Instituto Politécnico Nacional
Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales, Análisis y Diseño de Algoritmos
Sem: 2025-1, 3CM3, Práctica 1, Fecha 12/09/2024

Práctica 1: Determinación experimental de la complejidad temporal de un algoritmo

López Domínguez Daniel Efraín, Vite Valois Omar Abdiel.

dlopezd1701@alumno.ipn.mx [1], ovitev1900@alumno.ipn.mx [2]

Resumen: En esta práctica se realizó el análisis a posteriori de los algoritmos para encontrar los puntos silla de una matriz y encontrar puntos máximos locales en un arreglo de enteros. Para resolverlo usaremos gráficas y contaremos los pasos en el algoritmo que tengan un impacto en el tiempo de ejecución. Los resultados se observan interesantes analizando el mejor y el peor caso.

Palabras Clave: C, análisis a priori, complejidad algorítmica, función, iteración.

1 Introducción

Un algoritmo es el conjunto de instrucciones sistemáticas y previamente definidas que se utilizan para realizar una determinada tarea. Estas instrucciones están ordenadas y acotadas a manera de pasos a seguir para alcanzar un objetivo.

Todo algoritmo tiene una entrada, conocida como input y una salida, conocida como output, y entre medias, están las instrucciones o secuencia de pasos a seguir. Estos pasos deben estar ordenados y, sobre todo, deben ser una serie finita de operaciones que permitan conseguir una determinada solución. Sin embargo, en algunas ocasiones entre más grande sea la entrada de datos de un algoritmo, es importante analizar qué tan complejos pueden volverse estos mismos.

La complejidad de un algoritmo es una medida de cuán eficiente es el algoritmo para resolver el problema. En otras palabras, es una medida de cuánto tiempo y espacio (memoria) requiere el algoritmo para producir una solución.

En matemáticas, la complejidad se estudia a menudo en el contexto de los algoritmos. Esto se debe a que muchos problemas matemáticos se pueden resolver mediante algoritmos, y la eficiencia de estos algoritmos es un factor importante en su utilidad en la práctica.

Hay varias formas de medir la complejidad de un algoritmo. Uno de los más comunes es contar el número de operaciones básicas (como sumas o multiplicaciones) que realiza el algoritmo. Esto se conoce como la complejidad temporal del algoritmo. Otra forma de medir la complejidad es contar la cantidad de memoria (*en bytes o bits*) que requiere el algoritmo. Esto se conoce como la complejidad espacial del algoritmo.

El objetivo de esta práctica es realizar el análisis a posteriori de dos algoritmos, el primero es punto silla y el segundo es punto máximo, esto con el objetivo de encontrar el mejor y peor caso de cada uno para conocer su complejidad algorítmica y así mismo mediante graficas se observará el comportamiento que tienen los mismos.

2 Conceptos básicos

Θ (Teta) es un símbolo griego que se usa para describir el comportamiento asintótico de una función matemática, tanto en el mejor como en el peor caso. Esta notación proporciona una representación precisa de cómo crece una función en términos de una función comparativa, proporcionando una cota ajustada tanto superior como inferior. A diferencia de otras notaciones asintóticas como Ω (Omega) y O (O grande), que solo ofrecen una cota inferior y superior respectivamente, Θ indica que la función está acotada tanto superiormente como inferiormente por la misma función en un rango determinado. Esto la convierte en una herramienta útil para caracterizar el rendimiento de algoritmos y funciones en términos de su crecimiento asintótico exacto.

Definición formal:

$$\boldsymbol{\Theta\big(g(n)\big) = \{f(n) : \exists\, n, \quad c_1, \quad c_2, \quad n_0 \mid 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\, n \ge n_0\}}$$

O (Big O) es una notación que se utiliza para representar el conjunto de funciones que están acotadas por encima de una función f(n). Esta notación se emplea para describir el comportamiento asintótico en el peor caso de una función, proporcionando una cota superior para el crecimiento de la función. Además, la notación Big O permite expresar de manera clara y precisa los límites experimentales del rendimiento, ayudando a entender cómo se comporta el algoritmo o función en situaciones extremas o de alta carga.

Definición Formal:

$$\boldsymbol{O\big(g(n)\big) = \{f(n)\ \exists\ c > 0\ \wedge\ n_0\ constantes \mid 0 \le f(n) \le cg(n)\ \forall\, n \ge\ n_0\}}$$

Ω (Omega) es una notación que se emplea para representar una cota inferior de la función f(n). Se utiliza para describir el conjunto de funciones que están por debajo de la función experimental f(n), proporcionando una cota mínima para su crecimiento. Esta notación permite expresar los límites inferiores del rendimiento de un algoritmo o función, ayudando a comprender cómo se comporta en el mejor de los casos o bajo condiciones óptimas.

