Laboratorio Nro. 2
Complejidad de algoritmos

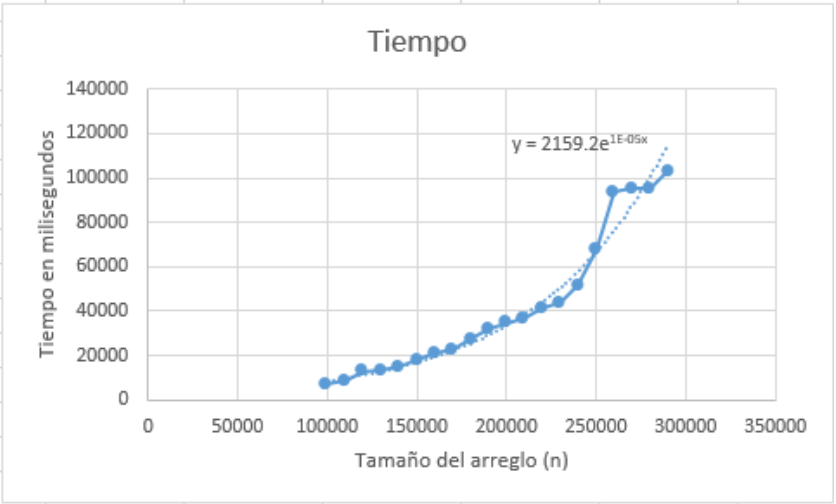
Simón Correa Henao
Universidad Eafit
Medellín, Colombia
scorreah@eafit.edu.co

David Gómez Correa
Universidad Eafit
Medellín, Colombia
dgomezc10@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

Punto 1.1 y 3.1 y 3.2 Tiempos de ejecución y grafica – Insert Sort

tamaño n	Milisegundos
100000	7009
110000	8768
120000	12972
130000	13251
140000	14651
150000	18204
160000	21131
170000	22425
180000	27241
190000	31923
200000	34695
210000	37080
220000	41390
230000	43567
240000	51953
250000	68158
260000	93832
270000	95071
280000	95535
290000	102941



Tiempos de ejecución y grafica – Merge

tamaño n	Milisegundos
10000000	1628
11000000	1792
12000000	1946
13000000	1716
14000000	1984
15000000	2293
16000000	2244
17000000	2352
18000000	2516
19000000	2586
20000000	2624
21000000	2849
22000000	3928
23000000	3539
24000000	3758
25000000	3807
26000000	3805
27000000	3946
28000000	4346
29000000	3842

3.3 ¿Es apropiado el Insert sort para un video juego con millones de elementos en una escena?

Claramente el algoritmo de insert sort no es apropiado para trabajar con millones de elementos, dado que su complejidad para casos muy demandantes como este en particular se comporta como una función n^2, es decir como una función cuadrática, por lo cual para una gran cantidad de datos, el tiempo que tomara ejecutar este algoritmo va a ser muy prolongado, y procesar datos de un juego en tiempo real requiere de tiempos de ejecución muy bajos para conseguir una ejecución optima.

3.4 ¿Por qué aparece un logaritmo en la complejidad asintótica, para el peor de los casos, de merge sort o insertion sort?

Antes que nada, es relevante notar que el algoritmo merge sort constantemente está dividiendo el tamaño del problema en 2. Esto último se ve reflejado en su ecuación de recurrencia, y dado que, al resolver una ecuación de recurrencia, sea mediante hipotesis inductiva o utilizando herramientas como Wolfram, que internamente divide por 2 el tamaño de su problema siempre da como solución una ecuación logaritmica, es por esto que se obtiene un logaritmo en el cálculo de complejidad del merge sort.

Cosa que no ocurre para el cálculo de complejidad del insertion sort.

