Algoritmos e Complexidade

Lic. em Engenharia Informática, 2o. ano, 2016-17

Equipa Docente:

- Jorge Sousa Pinto (coordenador) jsp@di.uminho.pt
  Atendimento: . . .
- Manuel Alcino Cunha alcino@di.uminho.pt
- José Bernardo Barros jbb@di.uminho.pt

Apoio na Web via Blackboard

Programa Resumido

I. Introdução à Análise de Correcção de Algoritmos

II. Análise de Tempo de Execução de Algoritmos

III. Estruturas de Dados Fundamentais: questões de eficiência na pesquisa

IV. Algoritmos Fundamentais sobre Grafos

V. Problemas NP-completos

Avaliação

- Época normal: $N_F = .5 * \mathit{Teste1} + .5 * \mathit{Teste2}$

- Nota mínima de 5 valores (em 20) em *ambos os testes*.

- Exame de recurso sobre toda a matéria

- Estrutura dos testes e exame:

  - Parte A: 75% valores de competências mínimas
  - Parte B: 25% valores de competências complementares

  A aprovação não implica qualquer nota mínima na parte A.

Bibliografia Recomendada

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd. edition, 2009.

[2] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th. edition, 2011.

[3] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 2nd. edition, 2013.

Importância dos Algoritmos

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**.

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

■ Importância de saber conceber e analisar algoritmos ■

- A *correcção* dos algoritmos é da maior importância. Basta pensar nas consequências bem conhecidas da existência de *bugs* . . .

- 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 no consumo de recursos: podem ser diferenças muito mais importantes do que as devidas ao hardware ou ao sistema operativo.

- Apesar de existirem já algoritmos eficientes para um grande número de problemas, há muitos outros para os quais isto não é verdade, e **novos problemas** surgem a todo o momento. Importa saber conceber novos algoritmos.

Introdução à Análise de Algoritmos

Diferentes aspectos e características dos algoritmos podem ser estudados, permitindo classificar e compará-los:

- A **correcção** de um algoritmo é fundamental. A análise da correcção de algoritmos será a primeira que abordaremos.

- Os **recursos** necessários à execução de um algoritmo:

  – memória
  – largura de banda
  – *hardware*
  – e particularmente: *tempo de execução*

  permitem comparar os algoritmos quanto à sua **eficiência**.

- A **estratégia** que um algoritmo adopta para desempenhar uma determinada tarefa ou resolver um determinado problema é muito importante na concepção de novos algoritmos.

I. Introdução à Análise de Correcção de Algoritmos

Tópicos:

- Asserções e especificação do comportamento de algoritmos. Pré-condições e pós-condições. Triplos de Hoare.
- Lógica de Hoare. Provas de correcção e geração de condições de verificação.
- Invariantes de ciclo. Provas de correcção envolvendo ciclos.
- Correcção parcial vs. correcção total. Variantes de ciclo.
- Casos de estudo: pesquisa em vectores.

Correcção de um Algoritmo

Um algoritmo diz-se **correcto** se para todos os valores dos *inputs* (variáveis de entrada) ele pára com os valores correctos dos *outputs* (variáveis de saída). Neste caso diz-se que ele **resolve** o problema computacional em questão.

Nem sempre a incorrecção é um motivo para a inutilidade de um algoritmo:

- Em certas aplicações basta que um algoritmo funcione correctamente para *alguns dos seus inputs*.
- Em problemas muito difíceis, poderá ser suficiente obter *soluções aproximadas* para o problema.

A análise da correcção de um algoritmo pretende determinar se ele é correcto, e em que condições.

Esta análise pode ser efectuada com um elevado grau de formalismo recorrendo a uma *lógica de programas*. Pode também ser conduzida de modo *semi-formal*.

