# Diseño de algoritmos iterativos

Facultad de Informática - UCM

27 de agosto de 2022

- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
- Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006

Capítulos 2, 3, y 4 de ambos libros.

- Complejidad de algoritmos iterativos:
  - Ejercicios resueltos: 3.21 a), 3.23 a), 3.25
  - Ejercicios propuestos: 3.12, 3.13
- Verificación de algoritmos iterativos
  - Ejercicios resueltos: 2.7, 2.8, 2.9, 2.13, 2.15
- Derivación de algoritmos iterativos
  - Ejercicios resueltos: 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.22, 4.23, 4.24, 4.25, 4.27

# Objetivos

- Reglas para comprobar que un programa es correcto respecto a su especificación (verificación).
- Cómo implementar un programa correcto por construcción (derivación).
- Ver soluciones de problemas iterativos típicos para conocer diversas formas de solución.

# Verificación

Verificar es demostrar que las instrucciones de un algoritmo son correctas, es decir, que para una entrada válida (precondición) producen el resultado esperado (postcondición).

Ejemplo: Problema: Intercambiar el valor de dos variables

Especificación:

| |
|---|
| x,y : entero |
| $P : \{x = X \wedge y = Y\}$ |
| Intercambiar |
| $Q : \{x = Y \wedge y = X\}$ |

¿ Cuales de los siguientes algoritmos son correctos?

| | | | |
|---|---|---|---|
| aux = x;<br>x = y;<br>y = aux; | x = y;<br>y = x; | x = x - y;<br>y = x + y;<br>x = y - x; | x = x + y;<br>y = x - y;<br>x = y - x; |

# Verificación

- Para verificar un algoritmo se definen predicados intermedios entre cada instrucción elemental, llamados *aserciones* o *asertos*:

$$\{R_0\}A_1\{R_1\}; \ldots; \{R_{n-1}\}A_n\{R_n\}$$

- Si para cada instrucción $A_k$ se satisface $\{R_{k-1}\}A_k\{R_k\}$ y $P \Rightarrow R_0$ y $R_n \Rightarrow Q$ entonces se satisface $\{P\}A\{Q\}$.
- La semántica del lenguaje define para cada tipo de instrucción del lenguaje las reglas que determinan si se satisface $\{R_{k-1}\}A_k\{R_k\}$ (reglas de verificación).

# Verificación

Axioma de asignación Para toda variable $x$, toda expresión válida del mismo tipo $E$ y todo predicado $R$, la especificación:

| | |
|---|---|
| $P : \{Dom(E) \wedge \{R_x^E\}$<br>$\mathrm{x = E}$<br>$Q : \{R\}$ | es correcta. |

$P$ se denomina precondición más débil (*pmd*).

- $Dom(E)$ el conjunto de todos los estados en los que la expresión $E$ está definida.
- $R_x^E$ el predicado resultante de sustituir toda aparición de $x$ por $E$ en el predicado $R$.

Ejemplo: Suponiendo $x$ entero determina la precondición más débil, $P$, que safisfaga la especificación:

| | |
|---|---|
| $P :$<br>$\mathrm{x = x - 2}$<br>$Q : \{x \geq 0\}$ | $Q_x^{x-2} : \{x - 2 \geq 0\} \equiv \{x \geq 2\}$<br>$\mathrm{x = x - 2} \quad \uparrow$<br>$Q : \{x \geq 0\}$ |

## Regla de inferencia de la asignación

La especificación $\{P\}\ \mathrm{x} = \mathrm{E}\ \{Q\}$ es correcta si $P \Rightarrow Q_x^E$.

## Regla de inferencia de la composición secuencial

La especificación $\{P\}\ A_1; A_2\ \{Q\}$ es correcta si existe un predicado $R$ tal que las especificaciones $\{P\}\ A_1\ \{R\}$ y $\{R\}\ A_2\ \{Q\}$ son correctas.