Definición Formal:

$$\boldsymbol{\Omega\big(g(n)\big) = \{f(n) : \exists\ c > 0\ \wedge\ n_0\ constantes\ |c\big(g(n)\big) \le f(n)\ \forall\, n \ge\ n_0\}}$$

El primero de los algoritmos, punto silla, se divide en dos partes. En una matriz $\boldsymbol{M_{mn}}$ se busca el primer punto donde $\boldsymbol{M_{ij}}$ menor o igual a los elementos de su respectiva fila y mayor o igual en todos los elementos de su respectiva columna. Después, la segunda parte del primer algoritmo

buscará todos los puntos silla que existan en la matriz. Para fines prácticos, en esta práctica usaremos matrices cuadradas, aunque el algoritmo puede usarse en Matrices mxn siendo n y m distintas y mayor que 0.

El segundo algoritmo también se divide en 2, búsqueda de puntos máximos locales. En un arreglo de tamaño n, $\boldsymbol{Arreglo[0,..,n-1]}$ buscaremos aquel primer punto que cumpla con $\boldsymbol{Arreglo[i-1] < Arreglo[i[< Arreglo[i+1]}$, la segunda parte abarcara todos los puntos máximos locales que existan en el arreglo.

Se planteó un algoritmo base por cada problema, una versión diferente a conveniencia según el problema lo indique, a continuación, los pseudocódigos diseñados para resolver los problemas planteados, añadimos algunas ideas que se desarrollaron en el apartado de experimentación y resultados.

```
function encontrarMinimoFila(Matriz, fila, tam)
    aux = Matriz[fila][0]

    for i from 1 to tam - 1 do
        if aux > Matriz[fila][i] then
            aux = Matriz[fila][i]
        end if
    end for

    return aux
end function
```

Ilustración 1. Encontrar el mínimo de una fila

El problema esta dividido en varias funciones que busquen mínimos de una fila y máximos de una columna. Como se muestra en la ilustración 1, la primer función recibe una matriz, una posición y el tamaño de la matriz, dada la posición en la fila de tamaño *tam,* se itera de 1 a tam – 1, comparando en cada una si la variable auxiliar es menor a la siguiente posición en la fila, si es menor, el auxiliar cambiará al valor con el que fue comparado. Al terminar, la función retorna el valor mínimo en la fila.

```
function encontrarMaximoColumna(Matriz, columna, tam)
    aux = Matriz[0][columna]

    for i from 1 to tam - 1 do
        if aux < Matriz[i][columna] then
            aux = Matriz[i][columna]
        end if
    end for

    return aux
end function
```

Ilustración 2. Encontrar el máximo de una columna

El algoritmo de la ilustración dos aborda el problema de encontrar el máximo de una columna, es casi idéntica a la función anterior, solo que esta itera a través de las columnas, regresa el mayor

de la columna, con estas dos funciones podremos construir otra función que busque los puntos sillas de una matriz.

```
function encontrarPuntoSilla(Matriz, tam)
    int i, j, minFila

    for i from 0 to tam - 1 do
        minFila = encontrarMinimoFila(Matriz, i, tam)

        for j from 0 to tam - 1 do
            if Matriz[i][j] == minFila then
                if Matriz[i][j] == encontrarMaximoColumna(Matriz, j, tam) then
                    print "El punto silla se encuentra en la fila i y la columna j con valor Matriz[i][j]"
                    return 0
                end if
            end if
        end for
    end for

    return -1
end function
```

Ilustración 3. Búsqueda Punto Silla

Para encontrar el punto silla de una matriz se necesitan ambas funciones anteriores, como se muestra en la ilustración 3, primero, anidaremos ciclos for para desplazarnos a través de las filas y las columnas, el primero señalaran las filas y el segundo las columnas, buscaremos el mínimo en la fila y cuando se encuentre el mínimo en esa columna se comprobará que es el máximo de su columna, si lo es, se muestra la posición y el valor terminando el programa, en caso contrario seguirá adelante.

```
function encontrarPuntoSilla(Matriz, tam)
    totalPuntosSilla = 0

    for i from 0 to tam - 1 do
        minFila = encontrarMinimoFila(Matriz, i, tam)

        for j from 0 to tam - 1 do
            if Matriz[i][j] == minFila then
                if Matriz[i][j] == encontrarMaximoColumna(Matriz, j, tam) then
                    print "El punto silla se encuentra en la fila i y la columna j con valor Matriz[i][j]"
                    totalPuntosSilla = totalPuntosSilla + 1
                end if
            end if
        end for
    end for

    print "Total de puntos silla: totalPuntosSilla"
end function
```

Ilustración 4. Búsqueda total de puntos silla

Si se quiere encontrar todos los puntos sillas de una matriz necesitaremos recorrer toda la matriz, aunque se haya encontrado algún punto silla, para eso modificaremos el primer algoritmo tal que, tan solo quitando el retorno, dependiendo si existen más de uno, uno o ninguno, el total de puntos sillas en la matriz, en las ilustración 4 se muestra un ejemplo.

