# ANÁLISIS Y DISEÑO DE ALGORITMOS I
# Periodo I - 2023

Jesús Aranda
jesus.aranda@correounivalle.edu.co

Universidad del Valle
Escuela de Ingeniería de Sistemas y Computación

Este documento es una adaptación del material original del profesor Oscar Bedoya

# Corrección en Computación Iterativa

---- Algoritmo iterativo

--- Correctitud de un algoritmo iterativo

---- Invariantes de ciclo

# Corrección en Computación iterativa

Una computación iterativa se caracteriza por comenzar en un estado inicial $S_0$ y transformar ese estado en otro pasando por una secuencia de estados intermedios hasta llegar a un estado final $S_j$

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_j$$

Todo estado se debe caracterizar por cumplir una condición llamada invariante de ciclo.

# Computación iterativa

## ¿Qué es una especificación?

Una especificación se define como la descripción de los siguientes parámetros:

Entrada: indica las precondiciones

Salida: indica las poscondiciones

Idea iterativa: muestra cómo deberían cambiar los estados, comenzando desde el inicial hasta llegar al final

Estados: especifica la forma de cada estado en forma de tupla, además, se muestra cuál es el invariante de estado

Estado inicial: muestra los valores que forman el estado inicial

Estado final: muestra los valores que forman el estado final

Transformación de estados: de manera formal especifica cómo se realizan, en términos generales, los cambios de un estado al siguiente

# Computación iterativa

## ¿Qué es demostrar correctitud de un algoritmo?

Un algoritmo es correcto *con respecto a* una especificación

Será correcto si para cada entrada que cumple las precondiciones, el algoritmo termina cumpliendo la poscondición

Además, para el caso específico de algoritmos iterativos, se cuenta con un método formal de probar la correctitud

# Corrección

Una especificación es la definición de un problema en términos de su precondición Q y poscondición R

Un algoritmo *A* es correcto con respecto a una especificación si para cada conjunto de valores que cumplen Q, los valores de salida cumplen R

Se denota como {Q} *A* {R}. “*A* es correcto con respecto a la precondición Q y a la poscondición R”

# Computación iterativa

Especificación para el cálculo de factorial

Entrada: N≥0

Salida: resultado=N!

Idea: Iteración

(0,1)→(1,1)→(2,2)→(3,6)→…→(N,N!)

Estados: Tupla de la forma (índice, resultado) tal que resultado=indice! (Invariante)

Estado inicial: indice=0, resultado=1

Estado final: indice=N

Transformación de estados:

(índice, resultado) → (índice +1, resultado*(índice+1) )

# Computación iterativa

Algoritmo para el cálculo de factorial

```
Factorial(int N){
        int indice=0;
        int resultado=1;
        while !(indice==N){
                indice=indice +1;
                resultado= resultado * indice;
        }
        System.out.println(resultado);
}
```

# Computación iterativa

Algoritmo para el cálculo de factorial

```
Factorial(int N){
        int indice=0;
        int resultado=1;
        while !(indice==N){
                indice=indice +1;
                resultado= resultado * indice;
        }
        System.out.println(resultado);
}
```

***¿Es correcto el algoritmo con respecto a la especificación?***

# Computación iterativa

Identifique en el algoritmo Factorial(int N) los estados inicial y final, así como la transformación dada en la especificación

¿Cómo se manejan las condiciones de entrada en el algoritmo?

# Computación iterativa

Algoritmo para el cálculo de factorial

```
Factorial(int N){
    int indice=0;
    int resultado=1;
    while !(indice==N){
        indice=indice +1;
        resultado= resultado * indice;
    }
    System.out.println(resultado);
}
```

*Condiciones iniciales*

# Computación iterativa

Algoritmo para el cálculo de factorial

```
Factorial(int N){
        int indice=0;
        int resultado=1;
        while !(indice==N){
                indice=indice +1;
                resultado= resultado * indice;
        }
        System.out.println(resultado);
}
```

Transformación de estados:

(índice, resultado) → (índice +1, resultado*(índice+1) )

