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 ɣ y ...

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

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 -- Caso base, cuando el arreglo tiene un elemento
productoElementos (x:xs) = x * productoElementos xs -- Parte recursiva
```

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

b) 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 en la lista, 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 función *productoElementos* $[y_{k+1}]$ podemos ver que se le esta pasando un único 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 expresarían 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.

c) 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

Precondicion: n debe de ser mayor o igual a 1 y el resultado de la operación debe ser mayor o igual a 3.

```haskell
sumaPrimerosN :: Int -> Int
sumaPrimerosN 1 = 3 -- Caso base, cuando n es igual a 1
sumaPrimerosN n = n * 3 + sumaPrimerosN (n - 1) -- Parte recursiva
```

Postcondicion: El resultado es la suma de los primeros n múltiplos de 3 y el resultado sigue siendo mayor o igual a 3.

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

Demostración

Por Demostrar: La función *sumaPrimerosN* devuelve la suma de los primeros n múltiplos de 3 para cualquier número n mayor o igual a 1.

Caso Base: Cuando $n = 1$.

Para el caso base, consideramos $n = 1$. La función *sumaPrimerosN 1* debe devolver 3, lo cual es correcto según la definición de la función y porque 3 es el primer numero múltiplo de 3. Por lo tanto, el código cumple con la base de inducción.

Hipótesis de inducción:

Suponemos que la función *sumaPrimerosN* devuelve la suma de los primeros n múltiplos de 3 para cualquier número n mayor o igual a 1.

Paso inductivo: Ahora demostraremos para $n + 1$.

Ahora, queremos demostrar que la hipótesis inductiva también es válida para $n + 1$.

Cuando el código se le pasa como parámetros $n + 1$, en la primera iteración tenemos la siguiente operación: `(n+1)*3 + sumaPrimerosN n`

Por hipótesis de inducción sabemos que para el caso *sumaPrimerosN n* se cumple, ya que en esa segunda llamada recursiva n se esta multiplicando por 3 y esto hace que el resultado de esa operación siga cumpliendo que es un múltiplo de 3, asi seguimos hasta llegar al caso base donde también se cumple esta propiedad.

Entonces el resultado que nos queda es: `(n+1)*3 + n*3 + (n-1)*3+ ... + 3`

Como podemos observar, el código cumple con el paso inductivo y la hipótesis, por lo tanto la función *sumaPrimerosN* devuelve la suma de los primeros n múltiplos de 3 para cualquier número n mayor o igual a 1.

c) Calcular el tiempo de ejecución del algoritmo dado.

La función `sumaPrimerosN` se implementa de manera recursiva y realiza llamadas recursivas hasta que `n` llega a 1. En cada ciclo se realiza la multiplicación del parámetro n por 3 y se hace la llamada recursiva con un elemento menos como parámetro. Por lo tanto, el tiempo de ejecución sería proporcional a `n`. **El tiempo de ejecución sería O(n)**.

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
```

```
        total ← A[i] + total * x0;
      Return total
end Horner

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:

a) Problema Evaluación de Polinomios (Algoritmo Horner).

Para realizar una demostración por invariante de ciclo vamos a necesitar definir una invariante de ciclo y verificar como es su funcionamiento o alteración en 3 etapas del código: inicializacion, mantenimiento y terminación, a continuación explicaremos como vamos a tomar la invariante y su verificación.

Invariante de Ciclo: En cada iteración del ciclo, `total` contiene la suma de los términos evaluados hasta ese punto, multiplicados por `x0` correspondiente según la posición en el arreglo `A`.

Ahora veremos las etapas en donde nuestra invariante sera verificada que tenga los valores esperados.

1. Inicialización: Antes de que comience el ciclo, `total` se inicializa en 0, lo cual es coherente con el invariante de ciclo ya que no hemos evaluado ningún término aún.

2. Mantenimiento: Supongamos que el invariante de ciclo es cierto al inicio de una iteración dada del ciclo ya que total la inicializamos en 0. En esa iteración, se evalúa el término `A[i]`, donde $i = n - 1$ y se suma a `total * x0`. El resultado se almacena nuevamente en `total`. El invariante de ciclo se mantiene porque `total` ahora contiene la suma de los términos evaluados hasta ese punto, multiplicados por `x0`. Además, `i` se decrement en 1, por lo que en la siguiente iteración se evaluará el siguiente término en el polinomio $n - 2$ y asi en otros ciclos. Por lo tanto, el invariante de ciclo se mantiene.

3. Terminación: Cuando el ciclo termina (es decir, cuando `i` llega a 0), el invariante de ciclo garantiza que `total` contiene la suma de todos los términos evaluados, multiplicados por `x0` según la posición en el arreglo `A`. Esto es exactamente lo que se espera del algoritmo de evaluación del polinomio en forma de Horner.

Por lo tanto, hemos demostrado que el algoritmo es correcto utilizando la Técnica del Invariante del Ciclo.

3. Problema Opcional.

Para los problemas elegidos en el Ejercicio I

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

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:

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

Problema α

Precondicion: La lista no debe ser null y debe tener al menos un elemento.

