Universidad Nacional Autónoma de México

Facultad de Ciencias

Departamento de Matemáticas

México, Septiembre 2020.

Introducción al Análisis de Algoritmos

Notas de Clase

(Primera Parte, Segunda Versión)

Dra. María De Luz Gasca Soto

Licenciatura en Ciencias de la Computación,
Departamento de Matemáticas,
Facultad de Ciencias, UNAM.

Prefacio

Estas notas forman parte del material que se revisa en el curso de Análisis de Algoritmos I que se imparte en la Facultad de Ciencias de la UNAM, para la Licenciatura en Ciencias de la Computación.

He tenido la oportunidad de impartir este curso los últimos años y he estado preparando y corrigiendo las presentes notas de clase para el curso, de las cuales esta resulta ser la segunda versión.

Considero que las áreas de Análisis, Diseño y Justificación de algoritmos debe ser tomado más en serio por las personas que de una u otra manera diseñan programas o bien están involucradas con la computación.

El presente trabajo, pretende dar una panorámica general de lo que es el análisis de algoritmos. Enfatizando que el análisis, diseño y justificación de algoritmos se puede realizar de manera formal usando como herramienta a la Inducción Matemática.

La bibliografía básica, para el material presentado en estas notas, está basada en los libros:

- **Udi Manber** [13]
  Introduction to Algorithms. A Creative Approach.
- **J. Kingston** [12]
  Algorithms and Data Structures: Design, Correctness and Analysis.
- **R. Neapolitan & K. Naimipour** [16].
  Foundations of Algorithms.

Dra. María De Luz Gasca Soto
Profesor Asociado, T.C.
Departamento de Matemáticas
Facultad de Ciencias, UNAM.

Índice general

Capítulo 4

Justificación de Algoritmos

Existen muchas razones para estudiar la Integridad de los Algoritmos (*Algorithm Corretness*), una de ellas es el mejoramiento de la calidad de programas a diseñar. Aunque resulta verdad que la integridad de cada algoritmo depende de las propiedades específicas del problema, se debe tener una estrategia para justificar tales propiedades. Probar que un algoritmo es correcto significa revelar o dar a conocer tal estrategia, la cual puede utilizarse para comparar, mejorar o desarrollar algoritmos.

El estudio de la integridad de algoritmos es conocido como **Semántica Axiomática**, *Axiomatic Semantic*, y las ideas básicas fueron dadas por Floyd [8] y Hoare [10]. Es posible, usando metodos de la semántica axiomática, probar que un programa es correcto tan rigurosamente como demostrar un teorema en Matemáticas.

Se dice que un **algoritmo es correcto** si garantiza la creación de una respuesta correcta para cada ejemplar del problema.

Especificar un problema puede ser una tarea muy díficil, se requiere de una gran precisión para hacerlo. Por ejemplo, un conjunto vacío *no* tiene elemento mínimo, así que la especificación de encontrar el elemento mínimo de un conjunto puede fallar si el conjunto resulta ser vacío.

Para facilitar la definición de especificaciones precisas se dan dos expresiones lógicas denominadas **Precondiciones**, PreC, y **Postcondiciones**, PostC. Una precondición asume algún valor de verdad con el cual debe iniciar el proceso. Una postcondición es un estado que garantiza una verdad acerca del resultado.

Ejemplo 4.1 Encontrar el mínimo en un conjunto S de números naturales.
PreC: El conjunto S es no vacío; el conjunto S es finito;
PostC: m es un mínimo elemento del conjunto S. Lo que resulta equivalente a:

$$(\exists\ x \in S \text{ tal que } m = x)\ \ \&\ \ (\forall\ x \in S, m \leq x)$$

Asumiendo que todos los ejemplares son no vacíos no habrá razón para preocuparse por lo que pueda suceder cuando llega un conjunto vacío. Resultará responsabilidad del usuario dar únicamente ejemplares que satisfagan la precondición, para de esta manera garantizar un resultado correcto.

