Práctica 1

Escobar Rosales Luis Mario

29 de septiembre , 2021

La práctica consiste en lo siguiente:

- Parte I : La práctica consiste en implementar métodos que calculen el triángulo de pascal y el n- ésimo número de fibonacci, al tiempo que estiman el número de operaciones realizadas. Esto se programará en forma recursiva e iterativa. Se deberá implementar la interfaz IComplejidad en una clase llamada Complejidad. Se entregan pruebas unitarias para ayudar a verificar que estas funciones estén bien implementadas. Adicionalmente deberán llevar la cuenta del número de operaciones estimadas en un atributo de la clase para generar un reporte ilustrado sobre el número de operaciones que realiza cada método.
- Parte II :Gnuplot es una herramienta interactiva que permite generar gráficas a partir de archivos de datos planos. Para esta práctica, los datos deben ser guardados en un archivo de este tipo y graficados con gnuplot

Desarrollo :

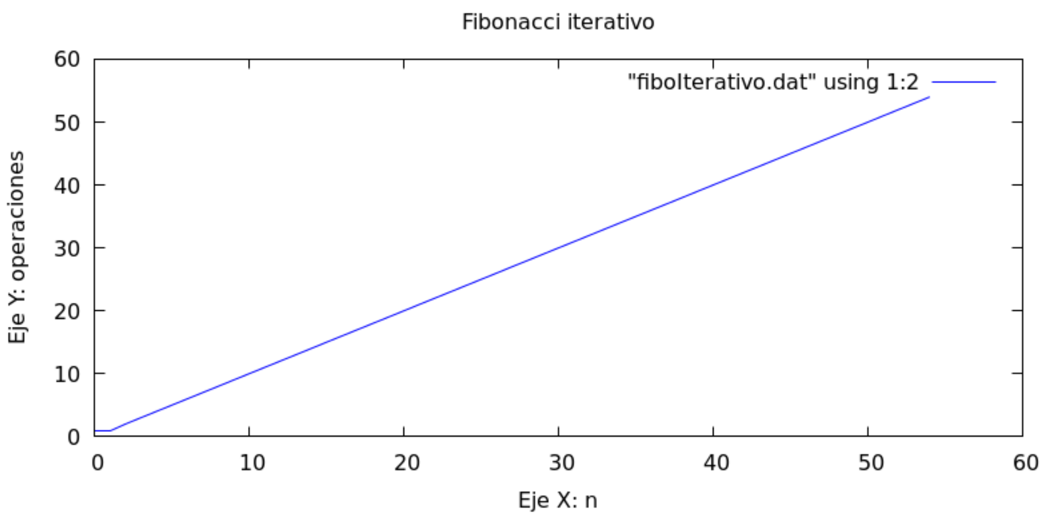
- Primero se creó la clase Complejidad donde se implementaran los métodos de la interfaz IComplejidad.
- Se agregaron 4 test en la clase ComplejidadTest, donde los resultados de los métodos se comparan con resultados calculados a mano.
- El primer método que fue implemenado fue fibonacci recursivo por su simpleza.Se agrego la función auxiliar privada para mejorar la implementación.
- El segundo método fue fibonacci iterativo, que consistió unicamente en un ciclo, donde los dos valores anteriores se iban sumando para obtener el actual, los valores iniciales realmente eran los casos base de fibo recursivo.
- Al igual que la implementación recursiva de fibonacci, la implementación recursiva para obtener un valor en el triángulo de Pascal, fue prácticamente directa.
- La última y más retadoraïmplementación fue Pascal iterativo.Para poder calcular el valor que se solicitaba dado los parámetros renglón, columna. Se necesito de una estructura auxiliar (un arreglo bidimensional) para ir guardando los datos calculados y facilitar el cálculo de la salida con su recorrido.

- Se Ejecutaron las pruebas unitarias con el comando ant test,corrigiendo los metódos donde los test habian fallado hasta que todas las pruebas fueran aprobadas
- Después se realizaron los metodos estatico escribeOperaciones, para poder guardar y escribir los datos de las salidas de los métodos
- Se creó la clase UsoComplejidad,donde está el metódo main. Aqui se ejecutan los metodos para generar los datos con sus respectivos archivos.
- Una vez recopilados los datos, se empieza el proceso de graficarlos con la herramienta gnuplot.

