Reporte de practica

Luis Fernando Nuñez Rangel

Reporte de practica sobre el analisis de la complejidad de los algoritmos mediante la notación Big O

Fibonacci

Fibonacci recursivo

Comenzaremos analizando el metodo fibonacci recursivo, el cual acorde a lo visto el la fig 1. Tiene una complejidad exponencial O(n²), pues el número de operaciones aumenta de forma exagera conforme se incrementa el numero de la entrada. Realizandose operaciones innecesarias dentro del propio algoritmo.

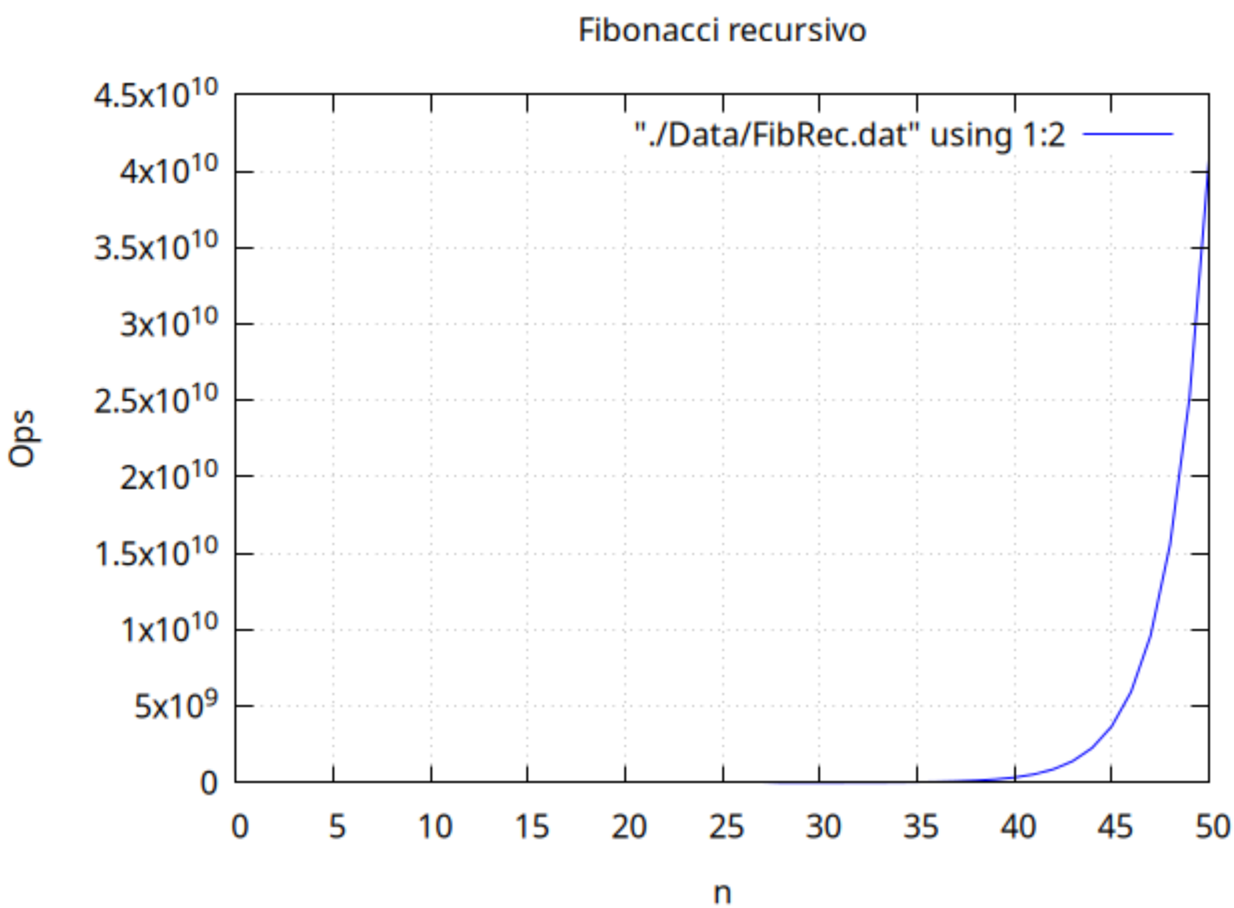

Figura 1

Fibonacci iterativo

Ahora analizemos el metodo iterativo, que a simple vista en codigo podria paracer mas complejo que el recursivo, sin embargo podemos apreciar que esto no es asi.

