Análisis de Algoritmos
Tarea 03: Justificación de algoritmos

Profesora: María de Luz Gasca Soto
Ayudantes: Rodrigo Fernando Velázquez Cruz
Teresa Becerril Torres
Nombre: Alvaro Ramirez Lopez
N°. de Cuenta.: 316276355
Correo: alvaro@ciencias.unam.mx

1. Problema 1

Considerar los siguientes problemas:

- Problema α: Calcular el producto de los elementos en el arreglo de números enteros $A[a..b]$.
- Problema β: Calcular la suma de los primeros n múltiplos de 3.
- Problema γ: Calcular la suma de los números impares en el arreglo de números enteros $A[a..b]$.

Elegir **dos** de los problemas anteriores, α o β o γ y ...

a) Proporcionar un algoritmo recursivo (código) que solucione el problema, indicando PreCondiciones y PostCondiciones

b) Demostrar que el algoritmo propuesto es correcto usando inducción matemática.

c) Calcular el tiempo de ejecución del algoritmo dado.

Solución:

Problema α

a) Proporcionar un algoritmo recursivo (código) que solucione el problema, indicando PreCondiciones y PostCondiciones

Precondicion: El arreglo no debe ser null y debe tener al menos un elemento.

```haskell
productoElementos :: [Int] -> Int
productoElementos [x] = x
productoElementos (x:xs) = x * productoElementos xs
```

Postcondicion: El resultado es el producto de todos los elementos del arreglo, el arreglo sigue siendo no null.

a) Demostrar que el algoritmo propuesto es correcto usando inducción matemática.

Demostración:

Por Demostrar: La función productoElementos devuelve el producto de todos los elementos de la lista xs para cualquier lista no vacía xs.

Caso base: Cuando xs es una lista con un solo elemento.

En este caso, al ser un único elemento, por vacuidad se cumple que el producto de todos los elementos de la lista es el elemento mismo, entonces se regresa el elemento.

Hipótesis de inducción: Supongamos que la función productoElementos devuelve el producto de todos los elementos de la lista xs de longitud k.

Paso inductivo: Ahora demostraremos que la función productoElementos devuelve el producto de todos los elementos de la lista ys de longitud k+1.

La función productoElementos ys podemos verla como una función que tiene un arreglo de tamaño $k+1$ y se representa de la siguiente forma $[y_1, y_2, y_3, ..., y_k, y_{k+1}]$, asi mismo, podemos expresarla de la siguiente forma $[y_1, y_2, y_3, ..., y_k] + +[y_{k+1}]$.

Una vez expresada de esta forma, podemos ver que la función productoElementos $[y_1, y_2, y_3, ..., y_k]$ es equivalente a la función productoElementos xs de longitud k, por lo que podemos aplicar la hipótesis de inducción y sabemos que la función productoElementos $[y_1, y_2, y_3, ..., y_k]$ devuelve el producto de todos los elementos de la lista ys de longitud k.

Nos queda el otro extremo donde esta el elemento y_{k+1}, cuando se le pasa como argumento a la funcion *productoElementos* $[y_{k+1}]$ podemos ver que se le esta pasando un unico elemento, entonces caen en nuestro caso base y esto nos devuelve el mismo elemento.

Entonces podemos multiplicar los resultados generados por las funciones *productoElementos* $[y_{k+1}]$ y *productoElementos* $[y_1, y_2, y_3, ..., y_k]$ y obtenemos el producto de todos los elementos de la lista ys de longitud k+1, los cuales se expresarian de la siguiente forma.

$$\text{productoElementos } ys = (y_1 * y_2 * ... * y_k) * y_{k+1}$$

Por lo tanto, hemos demostrado por inducción matemática que el código es correcto, ya que cumple con la hipótesis inductiva para todas las listas no vacías.

a) Calcular el tiempo de ejecución del algoritmo dado.

El tiempo de ejecución del código que calcula el producto de todos los elementos de una lista utilizando recursion depende de la longitud de la lista. En este caso, la función *productoElementos* se implementa de manera recursiva, y la cantidad de operaciones realizadas es directamente proporcional al tamaño de la lista. Cada llamada recursiva reduce la longitud de la lista en 1.

Por lo tanto, si tienes una lista de longitud n, la función realizará $n-1$ llamadas recursivas. Cada llamada implica una multiplicación y una operación de acceso a la lista, lo que significa que en total se realizarán aproximadamente $2 * (n-1)$ operaciones aritméticas y de acceso a la lista.

Entonces, la complejidad de tiempo para este código es $O(n)$, donde n es la longitud de la lista. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño de la lista.

Problema β

a) Proporcionar un algoritmo recursivo (código) que solucione el problema, indicando PreCondiciones y PostCondiciones

```haskell
sumaPrimerosN :: Int -> Int
sumaPrimerosN 0 = 0
sumaPrimerosN n
    | mod n 3 == 0 = n + sumaPrimerosN (n - 1)
    | otherwise = sumaPrimerosN (n - 1)
```

a) Demostrar que el algoritmo propuesto es correcto usando inducción matemática.

b) Calcular el tiempo de ejecución del algoritmo dado.

2. Problema 2

Considera los siguientes problemas:

a) Problema Evaluación de Polinomios.

El Algoritmo Horner evalúa, en el punto x = xo, el polinomio

$P_n(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$

b) Problema Búsqueda Binaria. Dado un arreglo de enteros $A[a..b]$, ordenado en forma ascendente, determinar si el elemento x ∈ A e indicar la posición donde se encuentra, si está.

Dados los códigos para los problemas de A y B, mostrados abajo, demostrar usando la Técnica del Invariante del ciclo, que los algoritmos (códigos) son correctos.

```
Procedure H (A:Array of integers ; x0: Integer)
  var i, total : Integer;
    total ← 0 ;
      for i = n - 1 to 0 by -1 do
        tal ← A[i] + total * x0;
      Return total
end Horner
```

```
1:  function BINARY-SEARCH(A, n, v)
2:      ▷ Initialize the search range
3:      l ← 1
4:      r ← n
5:
6:      while l ≤ r do
7:          mid ← floor ((l+r)/2)
8:          if A[mid] < v then
9:              l ← m + 1
10:         else if A[mid] > v then
11:             r ← m − 1
12:         else
13:             return m
14:     return null
```

```
BinarySearch(var A:Atype; a,b:integer;
```

```
                  x:keytype; var pos:integer):boolean
// PreCondicion: a<=b+1 and A[a]<=...<=A[b]
var i,j,mid:integer; found: boolean;
begin
  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;
      elseif (x < A[mid]) then j=mid-1;
        else                   i=mid+1;
    end_if;
  end_while;
  if found then pos=mid;
    else pos=j;
  end_if;
  return found;
end BinarySearch

// PostC: (found implica: a<=pos<=b and A[pos] = x) and
          (not found implica: 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]))
```

Solución:

Aquí va la solución.

3. Problema Opcional.

Para los problemas elegidos en el Ejercicio I

a) Proporcionar un algoritmo iterativo (código) que solucione el problema, indicando PreCondiciones y PostCondiciones

b) Demuestra que el algoritmo es correcto usando la técnica del Invariante del ciclo (loop invariant).

c) Determinar el desempeño computacional del algoritmo dado.

Solución:

Aquí va la solución.