

LA SUMA X-OR

Juan Carlos Flores Mora, Miguel Alejandro Flores Sotelo, Sergio de Jesús Castillo Molano

Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria de Ingeniería Campus Tlaxcala

21 de Junio del 2024

jfloresm2202@alumno.ipn.mx

mflores2200@alumno.ipn.mx

scastillom2200@alumno.ipn.mx

I. INTRODUCCIÓN

El algoritmo XOR tiene como objetivo calcular una suma especial definida usando programación dinámica y fuerza bruta. En este contexto, se utiliza una matriz S de tamaño $n \times n$ para almacenar y manipular valores basados en un arreglo de entrada A . El procedimiento involucra iteraciones y cálculos específicos que permiten obtener un resultado acumulado, comenzando desde una posición L hasta R dentro de la matriz S .

El objetivo del algoritmo XOR es calcular una suma especial definida por las reglas específicas de un pseudocódigo dado. Utilizando una matriz cuadrada S de tamaño $n \times n$, inicializada y manipulada a partir de un arreglo de entrada A , el algoritmo busca acumular la suma de ciertos elementos de S desde una posición inicial L hasta una posición final R .

II. MARCO TEÓRICO

En el algoritmo de la suma X-OR de matrices hay muchos pasos a seguir para poder llegar al resultado que necesitamos, los cuales son: Fuerza bruta, programación dinámica, Matplotlib, generación de código en Python.

A continuación, se explica el concepto de cada uno de los pasos para poder llegar a los resultados empleados y evaluados para el desarrollo de la práctica:

Fuerza bruta: Este método implica calcular la distancia entre cada par de puntos en el conjunto y seleccionar el par con la distancia mínima. Aunque su implementación es sencilla, su complejidad computacional es $O(n^2)$, donde n es el número de puntos en el conjunto.

Programación dinámica: es un enfoque eficiente para resolver problemas de optimización donde los subproblemas se superponen. Para resolver el problema de encontrar la distancia mínima entre cada par de puntos en el conjunto utilizando programación dinámica, primero ordenamos los puntos según su posición en el eje X . Luego, calculamos recursivamente las distancias mínimas entre los pares de puntos en cada mitad del conjunto y combinamos las soluciones para encontrar la distancia mínima global.

Técnicas de diseño algorítmico:

Programación Dinámica: Esta técnica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

Matplotlib: Es una biblioteca de Python ampliamente utilizada para crear visualizaciones estáticas, animadas e interactivas en 2D. Para este caso, se utiliza para la generación de gráficos a partir de arreglos bidimensionales que se emplean dentro de los programas, logrando así la representación de las pruebas con distintas multiplicaciones de matrices y visualizar el rendimiento para cada una de sus ejecuciones.

Programación en Python: Por medio de este lenguaje de programación, se toma en consideración la precisión de las operaciones de temporización, además de que el rendimiento del código es un poco más lento al ser Python un lenguaje interpretado. Por otro lado, se asegura que la implementación del código de como resultado las soluciones óptimas.

III. DESARROLLO

SUBPROBLEMAS:

En la secuencia $A_L + (A_L \oplus A_{L+1}) + \dots + (A_L \oplus A_{L+1} \oplus \dots \oplus A_R)$ se puede percatar que ésta intrínsecamente requiere del resultado del sumando anterior para realizar la consulta, simplificando la operación XOR entre dos operandos. En otras palabras, se puede obtener el resultado de la consulta calculando cada operación XOR entre dos números del arreglo. Esto se puede ilustrar mejor con un ejemplo:

Suponga que se quiere calcular del arreglo $A = [1, 7, 2, 6]$, con $L = 1$ y $R = 3$. La secuencia quedaría de la siguiente manera:

$$A_1 + (A_1 \oplus A_2) + (A_1 \oplus A_2 \oplus A_3)$$

Reemplazando los valores se tiene

$$1 + (1 \oplus 7) + (1 \oplus 7 \oplus 2) = 1 + (6) + (4) = 11$$

A partir de este procedimiento, se puede observar que se requiere del sumando con XOR anterior para así simplificar de la siguiente manera:

$$1 + (1 \oplus 7) + (1 \oplus 7 \oplus 2)$$

$$1 + (6) + (6 \oplus 2) = 1 + (6) + (4) = 11$$

En base a lo presentado, los subproblemas que se resuelven es la operación XOR entre dos operandos, en la que uno de ellos puede ser un elemento del arreglo ,o bien, puede ser resultado de una operación XOR anterior; y el otro es el número que sigue en la secuencia del arreglo.

FORMULA DE RECURRENCIA:

Una vez explicado lo anterior, para la formulación de la recurrencia se contempla la función XOR (A, L, j, R, n), donde A es el arreglo de números de n elementos, L marcando el inicio de la consulta y R su final; j como variable que permita hacer el recorrido de los elementos entre L y R, y n el número de elementos. Además, se considera una tabla de tamaño $n * n$, donde contiene la solución de cada operación XOR entre L y j.