El algoritmo siguiente resuelve la búsqueda del primer punto máximo local de un arreglo teniendo en cuenta que podría existir o no, es un algoritmo fácil de implementar, en cada iteración comprobamos si $\boldsymbol{Arreglo[i-1] < Arreglo[i] < Arreglo[i+1]}$ es verdadero, si no, aumenta i en uno, tal como se observa en la ilustración 5.

```
Function encontrarPrimerMaximoLocal(A, tam)
    int i

    For i from 1 to tam - 2 do
        If A[i-1] < A[i] AND A[i] > A[i+1] then
            Print "El primer máximo local se encuentra en la posición (i+1), valor: A[i]"
            Return i
        End If
    End For

    Print "No se encontró ningún máximo local."
    Return -1
End Function
```

Ilustración 5. Punto Máximo Local A

Para la segunda parte, la búsqueda de todos los puntos máximos locales, quitaremos el retorno, de tal manera que la búsqueda continua, aunque se haya encontrado un punto, si en caso de no haber encontrado ningún punto máximo local el programa imprimirá un aviso y se detendrá tal como se muestra en la ilustración 6.

```
Function encontrarTodosLosMaximosLocales(A, tam)
    int totalMaximosLocales = 0
    int i

    For i from 1 to tam - 2 do
        If A[i-1] < A[i] AND A[i] > A[i+1] then
            Print "Máximo local encontrado en la posición i, valor: A[i]"
            totalMaximosLocales++
        End If
    End For

    If totalMaximosLocales == 0 then
        Print "No se encontró ningún máximo local."
    Else
        Print "Se encontraron totalMaximosLocales máximos locales en total."
    End If
End Function
```

Ilustración 6. Máximos Locales B

3 Experimentación y Resultados

En la práctica, se analizará cómo se comportan los algoritmos, analizando el mejor y el peor caso. Primer punto silla, el punto silla en una matriz es tal que un elemento en la matriz sea el menor en su fila y el mayor en su columna, el algoritmo buscará fila por fila el menor y cuando lo encuentre, buscará en su columna si es el mayor, si no, se descarta y continúa la búsqueda. El mejor caso es cuando registra el menor tiempo en acabar o el menor número de pasos que registra el algoritmo, y en este algoritmo, el mejor caso es cuando el punto silla esta en la primer posición. Notemos que cuando crece el tamaño de la matriz también crece el numero de pasos que tiene una matriz.

En la figura 7 se muestra el crecimiento del número de pasos con respecto al tamaño que tiene la matriz, para generar la gráfica se usó 500 registros los cuales crecieron de manera lineal, se esperaba un crecimiento lineal pues solo recorrería una fila y una columna, no buscaría a través de toda la matriz, incluso si el tamaño de la matriz varía.

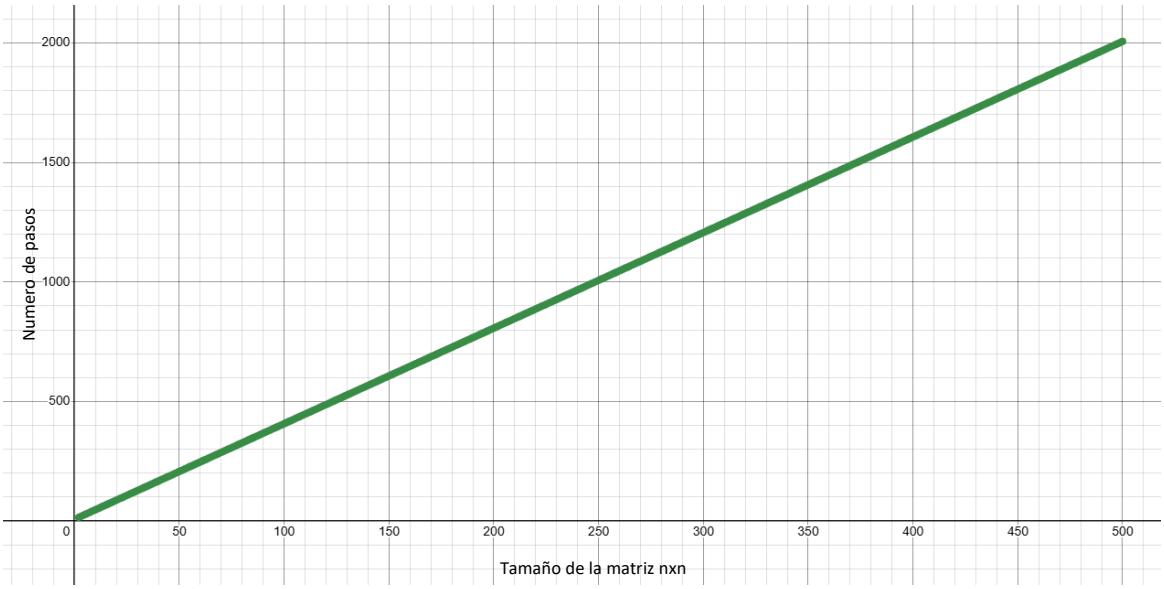

Ilustración 7. Crecimiento del algoritmo de búsqueda del primer punto silla

Para acotar por abajo a la función que generamos, buscamos una función tal que sea constante, esta función es f(x) = 4x, en la gráfica no aparece, pero como veremos en la ilustración numero 8 la función permanece constante debajo de la gráfica que describe al algoritmo.

