>>Corretude e Completude

Dizemos que A é correto quando, dada uma instancia I de P como entrada para A, se A fornece uma saída, esta é a saída esperada pelo problema para I.

Dizemos que A é completo quando ele sempre fornece saída para qualquer que seja a instancia dada como entrada.

Daí, consideramos que A resolve P quando A é tanto correto como completo.

*Completude:

Variante de Laço:

Definição(fraca): uma formula numérica é uma variante de laço quando conseguimos demonstrar as seguintes propriedades:

- A formula depende de informações associadas do laço;
- O resultado da formula sempre sera um numero natural;
- Toda iteração que chega ao fim, o valor da expressão diminui;

Exemplo:
Algoritmo: BuscaSequencial(V, x)
Entrada: Vetor numérico V, elemento x
Saída: "V" se x pertence a V, "F" c.c.

```
para i<-1,2,3...,|V| faça
| se x = V[i] então
| | retorne V
retorne F
```

Para analisar, verificaremos 3 coisas:
(1) Validade do valor inicial;   (>=0)
(2) Diminuição;
(3) Validade do limitante;   (>=0)

(1) Qual o valor inicial do que é variável?
i = 1

(2) Quais os valores válidos para o que é variável?
1 <= i <= n

(1) e (3) com n – i, precisamos mostrar que n é limitado por i, ou seja, a partir de i, chegarmos a conclusão que n sera alcançado)

1 <= i <= n   (o que vemos em (2) diminui tudo por n)
1-n <= i – n <= 0   (agora, multiplica por -1, invertendo tudo)
n-1 >= n – i >= 0   (aqui, temos que n – i no máximo terá n – 1 iterações e no mínimo 0)

(2) Supondo que temos uma iteração que inicia e termina, com i – i'. Obviamente, na próxima repetição, i – i'' = i' + 1.

O Valor da variante na iteração é n – i' e na próxima repetição será n – (i' + 1) = n – i' – 1 < n – 1'. Isso prova que esta diminuindo, ou seja, uma hora o laço vai parar, pois o laço prossegue até i=n, zerando a variante de iteração.

Mostramos que se uma iteração chega ao final, a variante se mantêm valida. Falta mostrar que toda iteração que inicia sempre termina.

Nesse caso, temos trivialmente, pois o corpo do laço não contem outros laços sem chamadas e algoritmos.

Exemplo:

Algoritmo: BuscaBinaria(V, x)

Entrada: Vetor numérico V, numero x

Saída "V" se x pertence a V, "F" cc

Requisito: V deve estar ordenado não decrescente

```
incio <- 1
fim    <- |V|

enquanto inicio<fim faça
|  meio <- piso de (Inicio+Fim)/2;
|  se x=V[meio] entao
|  |  retorne "V"
|  senao se x<V[meio] entao
|  |  fim<-meio-1;
|  senao
|  |  inicio<-meio+1;
retorne "F"
```

Vamos analisar se o laço corre riscos de travar por tempo indeterminado

Variante para o laço:

fim – inicio

(1) e (3) inicio < fim => 0 < fim – inicio

(2) Supondo uma iteração que chega ao fim com inicio = i e fim = f.
variante de laço: f – i
na próxima iteração podemos ter dois casos:
(a) inicio = i ; fim = f';
– f alterado pela diminuição no caso de x < meio
(b) inicio = i'; fim = f;
– i alterado pelo incremento no caso de x > meio

(a) Assim, vamos mostrar que f' < f. Para isso, devemos mostrar que meio <= f e, necessariamente f' = meio – 1 < f.

Sabemos que meio = piso((i + f) / 2) e que, pelo requisito da existência do laço que i <= f, temos meio <= piso((f + f) / 2) = 2f / 2 = f. assim, temos que f '< f. Como f seria diminuído nesse caso, ele estaria sempre diminuindo, ate encontrar i em algum momento, terminando assim o laço.

(b) De maneira semelhante, temos que

meio = piso((i + f) / 2) >= piso((i + i) / 2) = 2i / 2 = i, concluindo que nesse caso o i vai aumentar, atingindo f em algum momento, terminando assim o laço.

Métrica de Eliminação

- para lidar com o caso de funções recursivas;
- { A_1, A_2, ..., A_k } são mutuamente recursivos se existe a possibilidade de alguma chamada de A_i, através do encadeamento de chamadas aos demais, resultar na nova chamada a Ai sem que a primeira tenha retornado.
- Iremos tratar apenas de recursão simples;

- Definição(fraca): Formula numérica que sempre resulta em um natural e que a cada chamada recursiva diminui.

Exemplo:
Algoritimo: MergeSort(V, p, q)
Entrada: Vetor de elementos V, índices p e q
Saída: vazia
Requisitos: Elementos de V devem ser comparáveis pela ordem <=, naturais p >= 1, q <= |V|
Objetivo: alterar V[p..q] para ordem não-decrescente

```
se p<q entao
|  meio<-piso((p+q)/2)
|  mergesort(v,p,meio)
```

```
|  mergesort(v,meio+1, q)
|  merge(v,p,meio,q)
```

Métrica: $| q - p |$

(1) e (3) $>= 0$ --trivialmente, pois são índices de vetor.