Inicialmente, j tendrá el valor de L, por lo que surge el primer caso donde $L = j$, que se refiere cuando solamente se está seleccionado un solo elemento, es decir, el primer elemento A_L de la sumatoria; dado este caso, se retorna el valor del arreglo A en la posición L. Por otro lado, cuando $L \neq j$ y $j \leq R$, esto quiere decir que se está realizando una operación XOR y, por lo tanto, requiere del sumando anterior para realizar dicha operación.

Dicho esto, la recurrencia quedaría de la siguiente forma:

$$XOR(A, L, j, R, n) = \begin{cases} A_L & L = j \\ abs(XOR(A, L, j-1, R, n) - XOR(A, j, j, R, n)) & L \neq j \text{ y } j \leq R \end{cases}$$

ALGORITMO BOTTOM-UP

```
def xor(A, L, R, n):
    #Creación de la tabla - solución
    S = [[0 for _ in range(n)] for _ in range(n)]
    # Caso base
    for i in range(n):
        S[i][i] = A[i]
    # Calculo del resultado final
    resultado = S[L][L]
    for i in range(L + 1, R + 1):
        #Calculo de cada XOR
        S[L][i] = abs(S[L][i - 1] - S[i][i])
        #Suma del resultado de cada XOR
        resultado += S[L][i]
    return resultado
```

Dentro de los parámetros de la función XOR se encuentra: A siendo el arreglo de números con tamaño n, L que marca el inicio de la consulta, R que es señala hasta qué número se realiza la consulta, y n que es el número de elementos del arreglo.

Inicialmente, se crea la tabla de la solución S com valores iniciales 0 en todas sus celdas. Posteriormente, en S se colocan los casos bases de la recurrencia que, en este caso, sería los elementos del arreglo en la posición (L,L). Luego para retornar el resultado se hace la sumatoria de cada operación XOR, por lo que se empieza con el primer elemento de la consulta. Dentro del ciclo inicia con el siguiente número de la lista, donde en cada iteración realiza

la operación XOR y la suma al resultado. Finalmente, retorna el resultado de toda la sumatoria.

LLENADO DE TABLA:

Para realizar el llenado de la tabla, se retoma el ejemplo que se utilizó el mismo de la sección de SUBPROBLEMAS: A = [1, 7, 2, 6], L = 1 y R = 3. La tabla de solución S quedaría de la siguiente manera:

L \ j	1	2	3	4
1				
2	0			
3	0	0		
4	0	0	0	

Donde los índices de las filas corresponden a L y los de las columnas a j, variable que permite hacer las operaciones XOR y que es menor o igual a R, por lo que valores L mayores que R no son válidos y se quedan con valor 0 en esas casillas.

Tras esta ilustración, la tabla inicia el llenado de sus casos base, es decir, cuando $L = j$, de modo que su llenado lo hace en una sola diagonal de la tabla y asigna los números del arreglo.

L \ j	1	2	3	4
1	1			
2	0	7		
3	0	0	2	
4	0	0	0	6

Posteriormente, j se inicializa en 2 y comienza a realizar el cálculo de las operaciones XOR, en este caso, comienza calculando $(A_1 \oplus A_2)$, quedando de la siguiente manera:

$$\begin{aligned} (A_1 \oplus A_2) \\ S[L][j] &= abs(S[L][j - 1] - S[j][j]) \\ S[1][2] &= abs(S[1][2 - 1] - S[2][2]) \\ S[1][2] &= abs(S[1][1] - S[2][2]) \\ S[1][2] &= abs(1 - 7) \\ S[1][2] &= 6 \end{aligned}$$

L \ j	1	2	3	4
1	1	6		
2	0	7		
3	0	0	2	
4	0	0	0	6

Ahora j aumenta le suma 1 a su valor actual, es decir, $j = 3$. Aplicando el mismo procedimiento, se obtiene:

$$\begin{aligned} S[L][j] &= abs(S[L][j - 1] - S[j][j]) \\ S[1][3] &= abs(S[1][3 - 1] - S[3][3]) \\ S[1][3] &= abs(S[1][2] - S[3][3]) \\ S[1][3] &= abs(6 - 2) \\ S[1][3] &= 4 \end{aligned}$$

L \ j	1	2	3	4
1	1	6	4	
2	0	7		
3	0	0	2	

4	0	0	0	6
---	---	---	---	---

Finalmente, se hace la suma de los resultados de la fila 1, es decir, $S[1][1] + S[1][2] + S[1][3] = 1 + 6 + 4 = 11$, dando así el mismo resultado que se obtuvo en la sección de SUBPROBLEMAS.