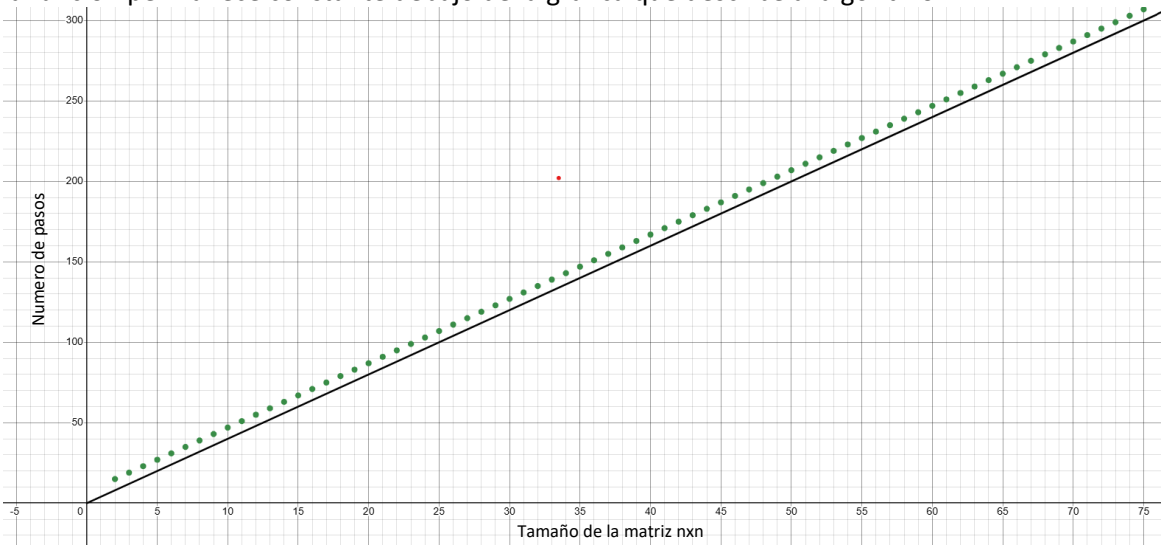

Ilustración 8. Cota inferior del algoritmo de búsqueda.

El pero caso resulta cuando no se encuentra el punto silla o se encuentra en la ultima posición, graficamos ambos casos como se muestra en la ilustración, el resultado hasta antes de los 100 registros fue que el peor caso y los casos aleatorios donde en su mayoría no se encontraba ningún punto silla fueron casi iguales, son ninguna distinción, sin embargo, cuando se extiende hasta los 500 registros el peor caso resulta ser cuando no se encuentra ningún punto silla, como se muestra en la ilustración 9, donde la grafica color rojo señala el caso donde el punto silla se encuentra en hasta la última posición de la matriz y la gráfica color azul representa cuando la matriz genera números aleatorios con una baja posibilidad de generar un punto silla, por lo tanto, tomaremos el peor caso donde la matriz no encuentra ningún punto silla. Para acotar esta función en color verde se usará $f(x) = 15x^2$como se muestra en la ilustración 9.

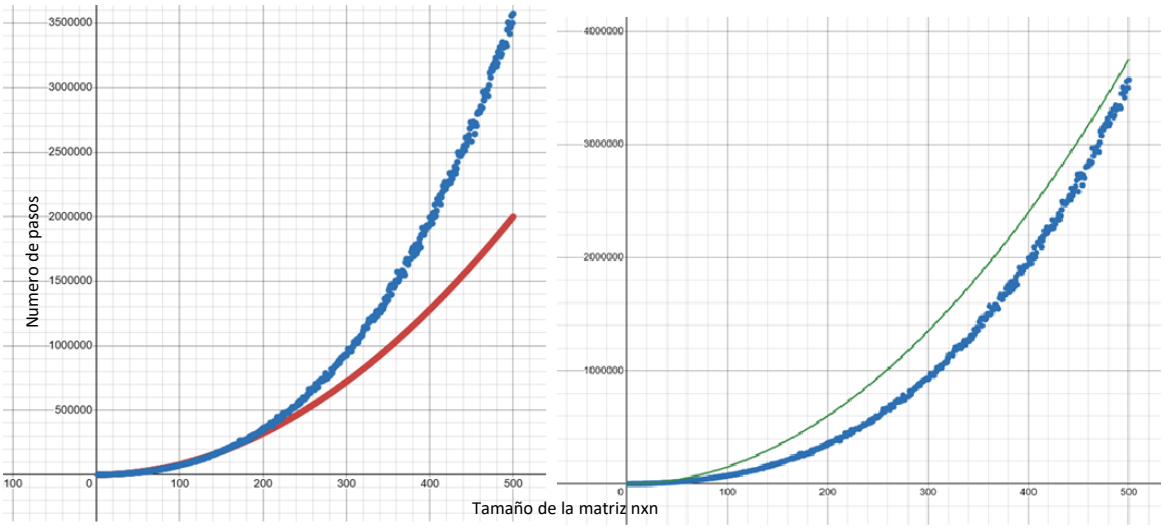

Ilustración 9. Comparación entre peores casos del algoritmo punto silla.

