Análise de Algoritmos: Funcionamento Correto

Determinar, a priori, se um algoritmo termina corretamente.

É melhor um programa tão simples que obviamente não tem erros, do que um programa tão complexo que não tem erros óbvios.

Especificações

Para provar que um algoritmo resolve corretamente um problema, precisamos de:

- Especificação rigorosa do problema
- Descrição rigorosa do algoritmo

Muitos problemas podem ser especificados por um par:

- Entradas: dados de entrada e restrições associadas (**pré-condições**)
- Saídas: dados de saída e restrições associadas (**pós-condições**)*
    - Objetivos de maximização/minimização são redutíveis a restrições

Correção parcial e total

- Correção parcial: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então, se terminar, produz saídas corretas, i.e., que obedecem às pós condições.
- Correção total: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então termina produzindo saídas que obedecem às pós-condições.

Pré-condições e Pós-Condições

- double squareRoot(double x)
    - Pré-condições
        - x >= 0 (senão, implementação lança exceção)
    - Pós-condições
        - RESULT * RESULT == x ... a menos de um certo erro
        - RESULT >= 0
    - Algoritmo
        - Método babilónico
- int binarySearch(T a[], unsigned n, T x)
    - Pré-condições
        - Array a ordenado
        - a != NULL
        - Operadores de comparação definidos para o tipo T

        - Pós-condições
            - (0 <= RESULT < n && a[RESULT] == x) ||
            - (RESULT == -1 && x não existe em a)
- void sort(T a[], unsigned n)
    - Pré-condições
        - a != NULL
        - Operadores de comparação definidos para o tipo T
    - Pós-condições
        - Array "a" está ordenado, isto é, a[0] <= a[1] <= ... <= a[n - 1]
        - Array final tem os mesmos elementos que o array inicial
- T max(T a[], unsigned n)
    - Pré-condições
        - a != NULL
        - n > 0
        - Operadores de comparação definidos para o tipo T
    - Pós-condições
        - RESULT pertence a a
        - Nenhum elemento de a é maior que RESULT

Invariantes e variantes de ciclos

A maioria dos algoritmos são iterativos, com um ciclo principal. Para provar que um ciclo está correto, temos de encontrar um <u>invariante do ciclo</u> - uma expressão booleana (nas variáveis do ciclo) 'sempre verdadeira' ao longo do ciclo, e mostrar que:

- é verdadeira inicialmente, i.e, é implicada pela pré-condição
- é mantida em cada iteração, i.e., é verdadeira no fim de cada iteração, assumindo que é verdadeira no início de cada iteração
- quando o ciclo termina, garante (implica) a pós-condição Para provar que um ciclo termina, temos de encontrar um <u>variante do ciclo</u> - uma função (nas variáveis do ciclo)
- inteira; positiva; estritamente decrescente

Exemplo: Insertion Sort

```
for j = 2 to n:
    key = A[j]
    //insert A[j] into sorted sequence A[1 ... j - 1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Análise

- Invariante do ciclo principal [I(j)] ?
    - A[1 … j - 1] contém os elementos originais, mas ordenados (j = 2, …, n+1)
    - É válido inicialmente j=2
        - É óbvio que A[1…1] contém os elementos originais, mas ordenados
    - É mantido em cada iteração
        - Assume-se que o invariante se verifica no início da iteração
        - O alg. insere A[j] na posição certa em A[1 … j] e incrementa j
        - Logo, no fim da iteração (com o novo j), verifica-se o invariante
    - No fim do ciclo (j = n + 1), garante a pós-condição
        - Invariante refere-se a A[1 … n], ou seja, todo o array
        - Logo, implica trivialmente a pós-condição, pois é coincidente
- Variante do ciclo princial [v(j)] ?
    - n + 1 - j (j = 2, …, n + 1)
    - Inteiro, pois n e j são inteiros
    - Não negativo, pois o valor máximo de j é n + 1
    - Estritamente decrescente, pois j é sempre incrementado
- Logo, o algoritmo está correto e termina

Exemplo: Binary Search

```
BINARY-SEARCH(A, n, x)
    low <- 1
    high <- n
    while low <= high:
        mid <- |_(low + high) / 2_|
        if x == A[mid] then
            return mid
        else if x < A[mid] then
            high <- mid - 1
        else
            low <- mid + 1
    return -1
```

Análise

- Invariante do ciclo principal [I(low, high)] ?
    - x só pode existir na área de pesquisa, entre low e high
    - É válido inicialmente (low = 1, high = n), pois a área de pesquisa é todo o array
    - É mantido em cada iteração
        - Uma vez que se assume que o array está ordenado…
        - quando se recua high, excluem-se elementos > x

        - quando se avança low, excluem-se elemento < x
    - No fim do ciclo, garante-se a pós-condição
        - Se o ciclo é interrompido (A[mid] == x), garante-se a cláusula em que se encontra x
        - Se o ciclo for até ao fim, a área de pesquisa fica vazia, o que, pelo invariante, implica que x não existe em A
- Variante do ciclo princial [v(low. high)] ?
    - Largura da área de pesquisa: high - low + 1
    - Inteiro, pois low e high são inteiros
    - Não negativo, pois no pior caso é low = high
    - Estritamente decrescente, pois em cada iteração ou se aumenta low, ou se diminui high, estreitando-se a área de pesquisa
- Logo, o algoritmo está correto e termina (correção total)