```
def productoElementos(lista):
  if len(lista) == 1:  # Lista con un solo elemento
      acumulador = lista[0]
      return acumulador
  else:
      acumulador = lista[0]
      for i in range(1, len(lista)):
          acumulador *= lista[i]
      return acumulador
```

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

problema β

Precondicion: n debe de ser mayor o igual a 1 y el resultado de la operación debe ser mayor o igual a 3.

```
def sumaPrimerosN(n):
  acumulador = 0
  for i in range(1, n+1):
      acumulador += i*3
  return acumulador
```

Postcondicion: El resultado es la suma de los primeros n múltiplos de 3 y el resultado sigue siendo mayor o igual a 3.

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

Problema α

Para realizar una demostración por invariante de ciclo vamos a necesitar definir una invariante de ciclo y verificar como es su funcionamiento o alteración en 3 etapas del código: inicializacion, mantenimiento y terminación, a continuación explicaremos como vamos a tomar la invariante y su verificación.

Invariante de Ciclo: La invariante que vamos a elegir es que el acumulador contiene el producto de todos los elementos de la lista hasta el momento de la iteración.

Ahora veremos las etapas en donde nuestra invariante sera verificada que tenga los valores esperados.

1. Inicialización: Antes de que comience el ciclo, el acumulador se inicializa en 0, lo cual es coherente con el invariante de ciclo ya que no hemos evaluado ningún término aún.

2. Mantenimiento: En nuestro algoritmo tenemos 2 casos que contemplar, el caso cuando `A` solo tiene un elemento y el caso cuando `A` tiene n elementos.

Caso cuando A solo tiene un elemento: En este caso, el acumulador se inicializa con el primer elemento de la lista, por lo que el invariante de ciclo se mantiene y como no hay mas elementos en la lista por los cuales multiplicar, el ciclo termina y se regresa el acumulador.

Caso cuando A tiene n elementos: En este caso, el acumulador se inicializa con el primer elemento de la lista, por lo que el invariante de ciclo se mantiene. En la siguiente iteración, el acumulador se multiplica por el siguiente elemento de la lista, por lo que el invariante de ciclo se mantiene. Esto se repite hasta que se termina de recorrer la lista, por lo que el invariante de ciclo se mantiene y como ya no hay mas elementos que recorrer, el ciclo termina y se regresa el acumulador.

3. Terminación: Aquí podemos verificar que el acumulador contiene el producto de todos los elementos de la lista, y este es distinto de 0 como lo era en la etapa de inicializacion, por lo que el invariante de ciclo se mantiene.

Por lo tanto, hemos demostrado que el algoritmo es correcto utilizando la Técnica del Invariante del Ciclo.

problema β

Para realizar una demostración por invariante de ciclo vamos a necesitar definir una invariante de ciclo y verificar como es su funcionamiento o alteración en 3 etapas del código: inicializacion, mantenimiento y terminación, a continuación explicaremos como vamos a tomar la invariante y su verificación.

Invariante de Ciclo: La invariante que vamos a elegir es que el acumulador contiene la suma de los primeros n múltiplos de 3 hasta el momento de la iteración.

Ahora veremos las etapas en donde nuestra invariante sera verificada que tenga los valores esperados.

1. Inicialización: La variable acumulador se inicializa en 0, lo cual es coherente con el invariante de ciclo ya que no hemos evaluado ningún término aún.

2. Mantenimiento: Durante el ciclo `for` que va de 1 hasta $n + 1$, el acumulador ira sumando las multiplicaciones de la variable `i` por 3, esto es valido porque cualquier numero multiplicado por 3 es multiplo del 3, por lo que el invariante de ciclo se mantiene en cuanto a su funcion propuesta, estos ciclos se repiten $n + 1$ veces.

3. Terminación: Verificamos el invariante de ciclo, el acumulador ahora contiene la suma de los primeros n múltiplos de 3, y este debe ser mayor o igual a 3, vemos que el acumulador igualmente es distinto de 0 a como lo era en la etapa de inicializacion, por lo que el invariante de ciclo se mantiene.

Por lo tanto, hemos demostrado que el algoritmo es correcto utilizando la Técnica del Invariante del Ciclo.

c) Determinar el desempeño computacional del algoritmo dado.

Problema α

El tiempo de ejecución de este algoritmo está dominado por el bucle `for` que recorre la lista. El bucle realiza una multiplicación por cada elemento de la lista, lo que implica que el tiempo de ejecución es lineal con respecto al tamaño de la lista `n`. Entonces la complejidad en tiempo seria de $O(n)$.

problema β

El tiempo de ejecución de este algoritmo está dominado por el bucle `for` que va desde 1 hasta $n + 1$. El bucle realiza una multiplicación y luego una suma por cada iteración, lo que implica que el tiempo de ejecución es lineal con respecto al tamaño de la lista `n`. Entonces la complejidad en tiempo seria de $O(n)$.