4.1. Algoritmos Recursivos.

En esta sección revisaremos estrategias basadas en la inducción matemática para demostrar (justificar) que un algoritmo recursivo es correcto, lo haremos por medio de ejemplos.

Un **algoritmo recursivo** es aquel que se usa a sí mismo para resolver un problema. El secreto para solucionar un problema utilizando un algoritmo recursivo es resolver el problema usando uno o más resultados del problema para ejemplares diferentes y, generalmente, más pequeños.

Cuando se demuestra que un algoritmo recursivo encuentra la solución correcta para algún ejemplar, se requiere asumir que encuentra la solución correcta para ejemplares más pequeños. Lo cual sugiere que se debe usar la inducción sobre el tamaño del ejemplar para probar que el algoritmo es correcto.

Se dice que un **programa o definición es recursivo** si se refiere directa o indirectamente a sí mismo. Los programas recursivos definen procesos y las definiciones recursivas conjuntos de objetos.

Ejemplo 4.2 Definición Recursiva de Identificadores. El conjunto de todos los identificadores de un lenguaje de programación pueden ser especificados de la siguiente manera:

un identificador es una letra o
un identificador seguido de una letra o un dígito.

Por ejemplo, *T, T3, T0, T3M, T04, T0M* son identificadores.

La definición del ejemplo anterior puede ser más explícita:

Ejemplo 4.3 Definición Recursiva para un Conjunto de Identificadores.
El conjunto válido de identificadores construye de la siguiente forma:

Todas las cadenas constituidas de una letra están en el conjunto.
Si algún elemento x está en el conjunto, entonces también están
los elementos $x\ell$ y $x\delta$, donde ℓ es una letra y δ un dígito.

Consideremos el problema de ordenar una secuencia en forma ascendente, para definir recursivamente el concepto de secuencia ordenada.

Ejemplo 4.4 Secuencia Ordenada.
Una secuencia se encuentra ordenada

si consiste de un solo elemento o
si el elemento más pequeño tiene el menor índice y
el resto de la secuencia está ordenada.

A continuación se da una definición más explícita para construir el conjunto de todas las secuencias ordenadas.

Ejemplo 4.5 Conjunto de Secuencia Ordenada.
Colocar en el conjunto todas las secuencias que contienen solo un elemento.
Si x está en el conjunto, entonces agregar todas las secuencias αx,
donde α es un elemento menor o igual que cualquier elemento en x.

4.1.1. Números de Fibonacci.

Listado 12 Algoritmo de Fibonacci.

```
Calcula_Fibonacci{
//  PreC: n es un n\'umero entero, n>= 0.
      x <--  AFibonacci(n)
// PostC: x  es el  n-esimo numero de Fibonacci. }

 AFibonacci (n: integer ): integer;
 {
  If   (n <= 1)  Then  Return  n ;
     Else
       Return  AFibonacci(n-1) + AFibonacci(n-2);
 }
```

Se definen los números de Fibonacci como:

$$F(0) = 0; \qquad F(1) = 1; \qquad F(n) = F(n-1) + F(n-2), \ \forall\, n \geq 2.$$

Los primeros número de Fibonacci son:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

El listado 12 describe un algoritmo recursivo para calcular el n-ésimo número de Fibonacci.

4.1.2. Factorial de n

Problema: Calcular el factorial de un número n, el cual se define como:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = \prod_{k=1}^{n} k \qquad \text{o bien} \qquad n! = n \cdot (n-1)!$$

Algunos ejemplos de Factorial: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$;

$$9! = 9 \times 8 \times 7 \times 6 \times 5! = 3024 \times 120 = 362,880.$$

El Listado 13 describe un algoritmo recursivo para calcular el Factorial de n.

Listado 13 Algoritmo que calcula del Factorial de n.

```
Calcula_Factorial
//  PreC: n  es un n\'umero entero, n >= 0
      x <-- Factorial(n)
// PostC: x = n!

 Factorial (n: integer ): integer;
 {
    If  ( n = 0 )  Then   Return  1;
       Else
            Return   n*Factorial(n-1);
 }
```

Se demostrará, utilizando inducción matemática, que el algoritmo es correcto.

