

Ficha 1

Análise da correcção de programas imperativos

Algoritmos e Complexidade
LEI / LCC / LEF

1 Especificações

1. Descreva o que faz cada uma das seguintes funções.

```
(a) int fa (int x, int y){  
    // pre: True  
    ...  
    // pos: (m == x || m == y) && (m >= x && m >= y)  
    return m;  
}
```

```
(b) int fb (int x, int y){  
    // pre: x >= 0 && y >= 0  
    ...  
    // pos: x % r == 0 && y % r == 0  
    return r;  
}
```

```
(c) int fc (int x, int y){  
    // pre: x > 0 && y > 0  
    ...  
    // pos: r % x == 0 && r % y == 0  
    return r;  
}
```

```
(d) int fd (int a[], int N){  
    // pre: N>0  
    ...  
    // pos: 0 <= p< N && forall_{0 <= i< N} a[p] <= a[i]  
    return p;  
}
```

```
(e) int fe (int a[], int N){  
    // pre: N>0
```

```

...
// pos: forall_{0 <= i < N} x <= a[i]
return x;
}

```

```

(f) int ff (int a[], int N){
    // pre: N>0
    ...
    // pos: (forall_{0 <= i < N} x <= a[i]) &&
    //      (exists_{0 <= i < N} x == a[i])
    return x;
}

```

```

(g) int fg (int x, int a[], int N){
    // pre: N>=0
    ...
    // pos: (p == -1 && forall_{0 <= i < N} a[i] != x) ||
    //      ( (0 <= p < N ) && x == a[p])
    return p;
}

```

2. Escreva as pré e pós-condições para as seguintes funções.

- (a) A função `int prod (int x, int y)` que calcula o produto de dois inteiros.
- (b) A função `int mdc (int x, int y)` que calcula o maior divisor comum de dois números inteiros positivos.
- (c) A função `int sum (int v[], int N)` que calcula a soma dos elementos de um array.
- (d) A função `int maxPOrd (int v[], int N)` que calcula o comprimento do maior prefixo ordenado de um array.
- (e) A função `int isSorted (int v[], int N)` que testa se um array está ordenado por ordem crescente.

2 Correção

1. Para cada um dos seguintes triplos de Hoare, apresente um contra-exemplo que mostre a sua **não** validade.

- (a)
$$\frac{\{True\}}{r = x+y; \{r \geq x\}}$$
- (b)
$$\frac{\{True\}}{x = x+y; y = x-y; x = x-y; \{x == y\}}$$

- (c) $\frac{\{True\}}{x = x+y; y = x-y; x = x-y; \{x \neq y\}}$
- (d) $\frac{\{True\}}{\text{if } (x>y) \text{ } r = x-y; \text{ else } r = y-x; \{r > 0\}}$
- (e) $\frac{\{True\}}{\text{while } (x>0) \{ y=y+1; x = x-1; \} \{y > x\}}$

2. Modifique a pré-condição de cada um dos triplos de Hoare da alínea anterior de forma a obter um triplo válido.
3. Para cada uma das 4 primeiras alíneas do exercício anterior, mostre que a alteração que propôs é de facto um triplo válido.

3 Invariantes

1. Considere as seguintes implementações de uma função que calcula o produto de dois números.

<pre> int mult1 (int x, int y){ // pre: x>=0 int a, b, r; a=x; b=y; r=0; while (a!=0){ r = r+b; a = a-1; } // pos: r == x * y return r; } </pre>	<pre> int mult2 (int x, int y){ // pre: x>=0 int a, b, r; a=x; b=y; r=0; while (a!=0) { if (a%2 == 1) r = r+b; a=a/2; b=b*2; } // pos: r == x * y return r; } </pre>
---	---

- (a) Para cada um dos predicados, indique se são verdadeiros no início (Init) e preservados pelos ciclos destas duas funções.

Predicado	mult1		mult2	
	Init	Pres	Init	Pres
$r == a * b$				
$a \geq 0$				
$b \geq 0$				
$r \geq 0$				
$a == x$				
$a! = 0$				
$b == y$				
$a * b == x * y$				
$a * b + r == x * y$				

- (b) Apresente invariantes dos ciclos destas duas funções que lhe permitam provar a sua correcção (parcial).
2. Para cada uma das funções seguintes, indique um invariante de ciclo que lhe permita provar a correcção parcial. Em cada um dos casos, mesmo informalmente, apresente argumentos que lhe permitam demonstrar as propriedades (inicialização, preservação e utilidade) dos invariantes definidos.

Em alguns casos é ainda fornecido um link para um sitio onde pode *experimentalmente* as suas soluções.