PROPUESTA TOP-DOWN

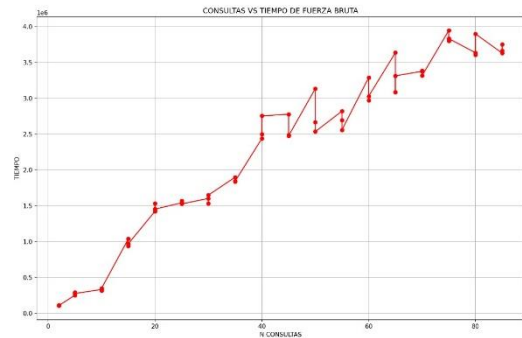
```

XOR(A, L, R, n) {
    // Declaración de la matriz S de tamaño n x n
    s[n][n]
    // Función recursiva para calcular s[i][j]
    funcion calcularS(i, j) {
        si (i == j) {
            s[i][j] = A[i]
        } sino {
            s[i][j] = abs(s[i][j-1])
        }
    }
    // Inicialización de la matriz S y cálculo según sea necesario
    para i = 1 hasta n {
        calcularS(i, i)
        para j = i + 1 hasta n {
            calcularS(i, j)
        }
    }
    // Cálculo del resultado acumulando valores desde L hasta R
    resultado = 0
    para i = L hasta R {
        resultado += s[i][i]
    }
    // Retornar el resultado final
    Return resultado
}

```

IV. RESULTADOS

GRAFICAS “Evaluación de la eficiencia de algoritmos de Fuerza Bruta y Top-down en función de las consultas vs tiempo”



GRAFICA 1.1 Figura 1 evaluación de eficiencia Tamaño de matrices vs tiempo (FUERZA BRUTA)

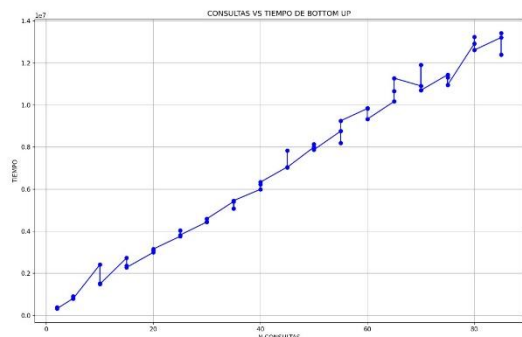
Eje X (horizontal): Representa el número de consultas

Eje Y (vertical): Representa el tiempo de ejecución en segundos.

La gráfica muestra una serie de puntos (representados con círculos rojos) conectados por una línea roja.

La tendencia general de la gráfica es ascendente, lo que indica que a medida que aumenta el número de consultas, también aumenta el tiempo requerido.

La gráfica parece tener una serie de incrementos con algunas variaciones, pero la tendencia general es creciente. A medida que el número de consultas aumenta, el tiempo de respuesta también incrementa, aunque no de manera completamente lineal.



GRAFICA 1.2 Figura 3 evaluación de eficiencia Tamaño de consultas vs tiempo (bottom-up)

Eje X (horizontal): Representa el número de matrices

Eje Y (vertical): Representa el tiempo de ejecución en segundos.

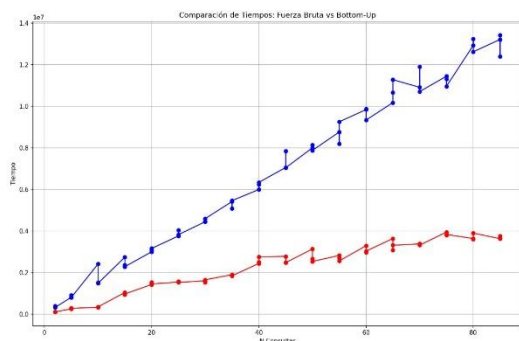
La gráfica muestra una serie de puntos (representados con círculos azules) conectados por una línea azul. La tendencia general de la gráfica es ascendente, lo que indica que a medida que aumenta el número de consultas, también aumenta el tiempo requerido.

COMPARACION:

La serie roja (Fuerza Bruta) muestra una tendencia ascendente, pero con un crecimiento más lento y menos pronunciado en comparación con la serie azul.

La serie azul (Bottom-Up) muestra una tendencia ascendente más pronunciada y consistente, indicando que a medida que el número de consultas aumenta, el tiempo de respuesta incrementa de manera más notable.

La comparación indica que el método de fuerza bruta toma menos tiempo para procesar un número creciente de consultas en comparación con el método bottom-up, según los datos presentados.



GRAFICA 1.3 Figura 3 evaluación de eficiencia Tamaño de matrices vs tiempo (FUERZA BRUTA VS BOTTOM-UP)

La gráfica compara los tiempos de procesamiento entre los métodos de fuerza bruta y bottom-up para un número creciente de consultas. El método de fuerza bruta parece ser más eficiente en términos de tiempo de procesamiento para el rango de consultas presentadas en la gráfica.