Teorema 4.1 Para todo número entero $n, n \geq 0$, el algoritmo Factorial regresa $n!$.

Demostración. Inducción sobre n.

Base de la Inducción.

Para $n = 0$ el algoritmo regresa 1, lo cual es correcto, ya que $0! = 1$.

Hipótesis de Inducción.

FACTORIAL(j) regresa $j!$, $\forall\ j,\ 0 \leq j \leq (n-1)$.

Paso Inductivo.

Por demostrar que FACTORIAL(n) regresa $n!$

Como $n > 0$, la pregunta ¿ $n = 0$? resulta falsa y el algoritmo realiza la acción del **else**, es decir, regresa $n*$ FACTORIAL$(n-1)$. Por hipótesis de induccion, FACTORIAL$(n-1)$ regresa $(n-1)!$, por lo tanto, FACTORIAL(n) regresa $n * (n-1)! = n!$.

Nótese que la prueba se facilita porque la llamada recursiva se realiza sobre un ejemplar de menor tamaño que el original y así la hipótesis de inducción puede ser aplicada. Aunque el Teorema 4.1 no dice nada acerca del comportamiento de FACTORIAL(n) cuando $n < 0$, el algoritmo nunca falla para tales valores de n.

4.1.3. Búsqueda Binaria.

Problema: Determinar si un número x se encuentra en el arreglo ordenado $A[a..b]$, utilizando Búsqueda Binaria.

El arreglo debe ser no vacío, debe al menos tener un elemento. Esto es, si n es el número de elementos en el arreglo se tiene que $n \geq 1$. Además si a y b son los índices extremos del arreglo, tendremos que $n = b - a + 1 \geq 0 \implies a \leq b + 1$.

El algoritmo de búsqueda binaria primero compara a x con el elemento que se encuentra a la mitad del arreglo, $A[mid]$. Si resulta ser igual, ya terminó. Si $x < A[mid]$, entonces x debería estar en la primera mitad del arreglo, parte izquierda o superior del arreglo. Si $x > A[mid]$, entonces x debería estar en la otra mitad del arreglo. El algoritmo puede expresarse de manera natural en forma recursiva, El Listado 14 muestra una versión recursiva de este algoritmo.

Se demostrará, utilizando inducción matemática, que el algoritmo es correcto.

Teorema 4.2 Para todo número entero $n, n \geq 0$, donde $n = b-a+1$ representa al número de elementos en el arreglo $A[a..b]$ el algoritmo BusquedaBinaria regresa correctamente el valor de la condición: $x \in A[a..b]$.

Demostración. Inducción sobre n.

Base de la Inducción: $n = 0$.

El arreglo es vacío, $a = b + 1$ y la expresión lógica ¿ $a > b$? resulta ser verdadera, por lo que el algoritmo regresa **False**. Lo cual es correcto, ya que x no está en el arreglo vacío.

Hipótesis de Inducción.

Supongase que para toda $j, 0 \leq j \leq (n-1)$, donde $j = b' - a' + 1$, el

Listado 14 Algoritmo de Búsqueda Binaria.

```
BusquedaB{
// PreC: a<= b+1  &  A[a..b]  es un arreglo ordenado.
       Encontro <-- BusquedaBinaria(A, a, b, x) ;
// PostC:} Encontro = x en  A[a..b]  &  A[a..b] sin cambios. }

BusquedaBinaria (A: AType; a,b: integer; x: KeyType ): boolean;
 // var  mid: integer;
 {
   If  ( a>b )   Then  Return  False;
     Else
        mid <-- (a+b) div 2;
        If  ( x = A[mid] )   Then
              Return   True;
        ElseIf  ( x < A[mid] )  Then
              Return     BusquedaBinaria(A, a, mid-1, x)
          Else  Return   BusquedaBinaria(A, mid+1, b, x)
 } // BusquedaBinaria
```

algoritmo BusquedaBinaria con los parámetros A, a', b', x regresa correctamente el valor de la cuestión $x \in A[a'..b']$.