Colocando todos los casos y delimitando por una cota inferior y superior, la grafica es la que se presenta en la ilustración numero 10, observamos la gran diferencia entre uno y otro caso, el cual, para 500 registros las diferencias resultan evidentes.

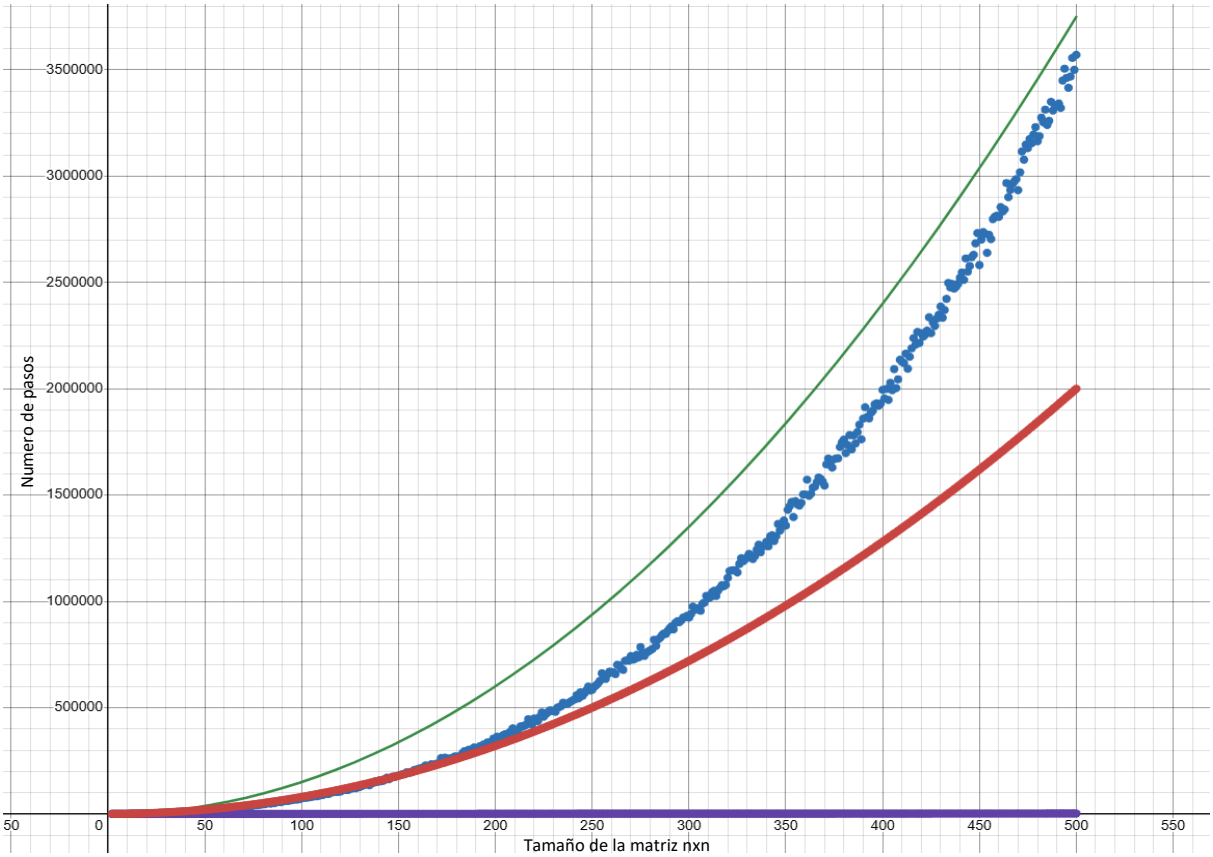

Ilustración 10. Comparación de casos acotados por arriba y por abajo.

Para la siguiente parte, el algoritmo deberá encontrar todos los puntos sillas existentes, la diferencia entre el mejor y el peor caso esta vez cambia, el mejor caso será cuando no se encuentre ningún punto silla y el peor caso es cuando encuentre n puntos sillas donde n = m que es la dimensión de la matriz M_{mxn}, en la figura 11 se presenta el mejor caso, a comparación del algoritmo

anterior este supera el número de pasos, el mejor caso en el algoritmo anterior era del orden $\Omega(\mathrm{n})$ y el actual es del orden $\Omega(n^2)$.