3.5 Condingbat: Array2

CenteredAverage:

$T(n) = c_1 + c_2 + c_3(n-1) + (c_4 + c_5n)(n-1) + (c_6 + c_4 + c_5n)(n-1) + c_7 + c_8(n-1) + c_9 + c_10(n-1) + c_11(n-1) + c_12(n-1) + c_13(n-1) + c_14$

En notación O: $O(n^2)$

FizzArray:

$T(n) = c_1 + c_2 + c_3n + c_4n + c_5$

En notación O: O(n)

ShiftLeft: T(n) = c_1 + c_2 + c_3 + c_4 + c_5n + c_6n + c_7
En notación O: O(n)

HaveThree:
T(n) = c_1+c_2+c_3+c_4 + c_5 + c_6 + c_7n+c_8n+c_9n+ c_10
En notación O: O(n)

ZeroMax:
T(n)=c_1+c_2n+c_3n+c_4+(c_5 + c_6n)n+c_7n*n+c_8n*n+c_9n+c_10
En notación O: $O(n^2)$

Condingbat: Array 3

SeriesUp:
T(n)= c_1+c_2+c_3+c_4+(c_5)*n+(c_6+(c_7*n))*n+c_8

En notación O: $O(n^2)$

Fix34:
T(n)=c_1+(c_2*n)+(c_3+(c_4*n))*n+(c_5*n)*n+c_6

En notación O: $O(n^2)$

LinearIn:
T(n)=c_1+c_2+c_3+(c_4*n)+(c_5+(c_6*n))*n+(c_7*n)*n+c_8*n+c_9

En notación O: $O(n^2)$

CountClumps:
T(n)=c_1+(c_2*n)+c_3*n+c_4*n+c_5

En notación O: O(n)

MaxMirror:
T(n)=c_1+(c_2*n)+c_3+(c_4*n)+(c_5+(c_6*n))*n+(c_7*n)*n+(c_8*n)*n+ (c_9*n)*n+(c_10*n)*n+(c_11*n)*n+c_12*n+c_13

En notación O: $O(n^2)$

3.6
Array 2

CenteredAverage: La variable n en el cálculo, representa el tamaño del arreglo dado por el usuario, c_14 nos habla que existen 14 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos anidados, uno al interior del otro por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad cuadrática.

FizzArray: La variable n en el cálculo, representa el tamaño del arreglo que requiere el usuario, c_5 nos habla que existen 5 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay un ciclo en el código, por lo cual la complejidad se torna O(n) es decir una complejidad lineal.

ShiftLeft: La variable n en el cálculo, representa el tamaño del arreglo proporcionado por el usuario, c_7 nos habla que existen 7 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay un ciclo en el código, por lo cual la complejidad se torna O(n) es decir una complejidad lineal.

HaveTree: La variable n en el cálculo, representa el tamaño del arreglo proporcionado por el usuario, c_10 nos habla que existen 10 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay un ciclo en el código, por lo cual la complejidad se torna O(n) es decir una complejidad lineal.

ZeroMax: La variable n en el cálculo, representa el tamaño del arreglo dado por el usuario, c_10 nos habla que existen 10 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos anidados, uno al interior del otro por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad cuadrática.

Array 3

SeriesUp: La variable n en el cálculo, representa el número de veces que el usuario quiere que se repita el patrón de series up, a su vez esta variable internamente nos da el tamaño del arreglo a crear; c_8 nos habla que existen 8 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos, un al interior de otro por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad cuadratica.

Fix34: La variable n en el cálculo, representa el tamaño del arreglo dado por el usuario, c_6 nos habla que existen 6 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos, un al interior de otro por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad potencial.

LinearIn: La variable n en el cálculo, representa el tamaño del arreglo más grande dado por el usuario, c_9 nos habla que existen 9 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos, un al interior, para evaluar la existencia del arreglo más pequeño dentro del más grande, por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad cuadratica.

CountClumps: La variable n en el cálculo, representa el tamaño del arreglo dado por el usuario, c_5 nos habla que existen 5 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que solo hay un ciclo, por lo cual la complejidad de dicho algoritmo es de O(n).