Análise de Correcção

A demonstração da correcção de um algoritmo cuja estrutura não apresente ciclos pode ser efectuada por simples inspecção. Exemplo:

```
int soma(int a, int b) {
  int sum = a+b;
  return sum;
}
```

Em alguns casos a correcção advém da própria especificação. Considere-se por exemplo uma implementação recursiva da noção de *factorial*. A implementação segue de perto a definição, pelo que a sua correcção é quase imediata.

```
int factorial(int n) {
  int f;
  if (n<1) f = 1;
  else f = n*factorial(n-1);
  return f;
}
```

Análise de Correcção – Pré-condições e pós-condições

A análise de correcção dos algoritmos baseia-se na utilização de *asserções*: proposições lógicas sobre o estado actual do programa (o conjunto das suas variáveis). Por exemplo,

- $x > 0$
- $a[i] < a[j]$
- $\forall i.\, 0 \leq i < n \Rightarrow a[i] < 1000$

Pré-condição Uma propriedade que se *assume como verdadeira* no estado inicial de execução do programa, i.e., só interessa considerar as execuções do programa que satisfaçam esta condição.

Pós-condição Uma propriedade que se *deseja provar verdadeira* no estado final de execução do programa.

Análise de Correcção – Triplos de Hoare

$$\{P\}\ C\ \{Q\}$$

- C é o programa cuja correcção se considera

- P é uma *pré-condição* e Q é uma *pós-condição*

O triplo $\{P\}\ C\ \{Q\}$ é *válido* quando todas as execuções de C partindo de estados iniciais que satisfaçam P, caso terminem, resultem num estado final do programa que satisfaz Q.

Exemplo: o triplo $\{\mathbf{x == 10 \wedge y == 20}\}\ \mathbf{swap}\ \{\mathbf{y == 10 \wedge x == 20}\}$

Pode ser visto como uma *especificação* para o programa $\mathbf{swap}$;
(um *caderno de encargos*)

depois de escrito o programa, ele terá que ser *mostrado correcto* em ordem a esta especificação.

Análise de Correcção – Lógica de Hoare

Utilizaremos um sistema formal conhecido como *Lógica de Hoare* para raciocinar sobre a correcção de programas.

$$\frac{}{\{Q[e/x]\}\ x = e\ \{Q\}}$$

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1;\ C_2\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ C_t\ \{Q\} \qquad \{P \wedge \neg b\}\ C_f\ \{Q\}}{\{P\}\ \textbf{if}\ (b)\ C_t\ \text{else}\ C_f\ \{Q\}}$$

Análise de Correcção – Lógica de Hoare

Regras de Consequência:

$$\frac{\{P\}\ C\ \{Q\}}{\{P'\}\ C\ \{Q\}} \quad \text{se} \models P' \to P$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P\}\ C\ \{Q'\}} \quad \text{se} \models Q \to Q'$$

Condições de Verificação – Exemplo 1

$$\{\mathbf{x == x_0} \wedge \mathbf{y == y_0}\}\ \mathbf{swap}\ \{\mathbf{y == x_0} \wedge \mathbf{x == y_0}\}$$

Considerando uma implementação concreta de **swap**:

$$\{\mathbf{x == x_0} \wedge \mathbf{y == y_0}\}$$

$$t = x;$$
$$x = y;$$
$$y = t;$$

$$\{\mathbf{y == x_0} \wedge \mathbf{x == y_0}\}$$

Condições de Verificação – Exemplo 1

$$\{\mathbf{x == x_0 \land y == y_0}\}$$

$$\models x == x_0 \land y == y_0 \rightarrow x == x_0 \land y == y_0 \quad (CV)$$

$$
\begin{array}{ll}
 & \{x == x_0 \land y == y_0\} \\
t = x; & \\
 & \{t == x_0 \land y == y_0\} \\
x = y; & \\
 & \{t == x_0 \land x == y_0\} \\
y = t; & \\
\{\mathbf{y == x_0 \land x == y_0}\} &
\end{array}
$$

Propagamos a pós-condição para trás, até que atingimos a pré-condição. Neste ponto utilizamos uma regra de consequência para garantir que a pré-condição é mais forte do que a condição propagada (neste caso é equivalente, mas não é sempre assim: a pré-condição podia ser por exemplo $x == x_0 \land y == y_0 \land x > 0$).