Paso Inductivo.

Del cálculo de $m \longleftarrow (a+b)$ div 2 se tiene que $a \leq m \leq b$.

Si $x = A[m] \implies x \in A[a..b]$ y el algoritmo regresa TRUE.

Si $x < A[m]$, como A está ordenado $\implies x \in A[a..b] \iff x \in A[a..m-1]$.
Por hipótesis de inducción, esto es contestado correctamente por la llamada:

$$\text{BusquedaBinaria}(A, a, (m-1), x),$$

ya que $(m = 1) < n - 1$ y de hecho $0 \leq (m-1) - a + 1 \leq n - 1$.
Si $x > A[m]$, se tiene un caso similar.

4.2. Algoritmos Iterativos.

En esta sección explicaremos la técnica del **Invariante del Ciclo** *(Loop Invariant)* para probar que un algoritmo iterativo, que posee ciclos, es correcto. Describiremos el proceso a partir de un ejemplo y después lo ejemplificamos solucionando otro ejercicio.

4.2.1. Suma de Elementos en un Arreglo

Problema: Calcular la suma de todos los números en el arreglo A con índice inferior a e índice superior b.

El Listado 15 muestra un algoritmo iterativo para resolver este problema. Nótese que por definición $A[a..(a-1)]$ representa un conjunto vacío cuya suma es cero, el algoritmo SumaA calcula esta suma nula correctamente.

Listado 15 Algoritmo que calcula la suma de los elementos de un arreglo.

```
SumaA(A:Array; a, b : integer)
//  PreC:    a <= b+1
{
     i <-- a;
     suma  <-- 0;
     While  ( i != b+1 )   Do  {
          suma <-- suma + A[i];
             i <-- i + 1;
     }
}
// PostC:  suma =  A[a] + A[a+1] + ...+ A[b]
```

El **Invariante de un Ciclo** es una condición que se supone verdadera del inicio al final de cada iteración del ciclo. Por inicio de la iteración se entiende el momento justo antes de evaluar la condición de entrada y que el ciclo sea ejecutado.

Pueden existir varios Invariantes en un ciclo, pero nos interesa aquel que mantiene una relación entre las variables que modifican su valor de acuerdo a la ejecución del ciclo. En el ejemplo, la relación $a \leq b+1$ no se altera en el ciclo, en cambio la asignación $i \leftarrow i+1$ si se altera.

El Invariante del ciclo para el algoritmo SumaA es $suma = \sum_{j=a}^{i-1} A[j]$ ya que expresa la relación entre las variables $suma$ e i. Es fácil ver que esta condición se mantiene desde el principio hasta el final de cada iteración.

Teorema 4.3 Invariante del Ciclo para el Algoritmo SumaA.
Al inicio de la k-ésima iteración del algoritmo SumaA, se mantiene la condición:

$$suma = \sum_{j=a}^{i-1} A[j].$$

Demostración. Inducción sobre k, número de iteraciones.

Base de la Inducción: $k = 1$. Al inicio de la primera iteración, la asignación de valores para $suma$ e i, claramente asegura que:

$$i = a \quad \text{y} \quad suma = 0 \implies 0 = suma = \sum_{j=a}^{a-1} A[j].$$

Por lo tanto, la condición se cumple.

Hipótesis de Inducción

Al inicio de la k-ésima iteración del algoritmo SumaA, se tiene que:

$$suma = \sum_{j=a}^{i-1} A[j].$$

Paso Inductivo.