Ejemplo: Suponiendo $x$, $y$ enteros, calcula el predicado $P$ más débil que satisfaga la especificación:

| | |
|---|---|
| $P$ : | $R_2 \equiv R_{1_x}^{x*x} : \{x * x + 1 > 0\} \equiv \{true\}$ |
| $\mathrm{x = x * x}$ | $\mathrm{x = x * x} \quad \uparrow$ |
| | $R_1 \equiv Q_x^{x+1} : \{x + 1 > 0\}$ |
| $\mathrm{x = x + 1}$ | $\mathrm{x = x + 1} \quad \uparrow$ |
| $Q : \{x > 0\}$ | $Q : \{x > 0\}$ |

## Regla de inferencia de la composición alternativa (I)

La especificación

$$\boxed{\begin{array}{l} \{P\} \\ \quad \texttt{if}\ (B)\ A_1\ \texttt{else}\ A_2; \\ \{Q\} \end{array}}$$

es correcta si las especificaciones

$$\boxed{\begin{array}{l} \{P \wedge B\} \\ \quad A_1 \\ \{Q\} \end{array}} \quad \text{y} \quad \boxed{\begin{array}{l} \{P \wedge \neg B\} \\ \quad A_2 \\ \{Q\} \end{array}}$$

son correctas. La *pmd* es $B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q))$

Ejemplo:

| | | |
|---|---|---|
| $P$ : | $P \wedge x \geq 0 \Rightarrow R_1$ | $P \wedge \neg(x \geq 0) \Rightarrow R_2$ |
| if (x <= 0) y = x; | $R_1 \equiv Q_y^x : \{x = Y\}$ | $R_2 \equiv Q_y^{-x} : \{-x = Y\}$ |
| else y = -x; | y = x $\uparrow$ | y = -x $\uparrow$ |
| $Q : \{y = Y\}$ | $Q : \{y = Y\}$ | $Q : \{y = Y\}$ |

$pmd \equiv (x \geq 0 \wedge x = Y) \vee (\neg(x \geq 0) \wedge -x = Y)$

## Regla de inferencia de la composición iterativa

La especificación $\boxed{\{P\}\ \texttt{while}\ (B)\ \texttt{do}\ A\ \{Q\}}$ es correcta si existe un predicado $I$ que llamaremos invariante y una función $t$ dependiente de las variables que intervienen en el proceso y que toma valores enteros, que llamaremos función cota, de forma que:

1. $P \Rightarrow I$
2. $\{I \wedge B\}\ A\ \{I\}$
3. $I \wedge \neg B \Rightarrow Q$
4. $I \wedge B \Rightarrow t > 0$
5. $\{I \wedge B \wedge t = T\}\ A\ \{t < T\}$

Existen muchos predicados invariantes, se ha de elegir uno que nos permita verificar las 5 condiciones de corrección del bucle, esto es

- Lo suficientemente fuerte para $I \wedge \neg B \Rightarrow Q$
- Lo suficientemente débil para $P \Rightarrow I$.

El invariante es un predicado que describe todos los estados por los que atraviesa el cómputo realizado por el bucle, observados justo antes de evaluar la condición $B$ de terminación.

Ejemplo:

$fact = 1;\ i = 2;$
$P : \{i = 2 \wedge fact = 1\}$
while $(i <= n)\{$
$\quad fact = fact * i;\ i = i + 1;$
$\}$

valores de las variables antes de evaluar la condición $i < n$.

| estado | $fact$ | $i$ |
|---|---|---|
| $P_0$ | 1 | 2 |
| $P_1$ | 2 | 3 |
| $P_2$ | 6 | 4 |
| $P_3$ | 24 | 5 |
| ... | | |

$Q : \{fact = \prod k : 0 < k \leq n : k\}$

¿ Que relaciones entre las variables se mantienen durante la ejecución del bucle?

Un invariante del bucle es:

$\boxed{I : \{0 \leq i \leq n + 1 \wedge fact = \prod k : 0 < k < i : k\}}$

Ejemplo:

$i = 0;\ q = 0;\ p = 1;$
$P : \{i = 0 \wedge q = 0 \wedge p = 1\}$
while $(i < n)\{$
    $q = q + p;\ p = p + 2;\ i = i + 1;$
$\}$
$Q : \{i = n \wedge q =? \wedge p =?\}$

valores de las variables antes de eval. cond. $i < n$.

| estado | $i$ | $q$ | $p$ |
|---|---|---|---|
| $P_0$ | 0 | 0 | 1 |
| $P_1$ | 1 | 1 | 3 |
| $P_2$ | 2 | 4 | 5 |
| $P_3$ | 3 | 9 | 7 |
| ... | | | |

¿ Que relaciones entre las variables se mantienen durante la ejecución del bucle?

Un invariante del bucle es: $\boxed{I : \{0 \leq i \leq n \wedge q = i^2 \wedge p = 2i + 1\}}$

# Verificación

La *función cota* o *función limitadora* es una función $t : estado \to \mathcal{Z}$ positiva que decrece cada vez que se ejecuta el cuerpo del bucle.

