Laboratorio 1

José Alejandro Guzmán Zamora

2 de agosto de 2018

- ## Problema 1

  Escribir pseudocódigo para un algoritmo de búsqueda lineal. Describir el loop invariant del algoritmo para demostrar que es correcto.

Pseudocódigo Búsqueda Lineal

```
1       for i from 1 to len(A):
2           if A[i] == v:
3               return i
4       return null
```

- Loop Invariant: todos los valores a la izquierda de i no son iguales a v.

- ## Problema 2

  Demostrar el running time del siguiente algoritmo.

Multiplicación de Matrices

```
1       n           for i from 1 to n:
2       n*p             for j from 1 to p:
3       n*p                 let sum = 0
4       n*p*m               for k from 1 to m:
5       n*p*m                   set sum = sum + A[i][k] * B[k][j]
6       n*p                 set C[i][j] = sum
7       1           return C
```

La primer línea, correspondiente al inicio del loop externo se ejecuta n veces. La segunda línea se ejecuta n veces porque pertenece al loop externo, sin embargo, la instrucción también es un loop que se corre p veces. Todo lo que

está adentro del segundo loop se va a ejecutar p veces por n veces, no obstante, hay que tomar en cuenta que existe otro ciclo adentro. El tercer ciclo se ejecuta m veces, por lo que las líneas 4 y 5 se ejecutan n * p * m. En cuanto a la sexta línea, se sabe que pertenece al segundo loop y la última línea es una instrucción que se corre solamente una vez antes de terminar el algoritmo.

Al tener la información relacionada con la cantidad de veces que se ejecuta cada línea, ahora sumamos las cantidades.

$$(n)+(n * p)+(n * p)+(n * p * m)+(n * p * m)+(n * p) + 1$$

Hay que tomar en cuenta que se está realizando un análisis asintótico, por lo que en este caso la suma de la constante 1 no es relevante. Posteriormente se saca el factor común n.

$$n(1 + 3p + 2pm)$$

De nuevo se omite la constante 1 y se factoriza el paréntesis

$$n(p(3+2m)$$

Esto finalmente se puede expresar como n*p*m.
El running time del algoritmo es

$$O(n*p*m)$$

- ## Problema 3

Demostrar el worst-case running time del algoritmo Bubble Sort y explicar cómo se compara con el best-case y worst-case running time de insertion sort.

	Bubble Sort Algorithm	
1	1	S is an array of integer
2	n-1	for i in 1 : length((S)-1) do
3	(n-1)(n-1)	for j in (i + 1) : length(S) do
4	(n-1)(n-1)	if S[i] > S[j] then
5	(n-1)(n-1)	swap S[i] and S[j]

En el peor de los casos, el input para el algoritmo de ordenamiento Bubble Sort, es una secuencia opuestamente ordenada. En el caso de que se desee ordenar el arreglo [8,7,6,5,4,3,2,1] ascendentemente, es necesario hacer swap en cada comparación. Tomando en cuenta los valores relacionados con la cantidad de veces que se deban de ejecutar las líneas, se obtiene lo siguiente:

$$1 + (n - 1) + (n - 1)(n - 1) + (n - 1)(n - 1) + (n - 1)(n - 1)$$

Nuevamente, se omite la sumatoria de la constante, y la expresión superior se puede simplificar de la siguiente manera

$$\begin{gathered}(n - 1) + 3(n - 1)(n - 1) \\ \downarrow \\ (n - 1)(1 + 3(n - 1)) \\ \downarrow \\ (n - 1)(n - 1) \\ \downarrow \\ O(n^2)\end{gathered}$$

Al considerar la complejidad del algoritmo de ordenamiento Insertion sort en el peor de los casos, se observa que ambos algoritmos (insertion, bubble) realizan $O(n^2)$ instrucciones con respecto al tamaño del input. A grandes rasgos, a partir del análisis asintótico los algoritmos de Insertion Sort y de Bubble Sort son igualmente eficientes en el peor de los casos. En el caso del best-case running time, hay que ser cuidadoso con la estructura del algoritmo Bubble Sort. Para este pseudocódigo específicamente, estudiar el mejor de los casos (un arreglo inicialmente ordenado) resulta en un running time idéntico al del peor de los casos. Esto es porque sin importar el input, el algoritmo siempre ejecuta n^2 comparaciones en el ciclo anidado. En este caso el best-case running time del algoritmo Insertion Sort es menor, ejecutando O(n) instrucciones cuando el arreglo ya está ordenado.