Por demostrar que la condición se mantiene después de una iteración más,

además suponemos que el ciclo aún no se termina, esto es que $i \neq b+1$. Sea $suma'$ e i' los valores de las variables $suma$ e i al inicio de la iteración $(k+1)$. Por demostrar que:

$$suma' = \sum_{j=a}^{i'-1} A[j].$$

Entonces,

$$suma' = suma + A[i] = \sum_{j=a}^{i-1} A[j] + A[i]$$

$$= \sum_{j=a}^{i} A[j] \quad = \quad \sum_{j=a}^{i'-1} A[j].$$

Por lo tanto la condición se mantiene al inicio de la iteración $(k+1)$.

Establecer el Invariante del Ciclo es la parte más díficil de la prueba, pero hay dos pasos que faltan por hacer para completar la demostración de que un algoritmo es correcto.

1. Mostrar que la Postcondición se satisface al final.
   Considerese la última iteración del ciclo en el algoritmo SumaA. Al final de la iteración, el Invariante se mantiene, como se ha mostrado. Entonces la evaluación de la condición: $i \neq b+1$ se realiza y resulta tener valor falso, por lo que termina el ciclo. Claramente, en este momento la condición:

   $$suma = \sum_{j=a}^{i-1} A[j] \quad \& \quad i = b+1$$

   se mantiene, pero $suma = \sum_{j=a}^{b} A[j]$ resulta ser la Postcondición.

   De esta forma se demuestra que la Postcondición se cumple cuando el algoritmo termina. En general, justo después de completar la ejecución del ciclo **While C** con el Invariante **I** del ciclo, la condición **I** y la negación de la condición **C** se satisface, esto es: **I** & **Not C** tiene valor Verdadero, lo cual es necesario para probar que la Poscondición se satisface.

2. Mostrar que no hay riesgo de un ciclo infinito.
   El método de la prueba consiste en identificar alguna variable entera que esté estrictamente incrementándose (decrementándose) de una iteración a la siguiente y mostrar que cuando es lo suficientemente grande (pequeña) el ciclo *debe* terminar.

   En el Algoritmo SumaA, i es estrictamente incrementada y cuando alcanza el valor $(b+1)$ el ciclo termina. Nótese que este argumento depende de que i no sea mayor que $(b+1)$ inicialmente. Es decir, la condición $i \leq (b+1)$ debe satisfacerse desde el inicio del algoritmo para garantizar la terminación del mismo.

En resumen, los pasos requeridos para probar que un algoritmo iterativo $\mathcal{A}$ es correcto, utilizando la Técnica del Invariante del ciclo, son:

1. Obtener una condicion **I**, candidata a ser el Invariante del Ciclo.
2. Probar, usando Inducción, que **I** es una Invariante para el ciclo.
3. Probar que: **I** & **Not C** $\Longrightarrow$ PostC,
   donde **C** es la condición de entrada al ciclo.
4. Probar que el ciclo es finito, esto es garantizar la finalización del mismo.

El Invariante del Ciclo incluye a todas las variables cuyos valores cambian en el ciclo y que a la vez expresan una relación entre ellas, la cual no cambia durante la ejecución del ciclo. El Invariante del Ciclo debe poseer informacion completa acerca del algoritmo para que éste tenga éxito, esto es, que alcance sus objetivos.

La forma general del invariante del ciclo puede ser obtenida de la Postcondición, ya que: **I** & **Not C** $\Longrightarrow$ PostC, donde **C** y PostC son conocidas.

Nótese que debemos tener bien definida la PostCondición. Algunas veces hay que generalizar la PostCondición para generar el invariante **I**. En particular, en el ejemplo del algoritmo SumaA reemplazamos b por $(i-1)$.

4.2.2. Convertir un número decimal a binario.

Listado 16 Algoritmo Convierte de Decimal a Binario.

```
ConvierteABinario ( n, b){
  // PreC: n  numero positivo
  int t = n;
  int k = 0;
  While ( t > 0 ) Do {
      k   <-- k + 1;
     b[k] <-- t mod 2;
      t   <-- t div 2;
  }
  // PostC: b arrego de bits
        - representacion binaria de n
```

Problema: Dado un número decimal convertirlo a su expresión binaria.