La función cota garantiza la terminación del bucle

Para encontrar una función cota se observan las variables que son modificadas por el cuerpo $A$ del bucle, y se construye con ellas una expresión entera $t$ que decrezca

Ejemplo: cálculo del cuadrado de un número.
Las variables $i$, $p$ y $q$ crecen, $n$ se mantiene inalterable, por lo tanto $n - i$ decrece.
Además, la condición del bucle $i \leq n$ garantiza que la función es positiva.

La función cota es : $t = n - i$.

# Ejemplos

Especifica una función que calcule la suma de las componentes de un vector.

Dada la siguiente implementación indica un invariante del bucle.

```
int suma(vector<int> const& v){
  int n = v.size();  x = 0;
  while (n != 0)
  {
    x = x+v[n-1];
    n = n-1;
  }
  return x;
}
```

# Ejemplos

Especificación.

$\{\mathbf{v.size}() \geq \mathbf{0}\}$
*suma* $(\mathbf{vector{<}int{>}}\ v)$ *dev* $\mathbf{int}\ x$
$\{x = (\Sigma i : 0 \leq i < \mathbf{v.size}() : v[i])\}$

```
int suma(vector<int> const& v){
  int n = v.size(), x = 0;
  while (n != 0)
  {
    x = x+v[n-1];
    n = n-1;
  }
  return x;
}
```

Invariante:
$I \equiv \{0 \leq n \leq \mathbf{v.size}() \wedge x = (\boldsymbol{\Sigma} i : n \leq i < \mathbf{v.size}() : v[i])\}.$

# Ejemplos

- Indicar un invariante para el siguiente bucle suponiendo $x$, $y$, $n$: enteros.
  $\{n \geq 0\}$

```
int x = 0, y = 1;
while (x != n)
{
  x = x+1;
  y = y+y;
}
```

  $\{y = 2^n\}$
- Invariante: $I \equiv \{0 \leq x \leq n \wedge y = 2^x\}$

# Derivación

**Derivar**: construir las instrucciones de un algoritmo a partir de su especificación asegurando su corrección.

Los algoritmos se ajustan al esquema:

$\{P\}$
$A_0$ `(Inicializacion)`
$\{I, Cota\}$
**while** `(B) {`
  $\{I \wedge B\}$
  $A_1$ `(Restablecer)`
  $\{R\}$
  $A_2$ `(Avanzar)`
  $\{I\}$
`}`
$\{Q\}$

- $A_0$ Hace que el invariante se cumpla inicialmente $A_2$ hace que la cota decrezca.
- $A_1$ mantiene el invariante a cierto.

- **Pasos** para construir un algoritmo con bucle:
  1. Diseñar el invariante y la condición del bucle sabiendo que se tiene que cumplir:
     $I \wedge \neg B \Rightarrow Q$
  2. Diseñar $A_0$ para hacer el invariante cierto: $\{P\}A_0\{I\}$
  3. Diseñar la función cota, $C$, de tal forma que: $I \wedge B \Rightarrow C \geq 0$.
  4. Diseñar $A_2$ y el predicado $R \equiv pmd(A_2, I)$.
  5. Diseñar $A_1$ para que se cumpla: $\{I \wedge B\}A_1\{R\}$.
  6. Comprobar que la cota realmente decrece:
     $$\{I \wedge B \wedge C = T\}A_1, A_2\{C < T\}$$

Evitar bucles anidados mediante el uso de variables acumuladoras.

Contar el número de elementos de un vector tales que su valor es mayor o igual que todos los valores que le preceden en el vector

P: $\{ \ 0 < v.size \ \}$
picos(vector<int> v) dev int n
Q: $\{ \ n = \#k : 0 \leq k < v.size : espico(v, k) \ \}$

donde: $espico(v, i) \equiv \forall k : 0 \leq k < i : v[i] >= v[k]$

# Derivación de algoritmos iterativos típicos

```
int numPicos(std::vector<int> const& v) {
  int n=1;  int imax = 0;
  for (int i = 1; i < v.size(); ++i)
     // invariant 1<=i<=v.size && 0<=imax<i
     // invariant forall l: 0<=l<i : v[imax]>=v[l]
     // invariant n= # k : 0 <= k < i : espico(v,k)
  {
     if (v[i] >= v[imax])
     {
        imax = i;
        n = n+1;
     }
  }
}
```

La moda es el valor que más veces aparece repetido en una colección.