//PRUEBAS DE FUNCIONAMIENTO

//Datos de entrada

```
datos[1].txt
Archivo  Editar  Ver

4 10
1 7 2 6
1 2
1 3
1 4
2 3
2 4
3 4
4 4
1 1
2 2
3 3
```

//DATOS DE SALIDA

Resultados de FUERZA BRUTA:

Consulta 1: Suma XOR de array[1] a array[2] = 7

Consulta 2: Suma XOR de array[1] a array[3] = 11

Consulta 3: Suma XOR de array[1] a array[4] = 13

Consulta 4: Suma XOR de array[2] a array[3] = 12

Consulta 5: Suma XOR de array[2] a array[4] = 13

Consulta 6: Suma XOR de array[3] a array[4] = 6

Consulta 7: Suma XOR de array[4] a array[4] = 6

Consulta 8: Suma XOR de array[1] a array[1] = 1

Consulta 9: Suma XOR de array[2] a array[2] = 7

Consulta 10: Suma XOR de array[3] a array[3] = 2

Resultados de BOTTOM-UP:

Consulta 1: Suma XOR de array[1] a array[2] = 7

Consulta 2: Suma XOR de array[1] a array[3] = 11

Consulta 3: Suma XOR de array[1] a array[4] = 13

Consulta 4: Suma XOR de array[2] a array[3] = 12

Consulta 5: Suma XOR de array[2] a array[4] = 13

Consulta 6: Suma XOR de array[3] a array[4] = 6

Consulta 7: Suma XOR de array[4] a array[4] = 6

Consulta 8: Suma XOR de array[1] a array[1] = 1

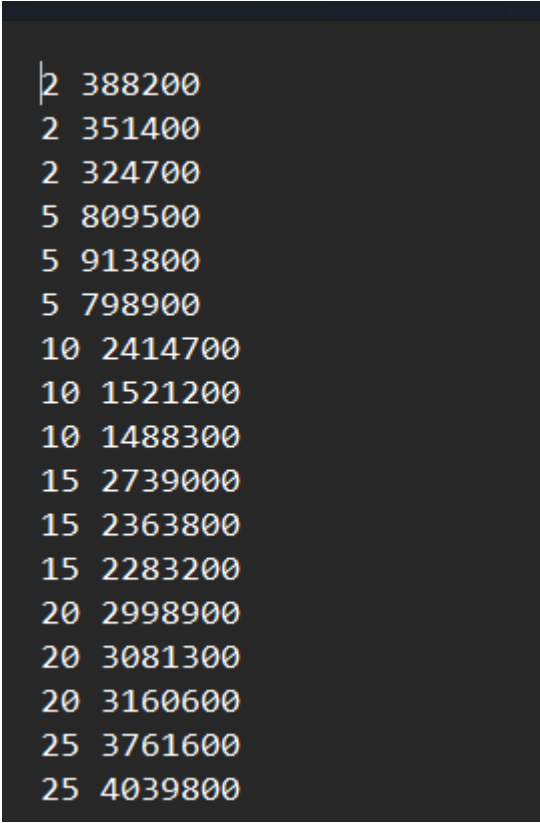
Consulta 9: Suma XOR de array[2] a array[2] = 7

Consulta 10: Suma XOR de array[3] a array[3] = 2

//Tiempos de Ejecución fuerza bruta

```
2 104600
2 113900
2 109900
5 251700
5 293100
5 276700
10 332800
10 318200
10 351000
15 1038900
15 940700
15 971400
20 1424400
20 1524500
```

Tiempo de ejecución button_up



2	388200
2	351400
2	324700
5	809500
5	913800
5	798900
10	2414700
10	1521200
10	1488300
15	2739000
15	2363800
15	2283200
20	2998900
20	3081300
20	3160600
25	3761600
25	4039800

- [2] Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms*. McGraw-Hill Education.
- [3] Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
- [4] Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.

V. CONCLUSIÓN

La comparación detallada entre los métodos de fuerza bruta y bottom-up, basada en la gráfica proporcionada, revela que el método de fuerza bruta es más eficiente en términos de tiempo de procesamiento para el rango de consultas analizado. Esta conclusión destaca la importancia de considerar el contexto específico y las características del problema al elegir el método de procesamiento más adecuado. Aunque el método bottom-up ofrece una estructura optimizada, su eficiencia puede no ser evidente en todos los escenarios, especialmente en aquellos con un número moderado de consultas. Por lo tanto, la elección del método debe basarse en una evaluación cuidadosa de los requisitos del problema y las características del entorno de aplicación.

V. REFERENCIAS

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press..