# Computación iterativa

El esquema de un algoritmo iterativo es el siguiente:

$S \leftarrow S_0$

while ! isFinal(S) do

    $S \leftarrow \text{Transform}(S)$

# Computación iterativa

Cómo probar que un algoritmo iterativo A es correcto con respecto a un especificación (precondición Q, poscondición R)

1. Inicialización: Pruebe que el estado inicial $S_0$ cumple el invariante
2. Mantenimiento de Invariante: Prueba que la transformación conserva el invariante
3. Éxito: Si S es un estado final $\wedge$ se cumple el invariante $P \rightarrow R$
4. Terminación: A termina

# Computación iterativa

```
Computa (int A, int B){
        int res=0, i=1;
        while (i<=B){
                i=i+1;
                res=res + A;
        }
        System.out.println(res);
}
```

Qué calcula Computa(2,3)?

# Computación iterativa

```
Computa (int A, int B){
        int res=0, i=1;
        while (i<=B){
                i=i+1;
                res=res + A;
        }
        System.out.println(res);
}
```

Q: $A, B \in Z \wedge B > 0$

R: $res = A * B$

# Computación iterativa

```
Computa (int A, int B){
        int res=0, i=1;
        while (i<=B){
                i=i+1;
                res=res + A;
        }
        System.out.println(res);
}
```

Identifique los estados y su invariante

# Computación iterativa

```
Computa (int A, int B){
        int res=0, i=1;
        while (i<=B){
                i=i+1;
                res=res + A;
        }
}
```

Considere cada estado como el par (i,res)

(1,0)→(2,A)→(3,A+A) → … →(B+1, A+ … +A)

Invariante P: res= $\sum_{p=1}^{i-1} A$

# Computación iterativa

Probar correctitud

1. Inicialización: *Pruebe que el estado inicial $S_0$ cumple el invariante*

   El estado inicial es (1,0), Se verifica que se cumpla el invariante, se tiene que i=1.

# Computación iterativa

Probar correctitud

1. Inicialización: *Pruebe que el estado inicial $S_0$ cumple el invariante*

El estado inicial es (1,0), Se verifica que se cumpla el invariante, se tiene que i=1.

$$res = \sum_{p=1}^{i-1} A = \sum_{p=1}^{0} A = 0$$

# Computación iterativa

2. Mantenimiento de Invariante: *Prueba que la transformación conserva el invariante*

Se considera que antes de entrar el ciclo, i=k y se prueba.

Si i=k, $res = \sum_{p=1}^{k-1} A$

# Computación iterativa

```
Computa (int A, int B){
        int res=0, i=1;
        while (i<=B){
                i=i+1;
                res=res + A;
        }
        System.out.println(res);
}
```

# Computación iterativa

2. Mantenimiento de Invariante: *Prueba que la transformación conserva el invariante*

Se considera que antes de entrar el ciclo, i=k y se prueba.

Si i=k, $res = \sum_{p=1}^{k-1} A$

Al ejecutar la iteración, i=k+1:

res = res + A ← *Se toma/observa del algoritmo!!!*

res = $\sum_{p=1}^{k-1} A$ + A = $\sum_{p=1}^{k} A$

# Computación iterativa

3. Éxito: *Invariante P ∧ S es un estado final → R*

El ciclo finaliza con i=B+1, este es el valor de i en el estado final. Se calcula res.

# Computación iterativa

3. Éxito: *Invariante P $\wedge$ S es un estado final → R*

El ciclo finaliza con i=B+1, este es el valor de i en el estado final. Se calcula res:

$$res = \sum_{p=1}^{i-1} A = \sum_{p=1}^{(B+1)-1} A = \sum_{p=1}^{B} A = A * B$$

# Computación iterativa

4. Terminación: A termina

En cada iteración i aumenta, por lo que en algún momento tendrá que alcanzar el valor de B y el algoritmo terminará

# Corrección de ciclos usando invariantes

*Entendiendo que muchos algoritmos incorporan estructuras iterativas; es importante poder verificar la corrección de los ciclos a lo largo de los algoritmos.*