El Listado 16 muestra un algoritmo iterativo para este problema. El algoritmo recibe dos parámetros: n un número decimal positivo y b un arreglo de bits, donde se guardará la representación binaria de n. El algoritmo consiste de un ciclo con tres instrucciones: la primera sólo incrementa a k el contador de bits; la segunda calcula t mód 2, el residuo de la división de $t/2$ - regresa 1 si t es impar y 0 si t es par; y la última instrucción divide a t entre 2, usando la división entera, es decir ignorando las fracciones.

Teorema 4.4 El Algoritmo ConvierteABinario calcula correctamente la representación binaria del número dado n y la almacena en el arreglo b.

Para demostrar que el Algoritmo Convierte A Binario es correcto, requerimos proponer un Invariante para el ciclo y demostrar, usando inducción matemática, que tal invariante es la correcta para justificar el algoritmo.

En el siguiente resultado se da el invariante, pero antes observemos que: La expresión $t \cdot 2^k + m$ deberá ser el corazón del invariante y, por su puesto, el corazón del algoritmo. En la $k-$ésima iteración del ciclo el arreglo binario b representa a los k bits menos significativos de n y el valor de t corresponde al resto de los bits.

Teorema 4.5 Invariante del Ciclo para el Algoritmo Convierte a Binario.
Al inicio de la k-ésima iteración del algoritmo ConvierteABinario, se tiene que:
Si m es el entero representado por el arreglo binario $b[1..k]$ entonces

$$n = t \cdot 2^k + m.$$

Demostración. Inducción sobre k, número de iteraciones.

Caso Base: $k = 0$.
Si $k = 0$ entonces, por definición, $m = 0$ y $n = t$.
Hipótesis de Inducción: Si m es el entero representado por el arreglo binario $b[1..k]$ entonces $n = t \cdot 2^k + m$.

Paso Inductivo. Por demostrar que el invariante se satisface una iteración más.

Supongamos que, al inicio de la k-ésima iteración, se tiene que $n = t \cdot 2^k + m$ y que, además, $t > 0$, para poder realizar una iteración más.

Primero supongamos que, al inicio de la k-ésima iteración, t es par. En este caso, $t \mod 2$ es 0. Así que no hay *contribución*, esto es, m no cambia. Después t es dividido por 2 y k es incrementado. Por lo tanto, la Hipótesis de Inducción aún es verdadera.

Ahora supongamos que, al inicio de la k-ésima iteración, t es impar. En este caso, $t \mod 2 = 1$ y en la posición $k + 1$ del arreglo b se pone un 1, esto implica se se hace una *contribución* de 2^k a m. Después t es dividido por 2, es decir su nuevo valor es $(t-1)/2$, pues es impar, y k es incrementado. Así que, al final de la k-ésima iteración, se tiene la expresión:

$$(t-1)/2 \cdot 2^{k+1} + m + 2^k = (t-1) \cdot 2^k + m + 2^k = t \cdot 2^k + m = n$$

que es justamente lo que queríamos probar.

Aun nos falta demostrar un par de condiciones más para completar la prueba.

El algoritmo es finito. El ciclo termina cuando $t = 0$ y esto si llega a suceder ya que en cada iteración t es dividido en dos y, además, estamos utilizando la división entera.

El Invariante y la negación de la condición del ciclo implican la PostCondición.
Si $t = 0$ tenemos que $n = 0 \cdot 2^k + m = m$ por lo tanto, $n = m$.

4.3. Ejercicios

Ejercicio 4.1 Justifique con detalle que la versión recursiva del algoritmo de búsqueda binaria dada en el Listado 14.