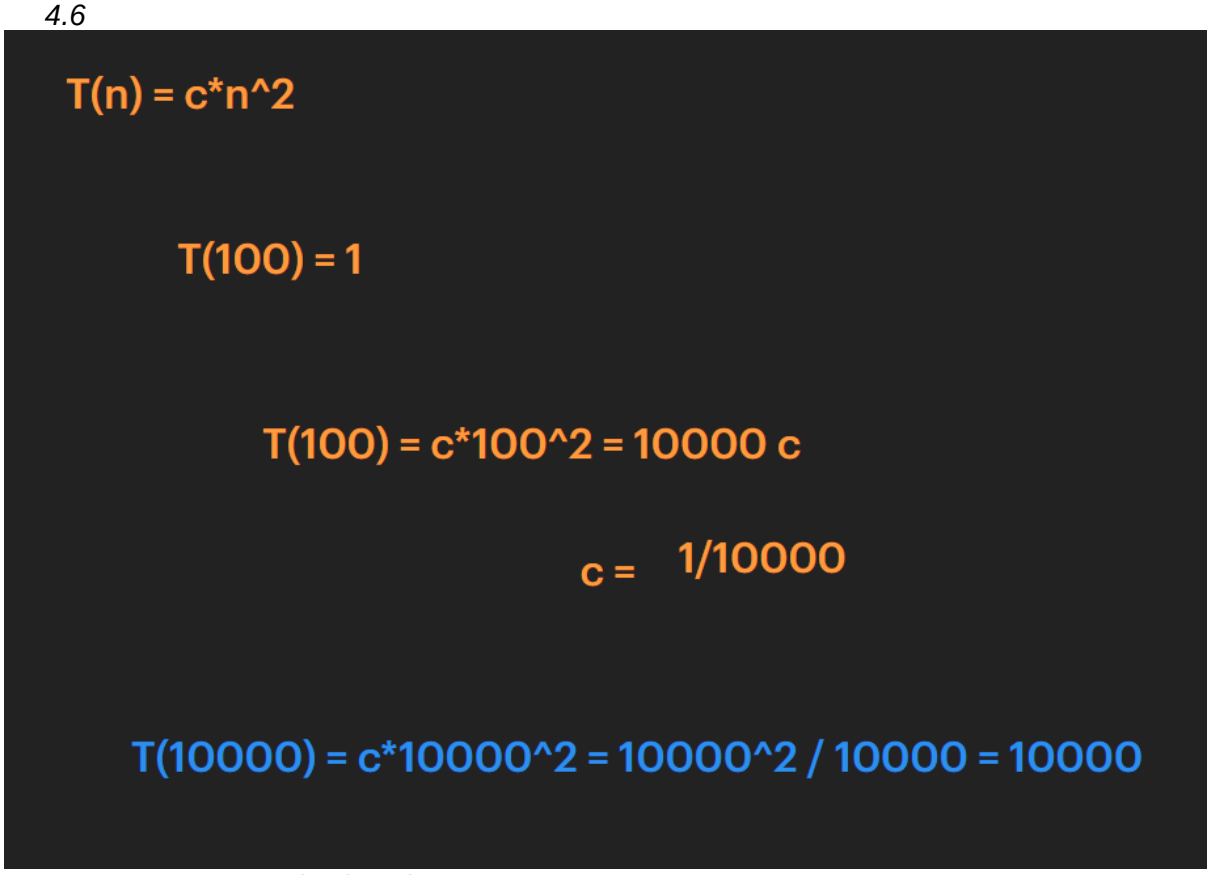
MaxMirror: La variable n en el cálculo, representa el tamaño del arreglo dado por el usuario, c_13 nos habla que existen 13 constantes dentro del algoritmo (operaciones de tiempo constante). Además, podemos observar que hay dos ciclos, un al interior de otro por lo cual la complejidad se torna $O(n^2)$ es decir una complejidad cuadratica, es cierto que existe un tercer ciclo dentro del algoritmo, pero este no se encuentra anidado dentro de otro, por lo cual, por reglas de O el tiempo que toma ejecutar este es menor al de los dos ciclos anidados.

4) Simulacro de Parcial

Respuestas:

4.1 C)
4.2 B)
4.3 B)
4.4 B)

4.5 D), A)
4.6



4.7 Respuestas: A), B) y C)
4.8 A)
4.9 A)
4.10 C)
4.11 C)
4.12 B)
4.13 C)
4.14 C)