Condições de Verificação – Exemplo 2

$$\{\mathbf{x} == \mathbf{x_0} \wedge \mathbf{y} == \mathbf{y_0}\}$$
$$\mathbf{sort2}$$
$$\{x \leq y \wedge ((x == x_0 \wedge y == y_0) \vee (y == x_0 \wedge x == y_0))\}$$

Considerando uma implementação concreta de **sort2**:

$$\begin{array}{l}\mathbf{if}\ (x > y)\ \{ \\ \quad t = x; \\ \quad x = y; \\ \quad y = t \\ \} \\ \mathbf{else}\ \{\ \}\end{array}$$

Propagaremos agora a pós-condição ao longo dos *dois ramos* do condicional, o que dará origem a duas utilizações da regra de consequência.

Condições de Verificação – Exemplo 2

$$\{\mathbf{x == x_0 \land y == y_0}\}$$

$\mathbf{if}\ (x > y)\ \{$

$\models x == x_0 \land y == y_0 \land x > y$

$\rightarrow y \leq x \land (y == x_0 \land x == y_0) \lor (x == x_0 \land y == y_0) \quad (CV_1)$

$$\{y \leq x \land (y == x_0 \land x == y_0) \lor (x == x_0 \land y == y_0)\}$$

$t = x;$

$$\{y \leq t \land (y == x_0 \land t == y_0) \lor (t == x_0 \land y == y_0)\}$$

$x = y;$

$$\{x \leq t \land (x == x_0 \land t == y_0) \lor (t == x_0 \land x == y_0)\}$$

$y = t;$

$\}$

$\mathbf{else}\,\{\ \ldots\ \}$

$$\{\mathbf{x \leq y \land ((x == x_0 \land y == y_0) \lor (y == x_0 \land x == y_0))}\}$$

Condições de Verificação – Exemplo 2

$$\{\mathbf{x == x_0 \wedge y == y_0}\}$$

$\mathbf{if}\ (x > y)\ \{$

$\quad \ldots$

$\}$

$\mathbf{else}\,\{$

$\quad \models x == x_0 \wedge y == y_0 \wedge \neg(x > y)$

$\quad \rightarrow x \leq y \wedge ((x == x_0 \wedge y == y_0) \vee (y == x_0 \wedge x == y_0)) \quad (CV_2)$

$$\{x \leq y \wedge ((x == x_0 \wedge y == y_0) \vee (y == x_0 \wedge x == y_0))\}$$

$\}$

$$\{\mathbf{x \leq y \wedge ((x == x_0 \wedge y == y_0) \vee (y == x_0 \wedge x == y_0))}\}$$

Análise de Correcção – Invariantes de Ciclo

No caso de algoritmos que incluam ciclos, uma prova de correcção implica uma análise de todas as suas iterações. Identificamos para isso um **invariante de ciclo** – uma propriedade que se mantém verdadeira em todas as iterações, e que reflecte as transformações de estado efectuadas durante a execução do ciclo.

$$\frac{\{I \wedge b\}\ C\ \{I\}}{\{I\}\ \mathbf{while}\,(b)\ C\ \{I \wedge \neg b\}}$$

A regra corresponde à ideia de prova indutiva no número de iterações de uma execução (que termina) do ciclo. Para provar que I é satisfeito no final da execução do ciclo (*pós-condição na conclusão*), mostra-se:

1. **Caso de base:** I é satisfeito antes da execução do ciclo (*pré-condição na conclusão da regra*);

2. **Caso indutivo:** Se I é satisfeito antes de uma iteração arbitrária, então é satisfeito depois (*premissa da regra*);

Análise de Correcção – Lógica de Hoare

$$\frac{\{I \wedge b\}\ C\ \{I\}}{\{I\}\ \textbf{while}\,(b)\ C\ \{I \wedge \neg b\}}$$