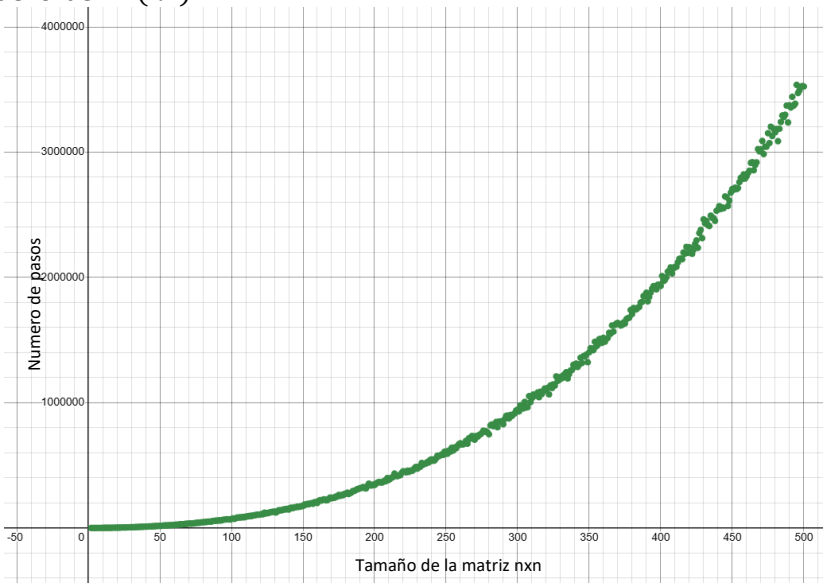

Ilustración 11. Mejor caso de búsqueda total de punto silla.

Para acotar por abajo a la funcion en el mejor caso, colocamos a $f(x) = 6x^2\{0 < x < 500\}$ y un $n_0 = 2$, observamos en la ilustración número 12 como se comporta la función con respecto al registro de datos.

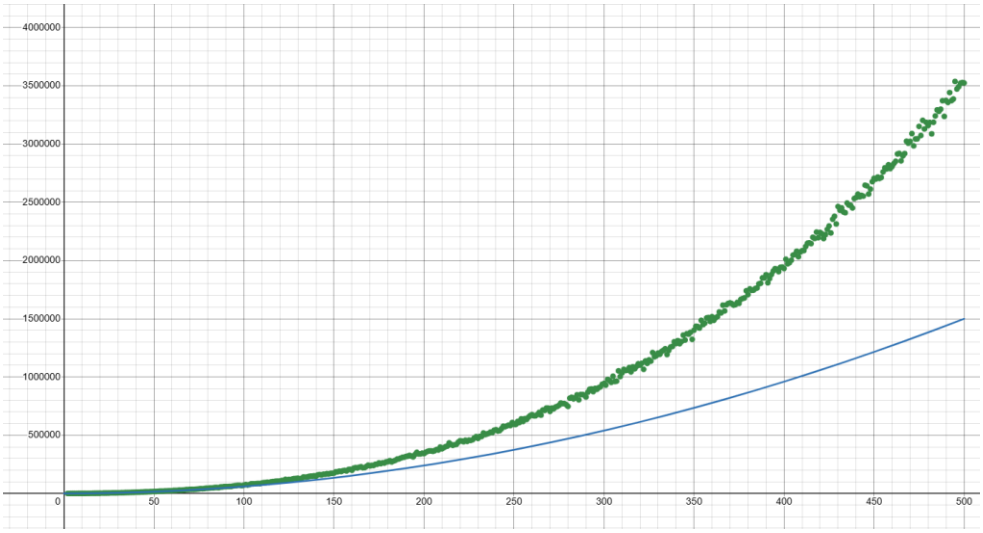

Ilustración 12. Mejor caso con cota inferior.

El peor caso se desarrolla cuando la matriz tiene n puntos silla, al iterar constantemente en todas las filas y columnas además de comprobar las posiciones, el algoritmo crece de manera cúbica, a comparación del mejor caso que fue cuadrática, el peor caso a los 500 registros alcanza los 251,251,005 pasos, y en el cuadrático alcanzo 3,524,566 pasos, una diferencia de 247,726,439 pasos.

En la ilustración siguiente ilustración en la parte izquierda acotamos por arriba a la función con $f(x) = 3x^2$ en color negro y una $n_0 = 2$, además se observa mucho mejor la diferencia en la parte derecha de la ilustración entre ambos, donde una incluso parece crecer linealmente.