- (a) Índice do menor elemento de um array. `minInd`

```
int minInd (int v[], int N) {
    // pre: N>0
    int i = 1, r = 0;
    // inv: ???
    while (i<N) {
        if (v[i] < v[r]) r = i;
        i = i+1;
    }
    // pos: 0 <= r < N && forall_{0 <= k < N} v[r] <= v[k]
    return r;
}
```

- (b) Menor elemento de um array. `minimo`

```
int minimo (int v[], int N) {
    // pre: N>0
    int i = 1, r = v[0];
    // inv: ???
    while (i!=N) {
        if (v[i] < r) r = v[i];
        i=i+1;
    }
    // pos: (forall_{0 <= k < N} r <= v[k]) &&
    //       (exists_{0 <= p < N} r == v[p])
    return r;
}
```

- (c) Soma dos elementos de um array. `soma`

```
int soma (int v[], int N) {
    // pre: N>0
    int i = 0, r = 0;
    // inv: ???
    while (i!=N) {
        r = r + v[i]; i=i+1;
    }
}
```

```

    // pos: r == sum_{0 <= k < N} v[k]
    return r;
}

```

(d) Quadrado de um número inteiro positivo. `quadrado1`

```

int quadrado1 (int x) {
    // pre: x>=0
    int a = x, b = x, r = 0;
    // inv: ??
    while (a!=0) {
        if (a%2 != 0) r = r + b;
        a=a/2; b=b*2;
    }
    // pos: r == x^2
    return r;
}

```

(e) Quadrado de um número inteiro positivo. `quadrado2`

```

int quadrado2 (int x){
    // pre: x>=0
    int r = 0, i = 0, p = 1;
    // inv: ??
    while (i<x) {
        i = i+1; r = r+p; p = p+2;
    }
    // pos: r == x^2
    return r;
}

```

(f) Tamanho do maior prefixo ordenado de um array. `maxPOrd`

```

int maxPOrd (int v[], int N){
    // pre: ??
    int r = 1;
    // inv: ??
    while (r < N && v[r-1] <= v[r])
        r = r+1;
    // pos: ??
    return r;
}

```

(g) Procura de um elemento num array. `procura1`

```

int procura1 (int x, int a[], int N){
    // pre: N>0

```

```

int p = -1, i = 0;
// inv: ??
while (p == -1 && i < N) {
    if (a[i] == x) p = i;
    i = i+1;
}
// pos: (p == -1 && forall_{0 <= k < N} a[k] != x) ||
//      ( (0 <= p < N ) && x == a[p])
return p;
}

```

(h) Procura de um elemento num array ordenado. `procura2`

```

int procura2 (int x, int a[], int N){
    // pre: N>0 &&
    //      forall_{0 < k < N-1} a[k-1]<=a[k]
    int p = -1, i = 0;
    // inv: ??
    while (p == -1 && i < N && x >= a[i]) {
        if (a[i] == x) p = i;
        i = i+1;
    }
    // pos: (p == -1 && forall_{0 <= k < N} a[k] != x) ||
    //      ( (0 <= p < N ) && x == a[p])
    return p;
}

```

(i) Procura (binária) de um elemento num array ordenado. `procura3`

```

int procura3 (int x, int a[], int N){
    // pre: N>0 &&
    //      forall_{0 < k < N-1} a[k-1]<=a[k]
    int p = -1, i = 0, s = N-1, m;
    // inv: ??
    while (p == -1 && i <= s) {
        m = (i+s)/2;
        if (a[m] == x) p = m;
        else if (a[m] > x) s = m-1;
        else i = m+1;
    }
    // pos: (p == -1 && forall_{0 <= k < N} a[k] != x) ||
    //      ( (0 <= p < N ) && x == a[p])
    return p;
}

```

(j) Soma dos primeiros números inteiros. `triangulo1`

```

int triangulo1 (int n){
    // pre: n>=0
    int r=0, i=1;
    // inv: ??
    while (i!=n+1) {
        r = r+i; i = i+1;
    }
    // pos: r == n * (n+1) / 2;
    return r;
}

```

(k) Soma dos primeiros números inteiros. `triangulo2`

```

int triangulo2 (int n){
    // pre: n>=0
    int r=0, i=n;
    // inv: ??
    while (i>0) {
        r = r+i; i = i-1;
    }
    // pos: r == n * (n+1) / 2;
    return r;
}

```

(l) Resto da divisão inteira. `mod`

```

int mod (int x, int y) {
    // pre: x>=0 && y>0
    int r = x;
    while (y <= r) {
        r = r-y;
    }
    // pos: 0 <= r < y && exists_{q} x == q*y + r
    return r;
}

```

(m) Valor de um polinómio num ponto.

```

float valor1 (float x, float coef[], int N){
    // pre: N >= 0
    float r=0; int i=0;
    // inv: ??
    while (i<N){
        r = r + pow(x,i) * coef[i];
        i = i+1;
    }
    // pos: r = sum_{0<=k<N} x^k * coef[k]
}

```

```

        return r;
    }

```

(n) Valor de um polinómio num ponto.

```

float valor2 (float x, float coef[], int N){
    // pre: N >= 0
    float r=0; int i=N;
    // inv: ??
    while (i>0){
        i = i-1;
        r = (r * x) + coef[i];
    }
    // pos: r = sum_{0<=k<N} x^k * coef[k]
    return r;
}

```

3. Para cada uma das funções da alínea anterior, indique um variante de ciclo que lhe permita provar a correcção total.

Identifique as alterações (fortalecimento) a fazer aos invariantes apresentados.