Derivar el siguiente algoritmo:

P: $\{a.size > 0\ \}$
moda (vector<int> a) dev int m
Q: $\{\ \forall k : 0 \leq k < a.size : ig(a, v[k]) \leq ig(a, m)\ \}$

donde: $ig(a, m) \equiv \#x : 0 \leq x < a.size : m == v[x]$

En las siguientes implementaciones suponemos que la moda es única en el vector. Si pudiesen existir varios valores que apareciesen el número máximo de veces habría que guardarlos en un vector.

1. Primera solución ordenar el vector

```
int moda1 (std::vector<int> & v){
  ordenar(a);
  int m = a[0]; int f = 1; int fm = 1;
  for (int i = 1;i < a.size; ++i)
  // inv. forall k: 0<=k<i: ig(a,v[k]) <= ig(a,m)
  // inv. fm = # k: 0<=k<i : v[k] == m
  // inv. f = # k: 0<=k<i : v[k] = a[i-1]
  // inv. 0 <= i <= a.size
    if (a[i] == a[i-1]){
      f := f+1;
      if (fm < f) { fm:=f; m := a[i]; }
    }
    else f := 1;
  return m;
}
```

# Derivación de algoritmos iterativos típicos

1. Segunda solución, utilizar vector para acumular los valores. Se crea un vector `a` con las apariciones de cada elemento y después se busca el máximo del vector `a`.

```
int moda2 (std::vector<int> const& v){
  //Busca el maximo del vector para fijar
  //el tamano del vector de apariciones
  int maxi = v[0];
  for (int i = 1; i < v.size; ++i)
    //inv. maxi = max k: 0<=k<i : v[k]
    maxi = std::max(maxi, v[i]);
  std::vector<int> a(maxi+1);
  // vector con las apariciones de los elementos
  for (int k = 0; k < v.size; ++k)
//inv. forall x:0<=x<a.size:a[x]=(#y:0<=y<k:v[y]=x)
    a[v[k]] = a[v[k]] + 1;
```

1. Segunda solución, utilizar vector para acumular los valores. Se crea un vector `a` con las apariciones de cada elemento y después se busca el máximo del vector `a`.

```
  // busca el maximo del vector de apariciones
  int moda = a[0];
  for (int x = 1; x < a.size; ++x) {
    //inv. moda = max k: 0<=k<i: a[k]
    if (moda < a[x]) {
      moda = a[x];
      }
  }
  return moda;
}
```

Algoritmos con ventana.

Dado un vector v encontrar el intervalo de tamaño m cuya suma sea máxima.

Derivar el siguiente algoritmo:

P: $\{ 1 \leq m \leq v.size \}$
Ventana (vector<int> v, int m) dev (int s)
Q: $\{ s = max\ t : 0 <= t < v.size : (\sum k : t \leq k < t + m : v[k]$

# Derivación de algoritmos iterativos típicos

```
int ventana (vector<int> const& v, int m) {
  int sumParcial = 0;
  for (int i = 0; i < m; ++i)
    // invariant 0 <= i <= m
    // invariant sumParcial==Sum k: 0<=k<i : v[k]
    sumParcial = sumParcial + v[i];
  int s = sumParcial;
  for (int j = 0; j < v.size()-m; ++j)
  { // invariant 0 <= j <= v.size-m
    // invariant sumParcial==Sum k: j<=k<j+m : v[k]
    // inv s=max k: 0<=k<j : (Sum x: k<=x<k+m : v[x])
    sumParcial = sumParcial + v[j+m];
    sumParcial = sumParcial - v[j];
    if (sumParcial > s) {s = sumParcial;}
  }
  return s;
}
```

Segmento de longitud máxima.

Indicar la longitud de la subsecuencia mas larga de ceros de una cadena numérica.

P:{ $v.size > 0$ }
segmentoMax (vector<int> v) dev int l
Q:{ $l = max\ i, j : 0 <= i <= j < v.size \wedge ig0(v, i, j) : j - i$ }

donde: $ig0(v, i, j) \equiv \forall x : i \leq x \leq j : v[x] = 0$

```
{
  int longMax = 0; int longActual = 0;
  for (int i = 0; i < v.size(); ++i) {
  // inv. longMax=max p,q:0<=p<=q<i && ig0(v,p,q):q-p
  // inv. longMax=min p:0<=p<i && ig0(v,p,i):i-p
    if (v[i]==0) {  // elemento bueno
      ++longMax;
      if (longMax < longActual) {
        longMax = longActual;
      }
    }
    else  {longActual = 0;}
  }
  return longMax;
}
```

## Algoritmo de partición.

Dado un vector definido entre dos índices $a$ y $b$, el problema pide separar las componentes del vector, dejando en la parte izquierda aquellos valores que sean menores o iguales que el valor de la componente $v[a]$ y en la parte derecha aquellos valores que sean mayores o iguales que el valor de dicha componente. El valor de $v[a]$ debe quedar separando ambas partes.
Este algoritmo se utiliza en la implementación del algoritmo de ordenación rápida (*quicksort*).

