

Se existissem c_2, n_0 tais que $\forall n \geq n_0$,

$$6n^3 \leq c_2 n^2$$

$$n \leq \frac{c_2}{6}$$

o que é impossível sendo c_2 constante e n arbitrariamente grande.

Exercício:

$\Rightarrow$ mostrar para $f(n) = an^2 + bn + c$ que $f(n) = \Theta(n^2)$

Notação O ("big oh")

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $O(g(n))$ como o seguinte *conjunto de funções*:

$$O(g(n)) = \{f(n) \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq f(n) \leq cg(n)\}$$

Para $n \geq n_0$, $g(n)$ é um limite superior de $f(n)$ a menos de um factor constante

Observe-se que $\Theta(g(n)) \subseteq O(g(n))$, logo

$$f(n) = \Theta(g(n)) \text{ implica } f(n) = O(g(n)).$$

Exemplo: $3n^2 + 7n = O(n^2)$, mas também $4n - 5 = O(n^2)$ [$\Rightarrow$ porquê?]

Notação Ω

Para uma função $g(n)$ de domínio $\mathbf{N}$ define-se $\Omega(g(n))$ como o seguinte *conjunto de funções*:

$$\Omega(g(n)) = \{f(n) \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0$$

$$0 \leq cg(n) \leq f(n)\}$$

Para $n \geq n_0$, $g(n)$ é um limite inferior de $f(n)$ a menos de um factor constante

Teorema 1. *Para quaisquer duas funções* $f(n)$, $g(n)$,

$$f(n) = \Theta(g(n)) \textit{ sse } f(n) = O(g(n)) \textit{ e } f(n) = \Omega(g(n))$$

$\Rightarrow$ Qual poderá ser a utilidade deste teorema?

Abusos de Linguagem. . .

$n = O(n^2)$	$n \in O(n^2)$
$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$	$2n^2 + 3n + 1 = 2n^2 + f(n)$ com $f(n) = \Theta(n)$
$2n^2 + \Theta(n) = \Theta(n^2)$	para qualquer $f(n) = \Theta(n)$ existe $g(n) = \Theta(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$

Propriedades das Notações Θ, O, Ω

Transitividade $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implica $f(n) = \Theta(h(n))$

Reflexividade $f(n) = \Theta(f(n))$

[Ambas válidas também para O e Ω]

Simetria $f(n) = \Theta(g(n))$ sse $g(n) = \Theta(f(n))$

Transposição $f(n) = O(g(n))$ sse $g(n) = \Omega(f(n))$

Funções Úteis em Análise de Algoritmos

Uma função $f(n)$ diz-se limitada:

- *polinomialmente* se $f(n) = O(n^k)$ para alguma constante k.
- *polilogaritmicamente* se $f(n) = O(\log_a^k n)$ para alguma constante k [notação: $\lg = \log_2$].

Algumas relações úteis (a função da direita cresce mais depressa):

$$\begin{array}{rcll}
\log_b n & = & O(\log_a n) & [a, b > 1] \\
n^b & = & O(n^a) & \text{se } b \leq a \\
b^n & = & O(a^n) & \text{se } b \leq a \\
 & & & \\
\log^b n & = & O(n^a) & \\
n^b & = & O(a^n) & [a > 1] \\
n! & = & O(n^n) & \\
n! & = & \Omega(2^n) & \\
\lg(n!) & = & \Theta(n \lg n) &
\end{array}$$

Classificação de Algoritmos

Para além da classe de complexidade e das propriedades de correcção, os algoritmos podem também ser estudados – e classificados – relativamente à categoria de problemas a que dizem respeito:

- Pesquisa
- Ordenação – vários algoritmos serão estudados com detalhe
- Processamento de strings (parsing)
- Problemas de grafos – segundo capítulo do programa
- Problemas combinatoriais
- Problemas geométricos
- Problemas de cálculo numérico.
- ...

Classificação de Algoritmos (cont.)