Pues mantiene una complejidad lineal O(n), pues el número de operaciones aumenta de forma proporcional con la entrada de los datos, es mejorable aun, sin embargo es un buen acercamiento a algo más eficiente.

Vease fig 2.

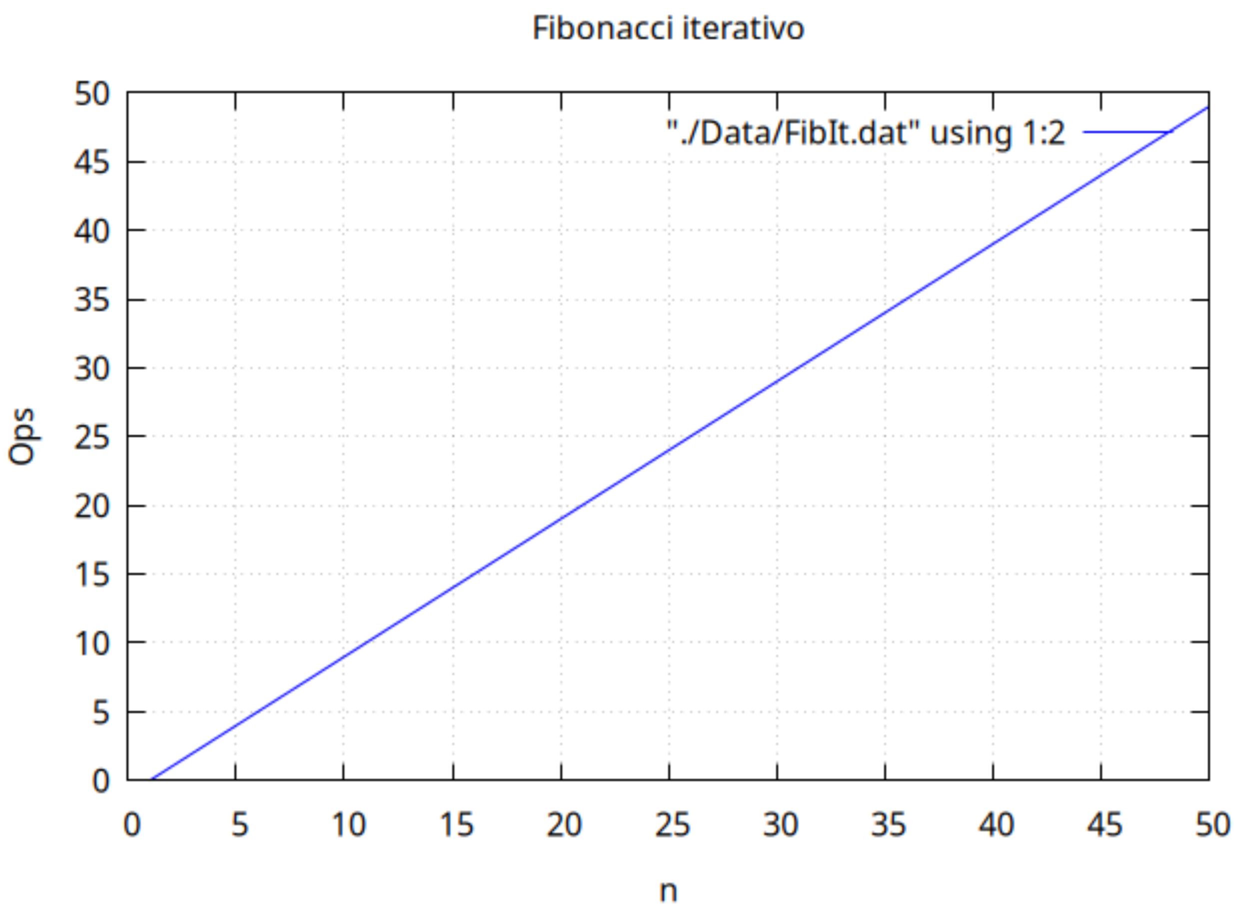

Figura 2

Triángulo de Pascal

El triángulo de pascal permite calcular de forma rapida y eficiente los coeficientes de un binomio elevado a la ***n*** ; $(a+b)^2 = a^2 + 2ab + b^2$, por poner un ejemplo.