(2) Supondo uma chamada que executa a recursão (a), e consequentemente a (b), devemos mostrar que as métricas (a) e (b) diminuem.

Métrica: $q - p$

(a) p não muda, q fica q' (= meio) } $q' - p$

(b) $p = p'$(=meio+1), q não muda. } $q - p'$

(a) $q' = meio = piso((p + q) / 2) < piso((2q) / 2) = q \Rightarrow q' < q \Rightarrow q' - p < q - p$.
Assim vemos que vai diminuindo, chegando uma hora ao fim.
(b) Espelhado de (a)

*Corretude

Algoritmo: BuscaSequencial(V, x)
Entrada: Vetor numérico V, elemento x
Saída: "V" se x pertence a V, "F" c.c.

```
para i<-1,2,...,|v| faça
|  se v[i]=x entao
|  |  retorne V          --(i)
retorne F                --(u)
```

Precisamos demonstrar que qualquer que seja o caminho tomado pelo algoritmo até cada um dos pontos de saída, espera como saída a mesma resposta fornecida pelo algoritmo.
Outra forma seria argumentar que todas as instancias que esperam uma resposta em especifico devem encontrar dentro do algoritmo exatamente um caminho que fornece essa resposta.

Demonstração (Corretude de BuscaSequencial)
-Os únicos pontos de saída de algoritmo são marcados por (i) e (u).
-Qualquer execução do algoritmo que alcance (i) deve ter como fato que existe um valor de i tal que V[i] = x, por conta do condicionamento. Além disso, devido ao laço, i ∈ {1, 2, ..., |V|} que é exatamente o conjunto dos índices validos para o vetor V.
-Podemos concluir que x ∈ V para a instancia em questão.
-Qualquer execução que alcance (u) tem que ter executado todas as iterações de laco do algoritmo ate que sua condição fique falsa. A única possibilidade é ter, para cada i, o condicional sendo falso. Ou seja, Para todo i ∈ {1, 2, …, |V|}, V[i] != x. Concluímos que x !∈ V.

Exemplo:
Algoritmo: SelectionSort(V)
Entrada: Vetor V
Saída: V ordenado de forma não-descrescente
Requisito: <= de ver ordem total para elementos de V

```
para i<-1,2,...,|V| faça
|  m<-Menor(V,i,|V|);
|  V[i]<-m;
retorne V
```

Invariante de Laço:
-É uma afirmação(ou propriedade) que e necessariamente verdadeira tanto no inicio quanto no fim de cada iteração de um laco.
-O Inicio de uma iteração e o momento da execução imediatamente anterior a execução da primeira instrução do corpo do laço.
-O fim da iteração é o momento imediatamente posterior à última instrução do corpo do laço.

Para mostrar que um invariante e valido podemos argumentar em duas etapas:
(1) Validade Inicial;
(2) Preservação da Validade;

(2)supondo uma iteração que inicia e termina, considerando o valor de i = i', e supondo que no início dessa iteração a propriedade é valida, teremos:

-Supondo que o algoritmo "Menor" seja correto, m $\in$ {1, 2, ..., |V|} e V[m] é o menor elemento em V[i...|V|}, todos os valores eram maiores ou iguais aos em V[1...i'], por hipótese, temos que o novo valor de V[i'] é tal que V[1...i'] está ordenado, já que V[1...i' – 1] estava.
Além disso, como V[i'] é o menor elemento de V[i'...|V|] , temos que em V[i' + 1...|V|] todos os elementos são maiores ou iguais aos que estão em V[1...i'].
Ao final da iteração, i = i' + 1 e, com isso, concluímos que V[1..i – 1] está ordenado e V[i...|V|] são maiores.

Em suma, analisar a corretude é: investigar cada caso possível de retorno e provar que cada possibilidade dará a resposta correta.

Prova por indução:

-Quando imaginamos um algoritmo do contexto de indução de um problema, tendo esse algoritmo sido escrito de forma recursiva, nos permite raciocinar que cada chamada deve ser capaz de resolver o mesmo problema para substancias;

-Esse raciocínio pode nos facilitar a escrita de uma demonstração de corretude por indução, onde as manipulações feitas pelas chamadas recursivas são consideradas corretas dentro da hipótese indutiva.

Algoritmo: Fibonacci (n)
Entrada: natural n
Saída: n-esimo termo da sequencia de fibonacci

```
se n=0 ou n=1 entao                          --(a)
| retorne n                                  --(1)
retorne fibonacci(n-1)+fibonacci(n-2)  --(2)
```

Teorema: O algoritmo Fibonacci e correto.

Demonstração:
Por indução no parâmetro n.