Uma outra forma de classificar os algoritmos é de acordo com a estratégia que utilizam para alcançar uma solução:

- Incremental (iterativa) – veremos como exemplo o *insertion sort*.
- Divisão e Conquista (*divide-and-conquer*) – veremos como exemplos os algoritmos *mergesort* e *quicksort*.
- Algoritmos Gananciosos (*greedy*) – veremos alguns algoritmos de grafos e.g. Minimum Spanning Tree (Árvore Geradora Mínima).
- Programação Dinâmica – veremos o algoritmo de grafos *All-Pairs-Shortest-Path*.
- Algoritmos com aleatoriedade ou probabilísticos – veremos uma versão modificada do algoritmo *quicksort*.

Caso de Estudo 1: Algoritmo "Insertion Sort"

Problema: ordenação (crescente) de uma sequência de números inteiros. O problema será resolvido com a sequência implementada como um *vector* ("array") que será *reordenado* ($\neq$ construção de uma nova sequência).

Tempo de execução depende dos dados de entrada:

- dimensão
- "grau" prévio de ordenação

Utiliza uma **estratégia incremental** ou iterativa para resolver o problema:

- começa com uma lista ordenada vazia,
- onde são gradualmente inseridos, de forma ordenada os elementos da lista original.

"Insertion Sort"

A sequência a ordenar está disposta entre as posições 1 e N do vector A.

```
void insertion_sort(int A[]) {
  for (j=2 ; j<=N ; j++) {
    key = A[j];
    i = j-1;
    while (i>0 && A[i] > key) {
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key;
  }
}
```

Análise de Correcção – Invariante de Ciclo

Invariante de ciclo: no início de cada iteração do ciclo `for`, o vector contém entre as posições 1 e $j-1$ os valores iniciais, já ordenados.

$\Rightarrow$ Verificação da *Preservação* obrigaria a estabelecer e demonstrar a validade de um novo invariante para o ciclo *while*:

No início de cada iteração do ciclo interior, a região $A[i+2, ..., j]$ *contém, pela mesma ordem, os valores inicialmente na região* $A[i+1, ..., j-1]$. *O valor da variável* key *é inferior a todos esses valores.*

$\Rightarrow$ *Terminação* [j=n+1] corresponde ao objectivo desejado: vector está ordenado.

Análise do Tempo de Execução

```
                                          Custo        n. Vezes
void insertion_sort(int A[]) {
  for (j=2 ; j<=N ; j++) {                  c1            N
    key = A[j];                             c2          N-1
    i = j-1;                                c3          N-1
    while (i>0 && A[i] > key) {             c4           S1
      A[i+1] = A[i];                        c5           S2
      i--;                                  c6           S2
    }
    A[i+1] = key;                           c7          N-1
  }
}
```

onde $S_1 = \sum_{j=2}^{N} n_j$; $S_2 = \sum_{j=2}^{N} (n_j - 1)$ onde n_j é o número de vezes que o *teste* do ciclo `while` é efectuado, para cada valor de j

Tempo Total de Execução

$$T(N) = c_1 N + c_2(N-1) + c_3(N-1) + c_4 S_1 + c_5 S_2 + c_6 S_2 + c_7(N-1)$$

Para determinado tamanho N da sequência a ordenar – o *input* do algoritmo – o tempo total $T(n)$ pode variar com o *grau de ordenação prévia* da sequência:

Melhor Caso: sequência está ordenada à partida

$n_j = 1$ para $j = 2, \ldots, N$; logo $S_1 = N - 1$ e $S_2 = 0$;

$$T(N) = (c_1 + c_2 + c_3 + c_4 + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

$T(N)$ é função *linear* de N.

No melhor caso, temos então um tempo de execução em $\Theta(N)$.

Tempo Total de Execução

Pior Caso: sequência previamente ordenada por *ordem inversa* (decrescente)

$n_j = j$ para $j = 2, \ldots, N$; logo

$$S1 = \sum_{j=2}^{N} j = \frac{N(N+1)}{2} - 1 \quad \text{e} \quad S2 = \sum_{j=2}^{N} (j-1) = \frac{N(N-1)}{2}$$

$$T(N) = (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})N^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)N - (c_2 + c_3 + c_4 + c_7)$$

$T(N)$ é função *quadrática* de N

O tempo de execução no pior caso está pois em $\Theta(N^2)$.

$\Rightarrow$ Alternativamente, podemos dizer que o tempo de execução do algoritmo está (em qualquer caso) em $\Omega(N)$ e em $\mathcal{O}(N^2)$.