Triángulo de Pascal Recursivo

Para el metodo recursivo observamos algo muy similar a lo antes visto con Fibonacci, un crecimiento exponencial $O(n^2)$ pues vemos que conforme la fila aumenta el numero de operaciones lo hace de igual manera, sin embargo cabe recalcar que esto lo hace con mayor tendencia a los numeros centrales del triángulo, pues logicamente, estos requiren por naturaleza más operaciones que los extremos. Vease Fig 3.

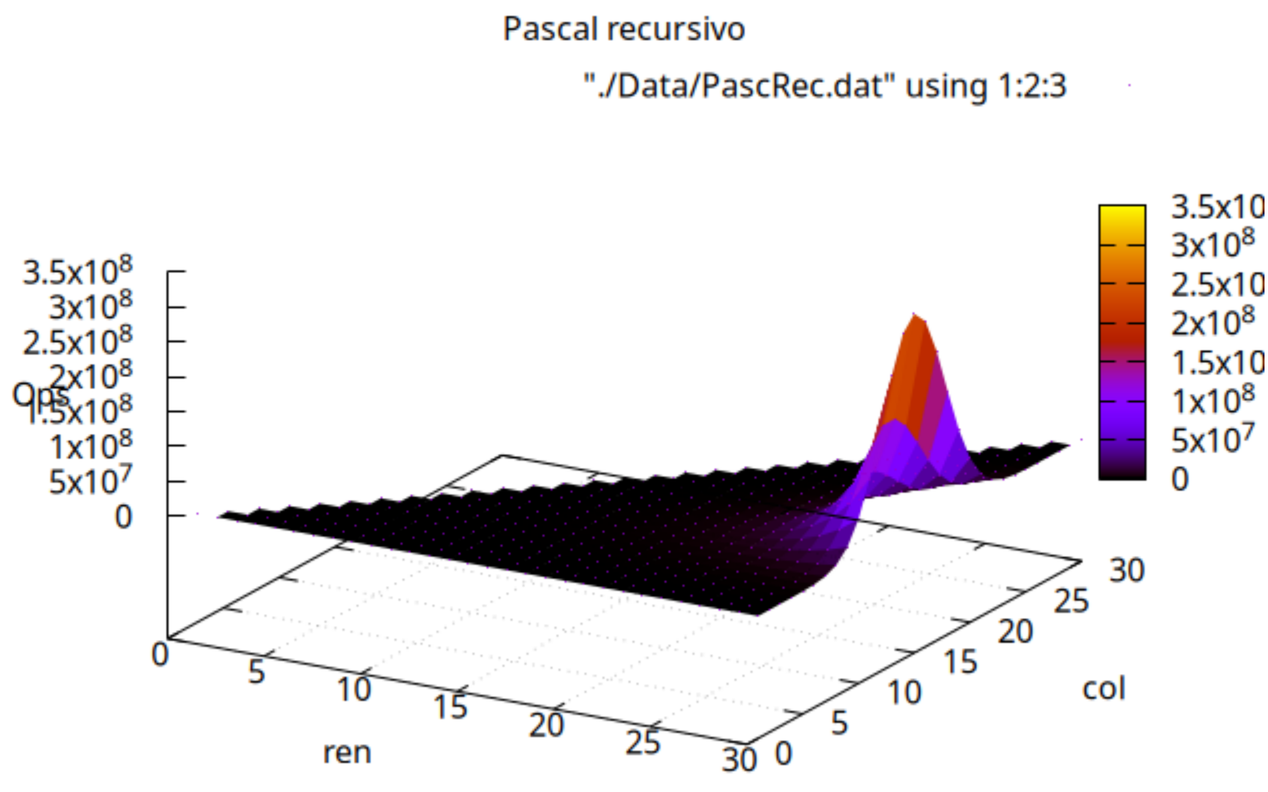

Figura 3

Triángulo de Pascal Iterativo

Este metodo resulta mas simple en complejidad; al igual que el metodo FibIt(n) mantiene una complejidad lineal O(n) mejorable aun para reducir su tiempo de ejecución, sin embargo igualmente

presenta una increible mejora con respecto al metodo recursivo.
Vease Fig 4.

