Universidad Nacional
Autónoma de México

Análisis de Algoritmos

Tarea 1

Almeida Rodríguez Jerónimo
jalrod@ciencias.unam.mx

Arteaga Vázquez Alan Ernesto
alanarteagav@ciencias.unam.mx

Eugenio Aceves Narciso Isaac
isaacn97@ciencias.unam.mx

1) 1) Propuesta de algoritmo:

```
1: procedure POSITIVOS(A[ ])          ▷ Recibe un arreglo con n ≥ 0 enteros
2:     is_positive ← True
3:     array_index ← 0
4:     if len(A[]) == 0 then
5:         return True                ▷ Por vacuidad devuelve True
6:     end if
7:     while array_index < len(A[ ]) do
8:         is_positive &= A[array_index] ≥ 0
9:         array_index += 1
10:    end while
11:    return is_positive
12: end procedure
```

2) **Argumento de correctud:**
Mostramos por invariante que el ciclo interno al algoritmo POSITIVOS es correcto:

Invariante: Después de recorrer un índice en el arreglo, la variable *is_positive* es TRUE si no se ha encontrado un número negativo, de lo contrario, se almacena la variable FALSE.

- *Inicialización:* Antes del ciclo, se inicializa la variable *is_positive* a la constante TRUE pues al no haber recorrido el arreglo (línea 2), por vacuidad no se puede hallar ningún número negativo, por lo cual, se cumple la propiedad.
- *Mantenimiento:* Suponemos que la propiedad se cumple en una iteración previa del ciclo, queremos mostrar que se cumple después de una nueva iteración.
  Como la propiedad se cumple, en la variable *is_positive* se encuentra la constante TRUE si no se ha hallado ningún número negativo, FALSE en otro caso. Por la línea 8, se consulta el valor del número en el índice actual y se realiza una comparación de 'mayor o igual' con el número cero. Si el número es positivo, la comparación resulta en un booleano TRUE, de lo contrario, resulta en un booleano FALSE. Consideremos ahora el valor booleano de la comparación, de acuerdo a la línea 8, se le aplica la operación booleana 'AND' al resultado de la comparación junto con el valor booleano almacenado en *is_positive*. Se tienen ahora los siguientes casos:

- Si *is_positive* era TRUE, y el valor de la comparación es TRUE, la operación & (AND) regresa TRUE lo cual se asigna a la variable *is_positive*, manteniendo la propiedad de que *is_positive* es TRUE al no haberse encontrado un número negativo.
- Si *is_positive* era TRUE, y el valor de la comparación es FALSE, la operación & regresa FALSE lo cual se asigna a la variable *is_positive*, así, *is_positive* cumple el invariante, al ser FALSE cuando se acaba de hallar un número negativo.
- Si *is_positive* era FALSE, y el valor de la comparación es FALSE, la operación & regresa FALSE lo cual se asigna a la variable *is_positive*, así, *is_positive* cumple el invariante, al ser FALSE cuando se acaba de hallar un número negativo.
- Si *is_positive* era FALSE, y el valor de la comparación es TRUE, la operación & regresa FALSE lo cual se asigna a la variable *is_positive*, así, *is_positive* cumple el invariante, al continuar siendo FALSE por haberse hallado previamente un número negativo.

Al fin del ciclo, se incrementa *array_index* en una unidad, lo cual permite continuar con la iteración.

- *Mantenimiento:* La cláusula del ciclo while establece que este se rompe cuando la variable *array_index* es igual a la longitud del arreglo, por lo cual, se ha recorrido exitosamente la totalidad del mismo. Al haber terminado de recorrer el arreglo, la variable *is_positive* mantiene el valor requerido al no modificarse una vez recorrida la totalidad del arreglo.
  Mostrando que el ciclo cumple el invariante, sólo restan estas dos observaciones para mostrar que el algoritmo es correcto:
  - Si el arreglo es vacío, por vacuidad no existe un elemento negativo, esto equivale a que todos son positivos. Así, el algoritmo es correcto al asignar True a *is_positive* cuando el arreglo es vacío.
  - Al regresar la variable *is_positive*, la cual cumple la invariante del ciclo while y el caso cuando el arreglo es vacío, concluimos que el algoritmo es correcto.

3)

2) 1) 1: **procedure** SEGUNDO(A)

```
2:     min ← ∞
3:     almostMin ← ∞
4:     thing ← 0
5:     while thing < len(A) do
6:         if A[thing] ≤ min then
7:             almostMin ← min, min ← A[thing]
8:         end if
9:         thing ← 1 + thing
10:    end while
11:    return almostMin
12: end procedure
```

2)

3) **Argumento de complejidad en tiempo:**
Las líneas 2 - 4 del algoritmo realizan asignaciones de variables, lo cual se realiza en tiempo $O(1)$. Luego, dentro del ciclo *while* se realiza en la línea 6 una acceso a memoria del arreglo y una comparación de números, lo cual conlleva en ambos casos tiempo constante. Dentro del condicional, se llevan a cabo dos asignaciones de variables, las cuales toman tiempo $O(1)$, luego, en la línea 9, se incrementa un contador, lo cual se lleva en tiempo a lo más constante. Así, dentro del ciclo se llevan a cabo operaciones de tiempo constante.
Notamos ahora que el ciclo *while* tiene un número de iteraciones acotado por el tamaño del arreglo que el algoritmo que recibe como entrada, a saber, por el número de elementos (la longitud) del arreglo. Sea n el número de elementos del arreglo de entrada, entonces el ciclo *while* lleva a cabo a lo más n iteraciones, por lo tanto, como en cada iteración se llevan a cabo operaciones de a lo más tiempo constante, es correcto decir que el ciclo corre en tiempo $O(n)$. Finalmente, la instrucción de retorno en la línea 11 se lleva a cabo en tiempo constante.
Así, la complejidad del algoritmo está dada por el máximo de las complejidades, la cual corresponde a la del ciclo *while*, así, concluimos que el algoritmo corre en tiempo $O(n)$.

3) 1) Propuesta de algoritmo:

2)

3)

Figura 1: Rutina Princupal

```
   input  : Un arreglo A de enteros, con |A| ≥ 2
   output: Un entero con el mínimo producto del arreglo A
 1 (m1, m2, M1, M2) ← minMax(A);
 2 if m1 < 0 then
 3 |   if M1 < 0 then
 4 |   |   return M1 * M2
 5 |   else
 6 |   |   return m1 * M1
 7 |   end
 8 else
 9 |   return m1 * m2
10 end
```

Algorithm 1: minProd

Figura 2: Subrutina que obtiene los valores para minProd

```
   input  : Un arreglo A de enteros, con |A| ≥ 2
   output: Una tupla (m1, m2, M1, M2) donde m1 contiene el mínimo de A, m2 el segundo
           menor elemento de A, M1 contiene el máximo de A y M2 contiene el segundo
           mayor elemento de A.
 1 m1 ← A[0];
 2 m2 ← A[0];
 3 M1 ← A[0];
 4 M2 ← A[0];
 5 i ← 1;
 6 while i < length(A) do
 7 |   n ← A[i];
 8 |   if n < m1 then
 9 |   |   m2 ← m1;
10 |   |   m1 ← n;
11 |   end
12 |   if n > M1 then
13 |   |   M2 ← M1;
14 |   |   M1 ← n;
15 |   end
16 |   i ← i + 1;
17 end
18 return (m1, m2, M1, M2);
```

Algorithm 2: minMax