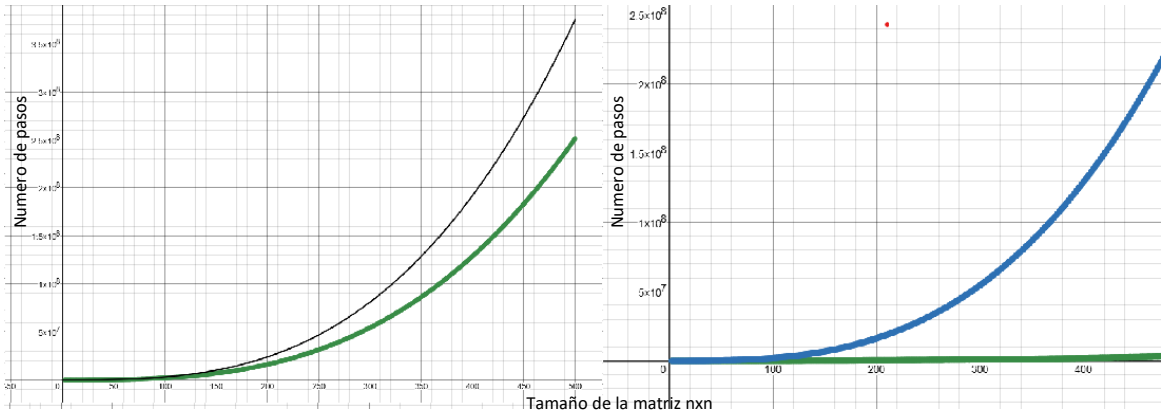

Ilustración 13 Peor Caso búsqueda total de punto silla

En la ilustración 14, se observa las funciones delimitadas por una cota superior y una inferior, la diferencia entre un caso y otro resultó ser significativamente más grande y aunque ambos casos la matriz se recorre completamente el total de pasos aumenta cuando existen n puntos sillas lo que aumenta el costo temporal de completar el algoritmo.

El siguiente algoritmo es encontrar el punto máximo local como lo planteamos en el apartado de conceptos básicos, para el mejor caso observamos que es cuando se busca en la primer posición y el peor es cuando no se encuentra ningún máximo local, como veremos a continuación, el mejor caso permanece constante y el peor es una función lineal, con una cota inferior con f(x) = 3 y cota superior con f(x) = 3x, como se muestra en la ilustración 14.

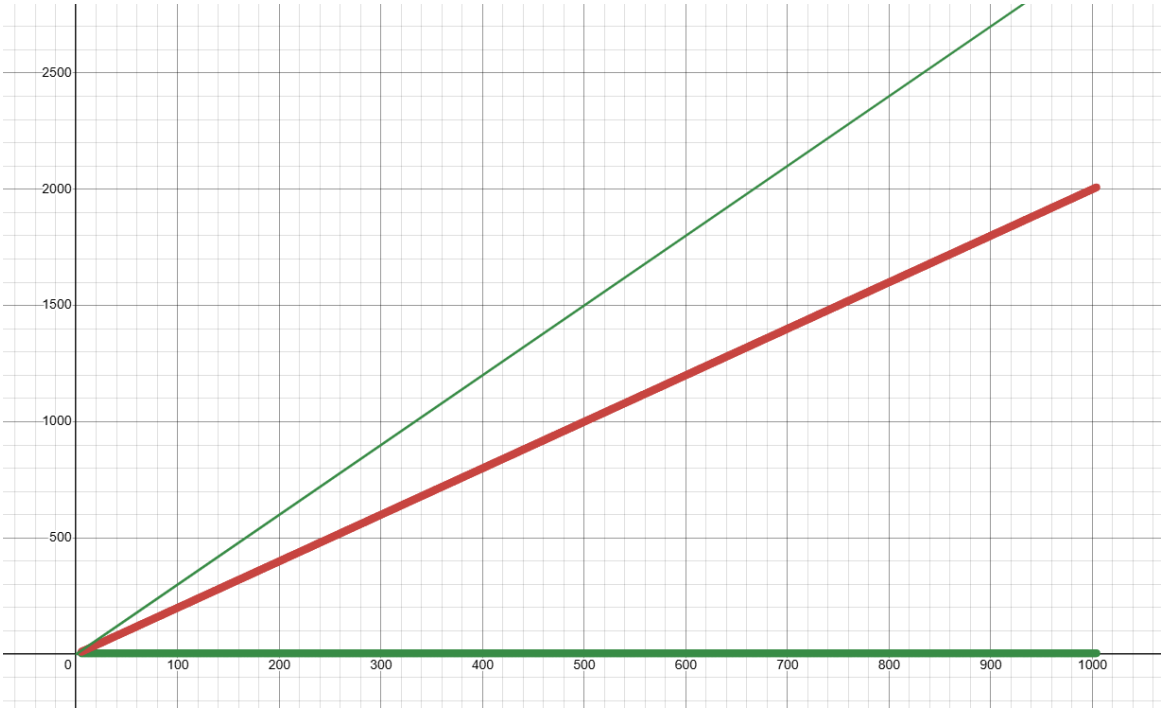

Ilustración 14 comparación de búsqueda del primer máximo local.

En la segunda parte, el mejor caso es cuando no se encuentra ningún punto local máximo en todo el arreglo y el peor caso es cuando se encuentran n puntos máximos locales, como se ilustra en la ilustración 15, donde el peor caso se ilustra con el color azul y el peor caso con el color verde, la complejidad varia por muy poco y resulta lineal la cual la delimitaríamos con una notación tipo theta. La cota inferior es igual a la función f(x) = 1.5x representada con el color negro y la función f(x) = 3.5x es la cota superior de color morado.