Base:
n=0 o algoritmo encontra o retorno (1) e da o resultado correto
n=1 o algoritmo encontra o retorno (1) e da o resultado correto

Hipótese:
Finonacci(k) fornece a resposta correta para K = {0,1,2,...,n-1}

Passo:
Fibonacci(n): fornece a resposta correta para n >= 2.

Como n > 2, o codicional (a) é falso, de forma que o algoritmo encontra o retorno (2)

O Valor retornado é Fibonacci(n – 1) + Finonacci(n – 2). Por hipótese de indução, essas chamadas devolvem respectivamente, (n - 1) e o (n – 2) ésimos termos. Por definição, a soma desses termos é exatamente o n-ésimo termo da sequencia. Portanto, a chamada Fibonacci(n) devolve a resposta correta.
Com isso, concluímos que o algoritmo está correto.

Exemplo:
Algoritmo: BuscaBinaria(V, p, q, x)
Entrada: Vetor V, índices p e q, elemento x
Saída: "V" se x pertence a V, "F" c.c
Requisito: Vetor Ordenado.

```
se p<=q entao
|  meio←piso((p + q) / 2);
|  se V[meio]=x entao
|  |  retorne "V"                                --(1)
|  se x<V[meio] entao
|  |  retorne BuscaBinaria(V, p, meio – 1, x)    --(2)
|  retorne BuscaBinaria(V, meio + 1, q, x)       --(3)
retorne "F"                                      --(4)
```

Teorema: O Algoritmo BuscaBinaria é correto

Demonstração:
Existem quatro pontos de retorno no algoritmo:
Caso 1: Retorno (4)
Nesse caso, devemos ter uma instancia com p > q, o que define uma faixa vazia de V.
Nesse caso, certamente x não pertence a V e retorna-se F, como esperado.
Caso 2: Retorno (1)
Temos que p <= q e que V[meio] = x. Pelo valor atribuído a meio, sabemos que p <= meio <= q.
Portanto, na instancia passada temos que x pertence a V e retorna-se "V", como esperado.

Considere agora o valor n dado por q – p + 1 e suponha que BuscaBinaria fornece a resposta correta para todos os valores de k < n para algum n.

Caso 3: Retorno (2)

Temos que p <= q e que V[meio] > x. Como V está ordenado, para todo índice j de {meio,...,q} certamente temos que V[j] > x, já que p <= meio <= q. Temos que x não pertence a V[meio,...,q].

Perceba que podemos dizer que x pertence a V é equivalente a dizer que x pertence a V[p...meio-1] ou x pertence a V[meio...q].

Como a segunda parte e falsa, para as instancias consideradas, temos que x pertence a V[p..q] = V[p...meio-1].

O vetor retornado é da chamada recursiva onde a métrica para n é (meio – 1) – p + 1, que sabemos ser estritamente menor que n para a chamada atual. Por hipótese, temos que o valor retornado é o da veracidade da expressão x e V[p...meio-1] que é exatamente o valor esperado para esse caso.

Caso 4: Retorno (3)

Para alcançar esse retorno, 3 requisitos deverão ser satisfeitos:

1- Vetor esta ordenado
2- p <= q
3- x > meio

O retorno chama a função BuscaBinaria(V,meio+1,q,x).
Se temos que x > meio e o vetor esta ordenado, se x estiver em V, ele só poderia estar em V[meio + 1...q], pois só esses índices tem valor maior que meio, logo, a recursão esta correta.

Exemplo:
Algoritmo: MaximoRecursivo (V, p, q)
Entrada: Vetor V, não ordenado, índices p e q
Saída: Maior valor em V

```
r<-0
se p<=q                          --(1)
|  para i<-p,p+1,p+2...q faça
|  |  se V[i]>r entao            --(2)
|  |  |  r<-V[i]
retorne r
```

Teorema: MaximoRecursivo é correto

Descrição:

Se não atingirmos a condicional(1), temos que p > q, logo, não da pra realizar a busca visto que isso não é um intervalo.

Se atingirmos a condicional(1), temos que p <= q, e portanto, uma faixa de índices para percorrer.

Caso base:

Supondo que a primeira interação aconteça, vemos que r recebera o valor de V[i] se for maior que r. Até então, temos que em r estará o maior valor possível ate a atual iteração.

Passo indutivo:

Tendo que i >= p + 1, cada iteração verificara se V[i] > r.

Se V[i] > r, significa que o índice atual tem o valor maior do que havia antes em r.

Porem, r foi comparado com cada um dos índices anteriores, substituindo sempre que era menor pelo valor maior das iterações anteriores.

Se V[i] > r, e r atualmente possui o maior valor entre os índices anteriores, então r recebera o valor maior.

Logo, r sempre ficara com o maior valor.

Quando não satisfazer a condicional(2), então passara para a próxima, sem alterar r, mantendo o valor maior também.

Logo, em qualquer condição, o valor de r se manterá o maior, retornando apenas no final.