P: $\{ \ 0 \leq a \leq b < v.size \ \}$
particion (vector<int> v, int a, int b) dev int p
Q: $\{ \ 0 \leq a \leq p \leq b < v.size \wedge \forall x : a \leq x < p : v[x] \leq v[p] \wedge$
$\forall y : p+1 \leq y \leq b : v[y] \geq v[p] \ \}$

Algoritmo de partición.

```
int i = a+1;
int j = b;
while (i <= j)
{ // inv. forall x: a<=x<i : v[x] <= v[a]
  // inv. forall y: j<=y<=b : v[y] >= v[a]
  if (v[i] > v[a] && v[j] < v[a])
  {
    std::swap(v[i],v[j]);
    i=i+1; j=j-1;
  }
  else if (v[i] <= v[a]) {i=i+1;}
  else if (v[j] >= v[a]) {j=j-1;}
}
p= j;
std::swap(v[a],v[p]);
}
```

Mezcla de dos vectores ordenados. Dados dos vectores ordenados obtener un tercer vector con la mezcla ordenada.
Un algoritmo semejante, pero sobre el mismo vector se utiliza en la implementación del algoritmo de ordenación por mezclas (*mergesort*).

P: $\{\ v1.size > 0 \wedge v2.size > 0 \wedge ord(v1) \wedge ord(v2)\ \}$
mezcla (vector<int> v1, vector<int> v2) dev vector<int> v
Q: $\{\ v1.size > 0 \wedge v2.size > 0 \wedge$
$\quad \forall u :: 0 \leq u < size - 1 : v[u] < v[u+1] \wedge$
$\quad \forall k :: 0 \leq k < v1.size : in(v1[k], v)$
$\quad \forall k :: 0 \leq k < v2.size : in(v2[k], v)$
donde:
$\quad ord(v) \equiv \forall u, w :: 0 \leq u < w < v.size : v[u] < v[w]$
$\quad in(x, v) \equiv \exists k : 0 \leq k < v.size : v[k] = x$

# Derivación de algoritmos iterativos típicos

```
int i = 0, j = 0; int k = 0;
while (i<v1.size && j < v2.size)
{ // inv. forall u::0<=u<k: v[u] < v[u+1]
  // inv. forall u::0<=u<i: in(v1[u], v)
  // inv. forall u::0<=u<j: in(v2[u], v)
  if (v1[i] < v2[j]) {v[k] = v1[i]; i = i+1;}
  else if (v2[j] < v1[i]) {v[k] =v2[j]; j=j+1;}
  else {v[k] = v1[i]; i=i+1; j=j+1;}
  k = k+1;
}
while (i < v1.size)
{ // inv. forall u::0<=u<k: v[u] < v[u+1]
  // inv. forall u::0<=u<i: in(v1[u], v)
  v[k]= v1[i]; i=i+1; k=k+1;
}
while (j < v2.size)
{  v[k]= v2[j]; j=j+1; k=k+1;  }
}
```

Algoritmos de matrices.

Comprobar si los elementos de una matriz son positivos:

P: { true }
MatricesPositivas(vector<vector<int>> m) dev bool s
Q: { $s \equiv \forall i, j :: 0 \leq i < m.size \wedge 0 \leq j < m[0].size : m[i, j] > 0$

- Se utilizan dos bucles anidados en el algoritmo. Para cada bucle se define su invariante.

```
{ int k1 = 0; bool s = true;
  for (int k1 = 0; s && k1 < m.size(); ++k1) {
  // inv. 0 <= k1 <= m.size
  // inv. s==forall i,j:0<=i<k1&&0<=j<m.size:m[i,j]>0
    for (int k2 = 0; s && k2<m[0].size(); ++k2){
    // inv.0 <= k2 <= m[0].size
    //inv. 0 <= k1 < m.size
    //inv. s == forall j:0<=j<k2:m[k1,j]>0
    {
      s = m[k1,k2]>0;
    }
  }
}
```