A utilização desta regra para provar a correcção de um triplo $\{P\}\ \textbf{while}\,(b)\ C\ \{Q\}$ com pré- e pós-condições arbitrárias (P, Q) dá origem, com ajuda da regra de consequência, às seguintes condições de verificação:

- **Inicialização do invariante** antes da primeira iteração: $\models P \rightarrow I$
- **Preservação do invariante**: as CVs da prova de correcção do triplo $\{I \wedge b\}\ C\ \{I\}$
- **Utilidade do invariante** se a execução do ciclo parar, o invariante é suficientemente forte para garantir a satisfação da pós-condição: $\models I \wedge \neg b \rightarrow Q$

Invariantes de Ciclo – Exemplo

Problema Divisão inteira (especificação simplificada sem vars. auxiliares).

$$\{x \geq 0 \wedge y > 0\}\ \texttt{divide}\ \{0 \leq r < y \wedge q * y + r == x\}$$

```
r = x; q = 0;
while (y <= r) {
  r = r-y;
  q = q+1;
}
```

Sejam $x == 14$ e $y == 3$:

r	q	$q * y + r$
14	0	14
11	1	14
8	2	14
5	3	14
2	4	14

Claramente $I \equiv \mathbf{0 \leq r \wedge q * y + r == x}$ é um invariante do ciclo: a **inicialização** e a **utilidade** são evidentes, e a **preservação** corresponde ao triplo:

$$\{I \wedge y \leq r\}\ \texttt{r = r-y; q = q+1}\ \{I\}$$

Condições de Verificação de Ciclos – Exemplo 1

Problema Pesquisa (ineficiente!) da última ocorrência do valor k num vector.

```
int procura(int vector[], int a, int b, int k) {
  int i, f;
  f = -1;
  for (i=a; i<=b; i++)
    if (vector[i]==k) f = i;
  return f;
}
```

Pré-condição $a \leq b$.

Pós-condição A variável f contém o valor -1 caso k não ocorra nas posições $[a \ldots b]$ do vector. Caso contrário, o valor de f será o maior no intervalo $[a \ldots b]$ tal que $vector[f] = k$.

■ Condições de Verificação de Ciclos – Exemplo 1 (cont.) ■

Invariante No início de cada iteração, $a \leq i \leq b+1$, e a variável f terá o valor -1 caso o valor k não ocorra nas posições $[a \ldots (i-1)]$ do vector. Caso contrário, f terá o maior valor no intervalo $[a \ldots i-1]$ tal que $vector[f] = k$.

Utilidade

- No final da execução do ciclo, $i = b+1$.
- Como o invariante se mantém válido ao longo de toda a execução do ciclo, então:
  - se o valor k existe entre as posições $[a..b]$ do vector, então f terá o valor da última posição onde foi encontrado;
  - caso contrário, f terá o valor -1.

Inicialização

- Antes da primeira iteração $i = a$ e $f = -1$.
- O valor k não pode existir nas posições $[a \ldots (a-1)]$ do vector: o invariante é verdadeiro.

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

Preservação

- Assume-se que o invariante é válido no início da iteração i.
- Se o valor k não estiver na posição i do array, o valor de f não será alterado. No início da iteração $i+1$ a validade do invariante mantém-se também inalterada.
- Se o valor k estiver na posição i do vector, o valor de f será alterado para i. No início da iteração $i+1$ o invariante é satisfeito: f terá o valor da última posição onde k foi encontrado.

Isto corresponde à prova de validade do triplo de Hoare $\{I \wedge b\}\ C\ \{I\}$.

Exercício: Reescrever esta prova em estilo formal, começando por escrever o invariante em linguagem matemática.

