Lab1

Fernando González

August 2, 2018

Problema 1

Algorithm 1 Busqueda Lineal

```
1: procedure LSearch
2:     Initialization:
3:     A=[x_1, x_2, x_3, x_4, x_5, x_6, x_7 ...]
4:     v = Input(num)
5:     for i=0; i in range A; i++ do
6:         if x == A[i] then
7:             Rerurn A[i]
8:             Goto Exit
9:         else if i == lenAandA[i]! = x[i] then
10:            Rerurn (The number is not part of the list.)
11:            goto top.
```

Mi *loop invariant* en mi algoritmo se mantiene, ya que al momento de hacer el *for loop* que va recorriendo mi *array*, me aseguro que cada vez que pasa por una posición revise si sea *v*, y todos los numeros ya recorridos no son *v*.

Problema 2

Algorithm 2 Multiplicación de Matrices

```
1: procedure xMatrix
2:     Initialization:
3:     Input: Matriz A(n x m) and Matriz B(m x p)
4:     Output: Matriz C(n x p)
5:
6:     for i from 1 to n do
7:         for j from 1 to p do
8:             Let sum = 0
9:             for k from 1 to m do
10:                Set sum ← sum + A[j][h] * B[k][j]
11:                Set C[i][j] ← sum
       return C
```

Ya que este algoritmo se basa en la multiplicación de matrices, es necesario analizar cada *loop* que se encuentra en el código. En el primer *loop*, este se tarda n veces la fila matriz A. El siguiente *loop* se empieza a dar la intresección entre las matrices de A Y B, por siguiente, se inicializa la suma que se hara por cada vez que se multiplican las matrices. En la linea 10, se empiezan a multiplicar las matrices *(Fila X Columa)*, se suma, el resultado se establece en *sum*. Por ultimo, por cada vez que se multiplican las matrices *(n*p*m)* se establecen en la nueva matriz C. Por lo tanto, este algoritmo tiene un running time de $O(n^3)$ dado la multiplicación de *(n*p*m)*.

~

6: n
7: n*p
8: n*p
9: n*p*m
10: n*p*m
11: n*p*m
12: 1

Problema 3

Algorithm 1 Bubble sort algorithm

```
S is an array of integer
for i in 1 : length(S) − 1) do
   for j in (i + 1) : length(S) do
      if S[i] > S[j] then
         swap S[i] and S[j]
      end if
   end for
end for
```

Worst Time: $O(n^2)$
Best Case: $O(n^2)$

Algorthm 2 *Insertion Sort*

~

Worst Time: $O(n^2)$
Best Case: $O(n)$

~

Worst case: *Bubble Sort* vs. *Insertion Sort*

Comparación:

En este caso, ambos tienen el mismo worst case, es decir, que ambos tardan la misma cantidad de tiempo en el peor de los casos. Pero, una pequeña diferencia que tiene el *Insertion Sort*, que lo hace un poco mas eficiente a nivel de comparaciónes, es que el *Insertion Sort* verifica de dos en dos, y les hace *switch*. Por otro lado el *Bubble Sort* lo hace de uno en uno, lo cual a nivel de comparaciónes es un poco menos eficiente.

Best case: *Bubble Sort* vs. *Insertion Sort*

Comparación:

En el best case de estos dos algoritmos si hay diferencia, ya que el best case de el *Insertion Sort* es $O(n)$ y el de *Bubble Sort* es $O(n^2)$. Esto quiere decir que el *Bubble Sort* es menos eficiente al momento de hacer el best case escenario, porque el *Insertion Sort* solo requiere de recorrer la lista una vez, *Bubble Sort* n^2.