Ejercicio 4.2 Justifique con detalle que la siguiente versión iterativa del algoritmo de búsqueda binaria es correcta.

```
procedure BinarySearch(var A:Atype; a,b:integer;
                       x:keytype; var pos:integer):boolean
    var i,j,mid:integer;
        found: boolean;
    begin
        (*Pre a<=b+1 and A[a]<=...<=A[b] *)
        i:=a; j:=b; found:=false;
        while (i!=j+1) and not found do
            mid:=(i+j) div 2;
            if x = A[mid] then found:=true;
            elsif x < A[mid] then j:=mid-1;
            else                  i:=mid+1;
        end;
    end;
    if found then pos:=mid; else pos:=j; end;
    (* Post: (found     => a<=pos<=b and A[pos] = x) and
             (nod found => a-1<=pos<=b and
             (for all k a<=k<=pos, A[k]<x) and
             (for all f pos+1<=k<=b, x<A[k])) *)
    return found;
end BinarySearch
```

Ejercicio 4.3 Pruebe que el proceso `SelectionSort(A,a,b)`; satisface la siguiente especificación:

*(*Pre:* $a < b+1$ **)*
`SelectionSort(A,a,b)`;
*(*Post:* $A[a] \leq A[a+1] \leq \ldots \leq A[b]$ **)*

```
Procedure SelectionSort(var A:Atype; a,b:integer):
    var i:integer;
    begin
        if a = b+1 then Do_Nothing
        Else
            i <- MinIndex(A,a,b);
            if i != a then Swap(A[i],A[a]);
            SelectionSort(A,a+1,b)
    end; {SelectionSort}
```

Puede asumir que `MinIndex(A,i,j)` regresa el índice de el elemento mínimo de un arreglo no vacío `A[i..j]` y `Swap(A[i],A[j])` intercambia los dos elementos indicados.

Ejercicio 4.4 Pruebe que el siguiente algoritmo es correcto con respecto a la precondición y a la postcondición dada.

*(*Pre: $a \leq b$ y $x \in A[a..b]$ *)*

```
i <- a;
While x != A[i] do i <- i+1;
```

*(*Post: $a \leq i \leq b$ y $b \notin A[a..i-1]$ y $x = A[i]$ *)*

Ejercicio 4.5 El siguiente algoritmo para evaluar el polinomio

$$a_0 + a_1x + a_2x^2 + \ldots + a_{k-1}x^{k-1}$$

en el punto $x = x_0$, es llamado Algoritmo de Horner. Pruebe que es correcto.

```
Procedure Horner(a:Pol; x0:iteger):integer
    var i, total:integer
    Begin
        total <- 0;
        For i=k-1 to 0 by -1 do total <- a[i] + total*x0;
        Return total
    End {Horner}
```

Ejercicio 4.6 Diseñe un algoritmo para convertir un número binario a un numero decimal. La entrada es un arreglo de bits b de tamaño k, y la salida es un número n. Justifique el algoritmo usando un invariante del ciclo.

Ejercicio 4.7 Diseñe un algoritmo que convierta un número en base 6, a binario. La entrada es un arreglo de dígitos en base 6, y la salida es un arreglo de bits. Probar que el algoritmo es correcto usando invariantes del ciclo.

Ejercicio 4.8 Diseñe un algoritmo recursivo para calcular x^n en tiempo $O(\log n)$. Demuestre usando inducción matemática que su algoritmo es correcto.

Ejercicio 4.9 Diseñe un algoritmo iterativo para calcular x^n en tiempo $O(\log n)$. Demuestre usando invariantes del ciclo que su algoritmo es correcto.

Ejercicio 4.10 Diseñe un algoritmo iterativo para calcular la suma $2^0+2^1+2^2+\cdots+2^n = 1+2+4+\cdots+2^n$. Demuestre usando invariantes del ciclo que su algoritmo es correcto y calcule su desempeño computacional.

Ejercicio 4.11 Proporcione un algoritmo iterativo para calcular la suma de los primeros n números impares. Demuestre usando invariantes del ciclo que su algoritmo es correcto y calcule su desempeño computacional.

Ejercicio 4.12 Proporcione un algoritmo recurivo para calcular la suma de los primeros n números impares. Demuestre usando inducción que su algoritmo es correcto y calcule su desempeño computacional.