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

Para a prova em lógica de Hoare recorremos a um programa equivalente:

```
f = -1; i = a;
while (i<=b) {
  if (vector[i]==k) f = i else {} ;
  i++;
}
```

Invariante

$$
\begin{aligned}
I \quad \equiv \quad & a \leq i \leq b+1 \;\wedge \\
& (k \notin vector[a \ldots (i-1)] \;\rightarrow\; f == -1) \;\wedge \\
& (k \in vector[a \ldots (i-1)] \;\rightarrow (f \in \{a \ldots i-1\} \;\wedge\; vector[f] == k \\
& \qquad\qquad \wedge\; (\forall j \in \{a \ldots i-1\}.\; vector[j] == k \;\rightarrow\; f \geq j)))
\end{aligned}
$$

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

Pré-condição

$$P \quad \equiv \quad a \leq b$$

Pós-condição

$$\begin{aligned} Q \quad \equiv \quad & (k \notin vector[a \ldots b] \ \rightarrow \ f == -1) \ \wedge \\ & (k \in vector[a \ldots b] \ \rightarrow (f \in \{a \ldots b\} \ \wedge \ vector[f] == k \\ & \qquad\qquad\qquad \wedge \ (\forall j \in \{a \ldots b\}.\ vector[j] == k \rightarrow f \geq j))) \end{aligned}$$

Note-se que $k \in vector[a \ldots b] \ \equiv \ \exists j \in \{a \ldots b\}.\ vector[j] == k$

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

O invariante é por vezes dado como uma *anotação* no código, tal como a pré- e a pós-condição:

```
// P
f = -1;
i = a;
while (i<=b) {
  // I: a <= i <= b+1 && (...) && (...)

  if (vector[i]==k) f = i else {} ;
  i++;
}
// Q
```

Recorrendo à regra da lógica de Hoare, vamos agora escrever 4 asserções: uma pré-condição e uma pós-condição para o ciclo, e igualmente para o *corpo* do ciclo.

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

```
// P

f = -1;
i = a;

// I
while (i<=b) {
   // I && i<=b
   CORPO
   // I
}
// I && !(i<=b)

// Q
```

Este código está agora dividido em *3 blocos*, cada um com uma pré-condição e uma pós-condição, e geraremos uma CV para cada um deles.

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

Código antes do ciclo:

```
// P

      // I [a/i][-1/f]

f = -1;

      // I [a/i]

i = a;

// I
```

Condição de verificação $CV_1 : \quad P \rightarrow I[a/i][-1/f]$

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

O código que surge depois do ciclo é neste exemplo *vazio*, pelo que não há qualquer propagação a fazer:

```
// I && !(i<=b)

// Q
```

Obtemos a condição de verificação $CV_2: \quad I \wedge \neg(i \leq b) \rightarrow Q$

Condições de Verificação de Ciclos – Exemplo 1 (cont.)

Código correspondente ao corpo do ciclo:

```
// I && i <= b
if (vector[i]==k)
        // I [i+1/i][i/f]
   f = i
else
        // I [i+1/i]
   {}
// I [i+1/i]
i++;
// I
```

$$CV_3 \quad : \quad I \wedge i \leq b \wedge vector[i] == k \rightarrow I[i+1/i][i/f]$$

$$CV_4 \quad : \quad I \wedge i \leq b \wedge \neg(vector[i] == k) \rightarrow I[i+1/i]$$

Condições de Verificação de Ciclos – Exemplo 2

Problema Pesquisa da primeira ocorrência do valor k num vector.

```
procura(int vector[], int a, int b, int k) {
  int i;
  i = a;
  while ((i<=b) && (vector[i]!=k))
    i++;
  if (i>b) result = -1;
  else result = i;
}
```

Invariante $a \leq i \leq b+1$ e k não se encontra nas posições $[a \ldots i-1]$ do vector.

Utilidade O ciclo termina com $i = b+1$, ou então com $vector[i] = k$. No primeiro caso, o invariante implica que k não se encontra nas posições $[a \ldots b]$ do vector; no segundo, que não se encontra nas posições $[a \ldots i-1]$, e $vector[i] = k$.

Exercício Demonstrar a correcção do algoritmo com base neste invariante.

Análise de Correcção – Correcção Total

A noção de correcção que temos considerado é *parcial* – a validade do triplo $\{P\}\ C\ \{Q\}$ requer que Q seja satisfeito no estado final do programa, *caso a execução deste termine*.