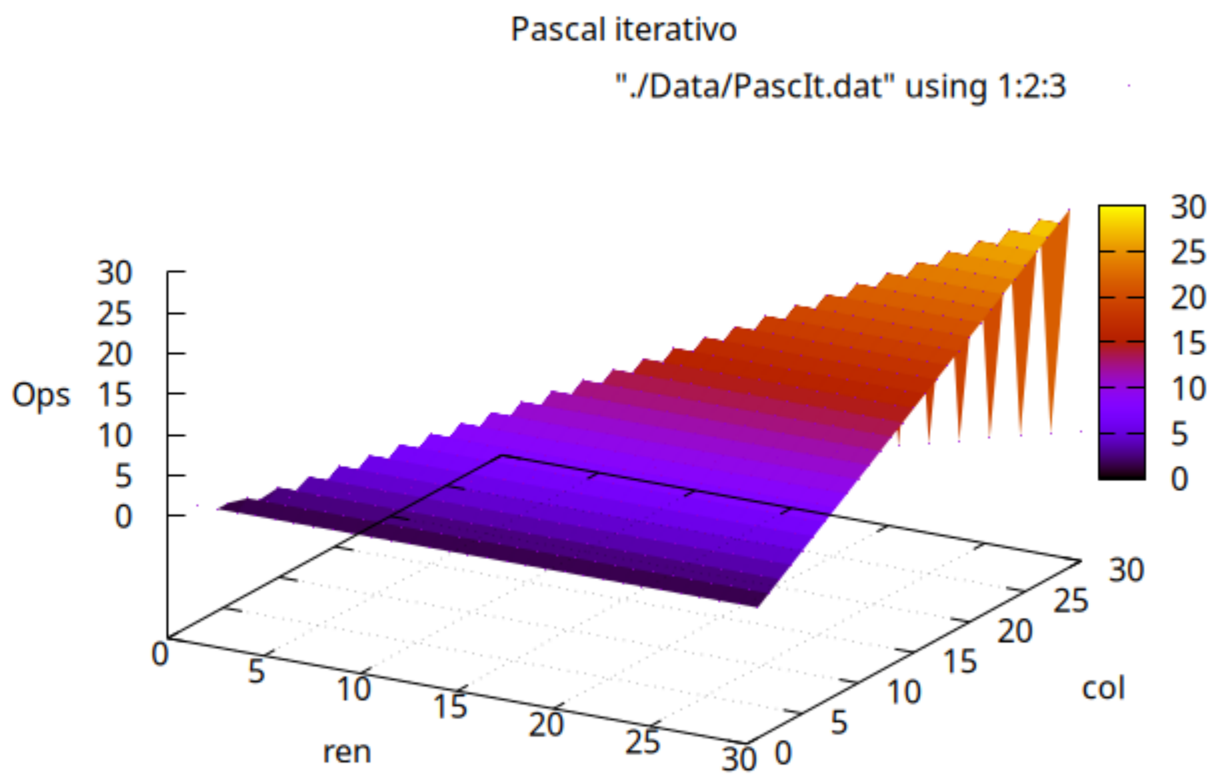

Figura 4

Preguntas

Para las versiones recursivas

¿Cual es el valor maximo de n que pudiste calcular para Fibonacci sin que se alentara la computadora?

- 80, La computadora se mantenia usable y sin desperfectos sin embargo a partir del 35 tardaba cada vez mas en calcular el siguiente elemento

¿Cuanto se tardó?

- Entre 10 y 15 minutos (Dependiendo si la tenia conectada a la corriente o no)

¿Cual es el valor maximo de ren que pudiste calcular para el triángulo de pascal sin que se alentara la computadora?

- 70, La computadora se mantenia usable pero mas de 70 comenzaba a dar trabas en la pc.

¿Cuanto se tardó?