*Con el fin de simplificar y aplicar el concepto de invariante de ciclo para la corrección, consideramos, en particular el cumplimiento de la invariante en tres momentos:*

*Inicialización: Verificar que la propiedad (invariante) se cumple antes de la primera iteración del ciclo.*

*Mantenimiento: Verificar que si la propiedad se cumple antes de una iteración también se cumpla después de esa iteración (es decir, antes de la siguiente iteración).*

*Terminación: Verificar que al terminar el ciclo, el cumplimiento de la invariante lleve al cumplimiento del propósito (resultado esperado) del ciclo.*

# Corrección de ciclos usando invariantes

```
INSERTION-SORT(A)
for j←2 to length[A]
    do key←A[j]
       i←j-1
       while i >0 and A[i] > key
          do A[i+1]←A[i]
             i←i-1
       A[i+1]←key
```

*Invariante del ciclo: Al inicio de cada iteración (iteración j) el subarreglo A[1.. j-1] corresponde a los primeros j-1 del arreglo original ordenados ascendentemente.*

*Inicialización: Efectivamente, antes de iterar por primera vez (cuando j se inicializa en 2) el subarreglo A[1] corresponde (trivialmente al primer elemento del arreglo ordenado ascendentemente (dado que sólo es 1 elemento)*

*Mantenimiento: Cada iteración mantiene la invariante, dado que durante la iteración j el elemento A[j] es reubicado en el subarreglo ordenado A[1… j-1] de tal forma que al terminar dicha iteración todo el subarreglo A[1… j] está ordenado, cumpliendo así la invariante para el inicio de la siguiente iteración (iteración j+1).*

*Terminación: Al llevar a cabo la última iteración (j vale n, donde n es e tamaño del arreglo A) el elemento A[n] es reubicado dentro del arreglo ordenado A[1 … n-1], por lo que el arreglo A[1 … n] queda ordenado. Es decir se cumple con el propósito del ciclo y del algoritmo*

# Corrección de ciclos usando invariantes

```
// Se asume que A es un
// arreglo ordenado
// ascendentemente

BINARY-SEARCH(A, X)

n ← length[A]
if (x < A[1] or x > A[n])
    return None
else
    left = 1
    right = n
    while (left ≤ right)
        mid ← (left + right)/2
        if A[mid] = X
            return  mid
        elseif  A[mid] < X
            left ←  mid + 1
        else
            right ← mid -1
    return None
```

*Invariante del ciclo: Si X corresponde a un elemento de A (sea A[i]), entonces $left \leq i \leq right$*

*Inicialización: Efectivamente, antes de iterar por primera vez, left vale 1 y right vale n, por lo que si X hace parte de A debe estár en una posición entre left y right.*

*Mantenimiento: Cada iteración mantiene la invariante, hay dos posibilidades, si el elemento es encontrado en dicha iteración significa que i = mid (A[mid] = X) donde mid claramente está entre left y right.*

*En caso de que no se haya encontrado todavía X puede ser que X sea menor que A[mid] por lo que si X está en A debe estarlo en una posición inferior a mid (es decir i < mid) por lo que i estaría entre left y mid-1, dado que mid-1 sería el nuevo valor de right significa que la invariante se cumple para el inicio de la siguiente iteración.*

*En caso de que no se haya encontrado todavía X puede ser que X sea mayor que A[mid] por lo que si X está en A debe estarlo en una posición superior a mid (es decir i > mid) por lo que i estaría entre mid+1 y right, dado que mid+1 sería el nuevo valor de left significa que la invariante se cumple para el inicio de la siguiente iteración.*

*Terminación: Al llevar a cabo la última iteración (asumiendo que X no ha sido encontrado) la condición del ciclo no se cumple más por lo que left es mayor que right lo cual implica que no puede existir una posición i en A tal que $left \leq i \leq right$ y por ende X no puede estar en A. Es decir, al no ser encontrado X en A en el ciclo, implica que X no hace parte de A. Cumpliendo con el propósito del ciclo.*