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:    index ← 0                    ▷ Índice para recorrer el arreglo
5:    while index < len(A) do
6:        if A[index] ≤ almostMin then
7:            almostMin ← A[index]
8:            if almostMin ≤ min then
9:                almostMin ← min, min ← A[index]
10:           end if
11:       end if
12:       index ← 1 + index
13:   end while
14:   return almostMin
15: end procedure
```

2) Procedemos ahora a demostrar que este algoritmo es correcto cuándo el arreglo contiene más de dos elementos.
Observamos que si no hay dos o más elementos en el arreglo no se puede asignar un segundo valor menor y entonces en la línea 14 se devuelve ∞.
Procederemos ahora a demostrar por inducción sobre el recorrido del arreglo suponiendo que la longitud es mayor o igual a dos.

C.B. Recorrer los primeros dos elementos del arreglo.
P.D. Que hasta este punto, el se tienen los dos elementos con menor valor del arreglo.
En la primer iteración tenemos que el valor de de $A[0] \leq almostMin$ por el valor asignado a $almostMin$ al incializarlo. Cómo la condición se cumple, entonces $almostMin$ obtiene el valor de $A[0]$ luego, la segunda condición se cumple, $almostMin = \infty$ y $min = almostMin$ En la segunda iteración, observamos que necesariamente se cumple la condición de línea 6 y el valor de $almostMin = A[1]$ luego, si $min <= almostMin$ estas dos variable intercambian valores, de otro modo, los valores se mantienen igual y en efecto, los valores han sido reducidos de manera correcta o mantenidos con el mismo valor de ser el caso.

H.I. Suponemos que el algoritmo es correcto por i pasos.

P.I. **P.D.** Que en la iteración $i + 1$ el algoritmo sigue devolviendo el valor correcto.
De aquí tenemos dos casos:
1) Si $A[i + 1] > almostMin$ entonces se sale del ciclo y se

devuelve el último valor almacenado en $almostMin$.
2) En el otro caso asignamos el valor de $A[i+1]$ a $almostMin$, de dónde tenemos otros dos subcasos:
2.a) Si el valor de $almostMin \geq min$, entonces se sale del ciclo y se continúa la ejecución.
2.b) De lo contrario, se intercambian los valores y ahora se sigue cumpliendo que $min \leq almostMin$. Se termina el ciclo y continúa la ejecución. De esto tenemos que en la $i - esima$ iteración los valores de $almostMin$ y de min son los correctos.
Es sencillo observar que este algoritmo va a terminar siempre que reciba un arreglo de tamaño finito debido a que la condición para que el ciclo `while` se siga ejecutando es que el índice (siempre creciente) sea menor a la longitud del arreglo.

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 de los ciclos *while* se realizan accesos 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 12, 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[1]:

2)

[1]Figuras 1 y 2

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

3) Del algortimo principal tenemos que en la primer línea se hace una llamada a la función auxiliar. Esta llamada se hace en tiempo constante. También podemos observar que el resto de las lineas se ejecutan en tiempo constante porque son evaluaciones de condicionales y operaciones aritméticas. Así, la complejidad en tiempo de esta rutina es de tiempo constante.

Ahora procedemos a demostrar la complejidad en tiempo del algortimo auxiliar.
Las asignaciones de las primeras 5 líneas se hacen en tiempo constante. Luego, tenemos el `while` que itera sobre todas las entradas del arreglo. Esto lo hace en un tiempo lineal sobre el el número de elementos del arreglo. Dentro de este ciclo observamos que tenemos un acceso al arreglo, dos comparaciones y cinco asignaciones, todas estas ejecutadas en tiempo constante. Así, el tiempo de ejecución de esta subrutina es de $5c + 8c * n$, con n la longitud del arreglo.
Por lo tanto, la complejidad del algoritmo $minProd$ es $O(n)$.