- Aproximadamente 20 minutos

Justificacion a partir del codigo

Una vez que ya analizamos los resultados en las graficas y que podemos ver claramente la complejidad de cada algoritmo, veamos ahora el por que de esta misma en la estructura del codigo.

```
/// <summary>
/// Función auxiliar que permite calcular de forma recursiva el
/// n-esimo termino de la sucesión
/// </summary>
/// <param name="n"> El indice del elemento que se desea calcular
/// <returns>El n-esimo elemento de la sucesión de Fibonacci</returns>
private int FibonacciRecAux(int n){
    contador++;
    if (n<2) return n;
    return FibonacciRecAux(n-1) + FibonacciRecAux(n-2);
}
```

Como podemos apreciar en la estructura del codigo por cada llamada explicita a la función terminamos haciendo el doble demanera que al llegar finalmente a la condicion n<2 hemos llamado un enorme numero de veces la función, ademas el codigo es en parte ineficiente, pues hace algunos calculos mas de una sola vez.

En cambio analicemos el metodo iterativo que a simple vista podría parecer peor en nivel de complejidad al depender de un bucle sin embargo eh aqui del por que de su complejidad lineal.

```
/// <summary>
/// Función que permite calcular el n-esimo termino de forma iterativa
/// </summary>
/// <param name="n"> El indice del elemento que se desea calcular
/// <returns>El n-esimo elemento de la sucesión de Fibonacci</returns>
/// <exception cref="IndexOutOfBoundsException"> Si el valor de <code>n</code> es
/// invalido.</exception>
public int FibonacciIt(int n) {

    ArgumentOutOfRangeException.ThrowIfNegative(n);

    int current = 0;
    int low = 0;
    int fast = 1;

    contador = 0;

    for(int i=1;i<n;i++){
        current = low + fast;
        low = fast;
        fast = current;

        contador++;
    }

    return current;
}
```

Analizando el codigo vemos que el bucle for se ejecuta un total de n-1 veces, aunque si bien, la complejidad temporal crece conforme crece n, lo hace de una forma controlada y lineal, no exponencial.

Lo mismo sucede cuando analizamos el codigo del triangulo de pascal, tanto iterativo como el recursivo.

```
    /// <summary>
/// Método para calcular, de forma recursiva, el elemento en la fila
/// <code>i</code> y columna <code>j</code>
/// del triángulo de Pascal de forma recursiva
/// </summary>
///
/// <param name="ren">El número de fila.</param>
/// <param name="col">El número de columna.</param>
/// <returns>El elemento en la i-ésima fila y la j-ésima columna del
/// triángulo de Pascal.</returns>
private int TPascalRecAux(int ren, int col){
    contador++;
    // Casos fundamentales o triviales
    if (col == 0 || ren == col || ren == 0) return 1;
    return TPascalRecAux(ren-1,col-1) + TPascalRecAux(ren-1,col);
}
```

De una manera similar al caso de Fibonacci, vemos que a menos que unas condiciones especificas se cumplan, la funcion se llama a si misma 2 veces mas, por lo que su crecimiento será exponencial entre mas aumente ren.

Caso contrario al algoritmo iterativo, el cual depende de un solo bucle, por lo que su crecimiento será lineal, crece de manera proporcional con n

```
    /// <summary>
    /// Método para calcular, iterativamente, el elemento en la fila
    /// <code>i</code>, en la columna <code>j</code> del triángulo de Pascal.
    /// </summary>
    ///
    /// <param name="ren">El número de fila.</param>
    /// <param name="col">El número de columna.</param>
    /// <returns> El elemento en la i-ésima fila y la j-ésima columna del
    /// triángulo de Pascal.</returns>
    /// <exception cref="IndexOutOfBoundsException">Si los indices <code>i</code>
    /// <code>j</code> son inválidos. </exception>
    public int TPascalIt(int ren, int col){
        if (ren < 0 || col < 0 || (col > ren && ren < 5)) throw new ArgumentOutOfRangeE

        contador = 0;

        if (col == 0 || ren == col){
            contador++;
            return 1;
        }

        int number = 1;
        for (int i = 1; i <= col; i++) {
            number = number * (ren - i + 1) / i;
            contador++;
        }

        return number;
    }
``
```