Pode-se considerar uma noção de correcção mais forte, que exija a terminação do programa. Usaremos uma notação diferente para os triplos de Hoare correspondentes: o triplo

$$[P]\ C\ [Q]$$

é *válido* se e só se todas as execuções de C partindo de estados iniciais que satisfaçam P *terminam*, e resultam num estado final que satisfaz Q.

Num programa imperativo, as construções capazes de introduzir não-terminação são os *ciclos* e a *recursividade*. Veremos apenas como estudar a terminação dos primeiros, recorrendo à noção de *variante de ciclo*.

Um variante de ciclo é uma *expressão inteira não-negativa*, construída a partir das variáveis do programa, e cujo valor *decresce estritamente com a execução de cada iteração.*

```
int procura(int vector[], int a, int b, int k) {
  int i, f;
  f = -1;
  for (i=a; i<=b; i++)
    if (vector[i]==k) f = i;
  return f;
}
```

Variante $v = b - i$

Note-se que v é não-negativo quando são executadas iterações: $\models i \leq b \rightarrow v \geq 0$

Por outro lado, o valor de v decresce estritamente em cada iteração, uma vez que $i' = i + 1$ e $b' = b$, logo $v' = b' - i' = b - (i + 1) = b - i - 1 = v - 1 < v$.

Análise de Correcção – L. de Hoare para Correcção Total

$$\frac{[I \wedge b \wedge v == v_0]\ C\ [I \wedge v < v_0]}{[I]\ \mathbf{while}\,(b)\ C\ [I \wedge \neg b]} \qquad \models I \wedge b \rightarrow v \geq 0$$

A utilização desta regra para provar a correcção de um triplo $[P]\ \mathbf{while}\,(b)\ C\ [Q]$ dá origem às seguintes condições de verificação:

- **Inicialização do variante**: $\models I \wedge b \rightarrow v \geq 0$
- **Preservação do invariante e decréscimo do variante**: as CVs da prova de correcção de $[I \wedge b \wedge v == v_0]\ C\ [I \wedge v < v_0]$
- **Inicialização do invariante**: $\models P \rightarrow I$ (regra de consequência)
- **Utilidade do invariante**: $\models I \wedge \neg b \rightarrow Q$ (regra de consequência).

Variantes de Ciclo – Exemplo 1 (cont.)

O variante pode também ser dado como uma *anotação* no código:

```
// P
f = -1;
i = a;
while (i<=b) {
  // I: a <= i <= b+1 && (...) && (...)
  // V: b-i

  if (vector[i]==k) f = i else {} ;
  i++;
}
// Q
```

Recorrendo à regra da lógica de Hoare, vamos agora escrever 4 asserções: uma pré-condição e uma pós-condição para o ciclo, e igualmente para o *corpo* do ciclo.

Variantes de Ciclo – Exemplo 1 (cont.)

```
// P

f = -1;
i = a;

// I
while (i<=b) {
   // I && i<=b && b-i == v0
   CORPO
   // I && b-i < v0
}
// I && !(i<=b)

// Q
```

Surge agora adicionalmente a condição de verificação $CV_0 : I \wedge i \leq b \rightarrow b - i \geq 0$.

Variantes de Ciclo – Exemplo 1 (cont.)

Código correspondente ao corpo do ciclo:

```
// I && i <= b && b-i == v0
if (vector[i]==k)
        // I [i+1/i][i/f]
   f = i
else
        // I [i+1/i]
   {}
// I [i+1/i]
i++;
// I && b-i < v0
```

$$CV_3 \quad : \quad I \wedge i \leq b \wedge b - i == v_0 \wedge vector[i] == k \rightarrow (I \wedge b - i < v_0)[i+1/i][i/f]$$

$$CV_4 \quad : \quad I \wedge i \leq b \wedge b - i == v_0 \wedge \neg(vector[i] == k) \rightarrow (I \wedge b - i < v_0)[i+1/i]$$