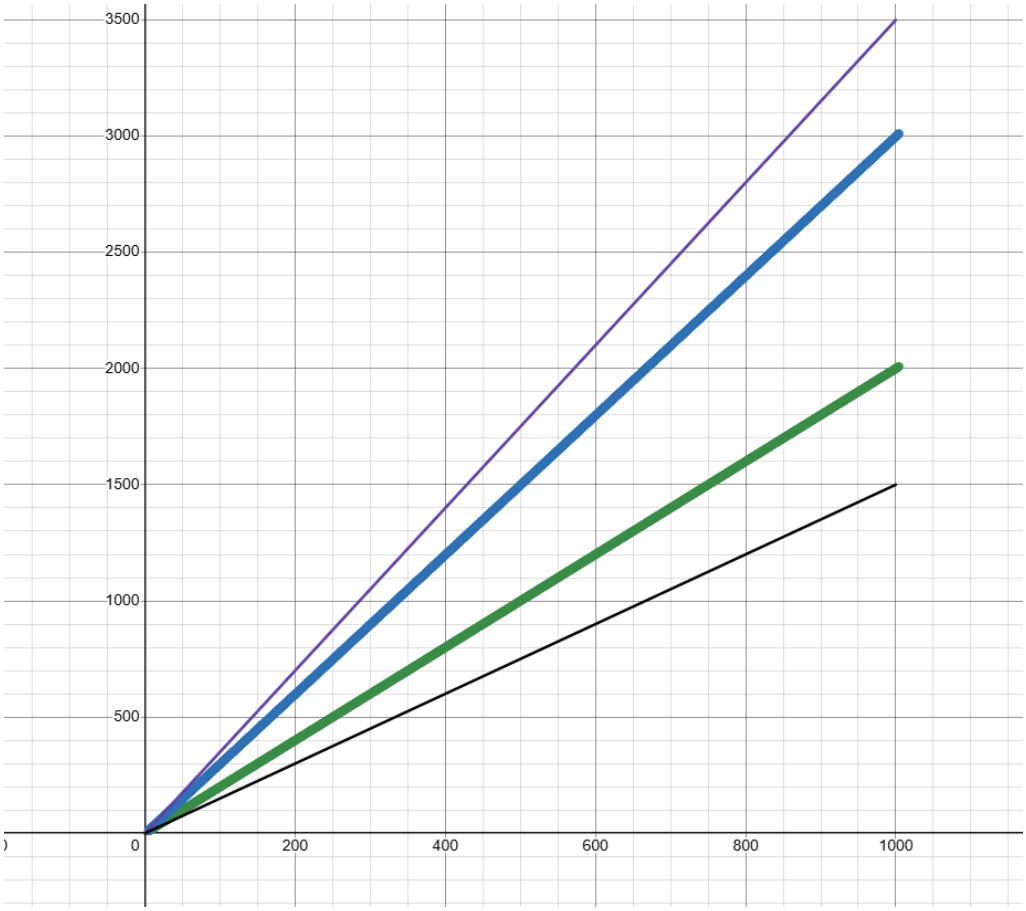

Ilustración 15. Comparación del algoritmo de búsqueda de todos los puntos máximos locales

4 Conclusiones

Conclusión general: Como se ha visto durante el desarrollo de la práctica, el análisis de estos algoritmos ha permitido conocer cómo se comportan de acuerdo con la evaluación de diferentes entradas de datos. Las gráficas revelaron a que tipo de complejidad pertenecen los diferentes casos, y concuerda con lo que se esperaba ya que en el mejor caso se obtuvo una función lineal y en el peor de los casos una función cuadrática, así mismo como en otras pruebas se observó que los puntos están ubicados entre ambas graficas de las funciones. El único problema que se tuvo fue en el algoritmo de punto silla al buscar obtener la gráfica lineal, pero al modificar algunas cosas en el código se pudo corregir este problema. Por lo tanto, se concluye que los resultados obtenidos concuerdan con el análisis a priori.

Conclusiones López Domínguez Daniel Efraín: En conclusión, el análisis de los algoritmos usando las notaciones asintóticas Θ, O y Ω nos ayuda a entender cómo crecen los algoritmos a medida que aumentan los datos. El algoritmo para buscar puntos silla en una matriz tiene una complejidad cuadrática en el mejor caso y puede ser mucho más lento en el peor caso, con un crecimiento cúbico. Por otro lado, el algoritmo de máximos locales es más eficiente, con un crecimiento constante en el mejor caso y lineal en el peor. Este tipo de análisis es útil para predecir el rendimiento de los algoritmos y elegir los más eficientes dependiendo de la situación.

Conclusiones Vite Valois Omar Abdiel: De acuerdo con lo hecho en la práctica pude

comprender de manera más profunda el funcionamiento de los algoritmos, ya que de forma práctica se nota más cómo se comportan las cosas en diferentes casos y la diferencia entre estos mismos, las gráficas y todos los elementos ayudaron a profundizar en la teoría de lo visto en clase.

5 Bibliografía

Cormen, T. & Leiserson, C. (2009). Introduction to Algorithms. The MIT Press.

Dasgupta, S., Papadimitriou, C. & Vazirani, U. (2006). Algorithms. McGrawHill Education

Calvo J. (2022). La complejidad de los algoritmos. Europeanvalley. https://europeanvalley.es/noticias/la-complejidad-de-los-algoritmos/