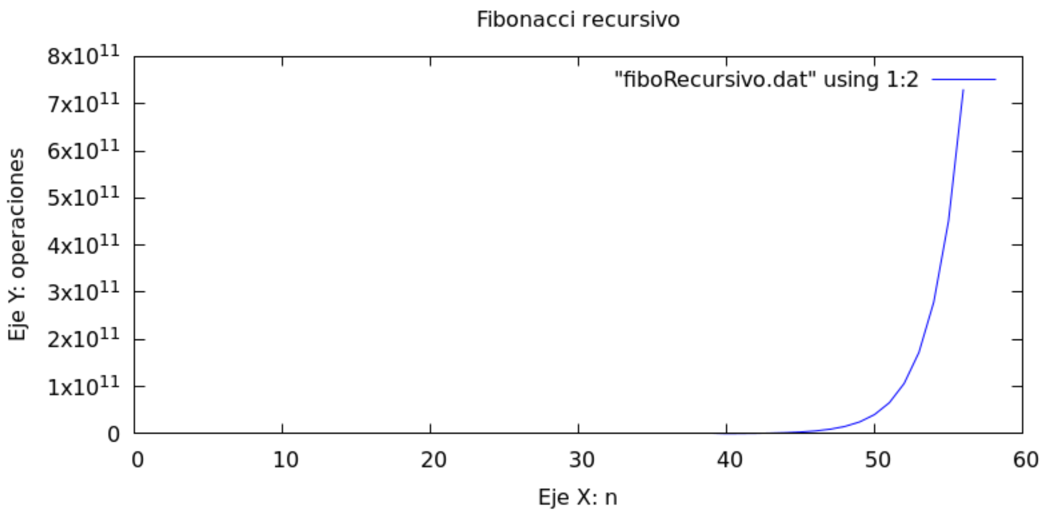
Graficas :

A partir de la generación de datos, de la salida del programa, se grafico la relación entra la entrada y el número de operaciones de cada método.

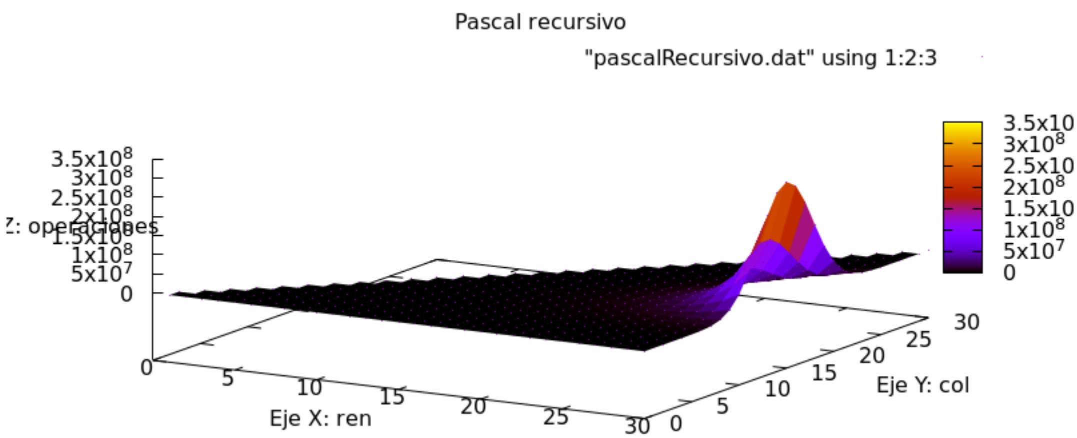
- Fibo Iterativo:



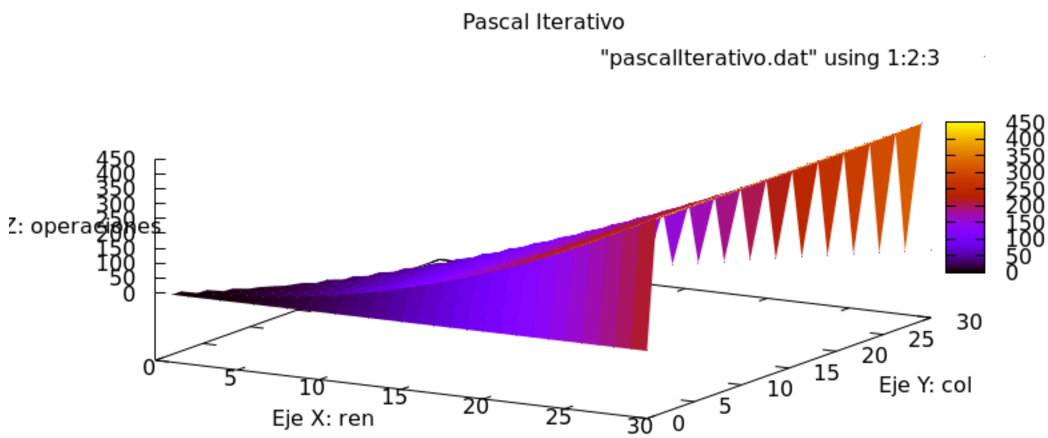
- Fibo Recursivo:



- Pascal recursivo 3D:



- Pascal iterativo 3D:



Preguntas

1.-Para las versiones recursivas:

- ¿Cuál es el máximo valor de n que pudiste calcular para fibonacci sin que se alentara tu computadora? (Puede variar un poco de computadora a compu- tadora (± 3), así que no te esfuerces en encontrar un valor específico).

  El máximo valor que mi computadora pudo calcular sin alentarse:, (fue precisamente el pico de datos de la grafica fiborecursivo), $n = 50$

- ¿Cuál es el máximo valor de ren que pudiste calcular para el triángulo de Pascal sin que se alentara tu computadora?

  Para Pascal recursivo fue un valor bastante cercano al de fibonacci siendo ren $= 45$

2.-Justifica a partir del código ¿cuál es el orden de complejidad para cada uno de los métodos que programaste?
Analicemos cada código para ver su orden de complejidad:
justificar con graficas, el crecimiento de las funciones

a) Fibo iterativo

```java
    @Override
    public int fibonacciIt(int n){
      if(n<0) throw new IndexOutOfBoundsException();
      int aux;
      int anterior = 0;
      int actual = 1;
      int i = 1;
      while(i < n){
        aux = anterior;
        anterior = actual;
        actual = anterior + aux;
        i++;
      }
      contador = i;
      return actual;
    }

}
```

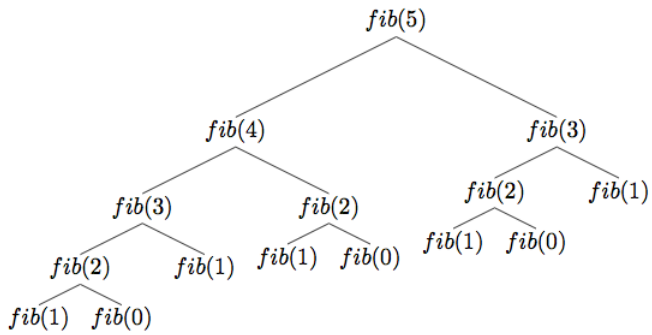
Podemos ver que la implementación es bastante simple, con únicamente un ciclo que va desde 1 hasta n.Además podemos ver en su gráfica que la relación de aunmento entre la entrada n y en numero de operaciones el lineal, es decir, numero de operaciones = entrada "n".Un recorrido lineal. Podemos concluir que su complejidad es $O(n)$

b) Fibo recursivo

```java
       * @throws IndexOutOfBoundsException Si el valor de <code>n</code>es
       */
      @Override
      public int fibonacciRec(int n){
        contador = 1;
        if(n < 0 ) throw new IndexOutOfBoundsException();
        //if(n == 0) return 0;
        //return fibo_aux(n);
        return n == 0 ? 0 : fibo_aux(n);
      }
      private int fibo_aux(int n){
      contador++;
      if(n <= 1) return n;
      else return fibo_aux(n - 1) + fibo_aux(n - 2); //Si n = 2, entonces l
    }
```

Podemos analizar este metódo de 3 maneras:

- Gráfica: Podemos observar en la gráfica de fibonacci recursivo que, la relación entre la entrada n y el número de operaciones crece de forma exponencial.
- Llamadas de metódo: Veamos que sucede cuando ejecutamos el metódo:



  Podemos ver que por cada vez que llamamos al metódo, este a su vez , tiene que llamar a otros dos, sucesivamente hasta que el metódo llegue al caso base.
  Entonces podriamos decir que está en orden $O(n^2)$

- Analisis de llamadas recursivas: Veamos que el metódo en general es de la forma:
  - fiboaux(n)
  - if(n <= 1) return n
  - else return fiboaux(n-1) + fiboaux(n-2)

  Veamos que :
  F(0)= 0 Solo hay una operación siempre.Es el caso base F(1)= 0 Solo hay una operación siempre. Es el caso base F(n)= F(n-1)+F(n-2) Veamos que F(n-2) <= F(n-1), podemos asumir que F(n-2)≈F(n-1) $\longrightarrow$
  $F(n) = F(n-1) + F(n-1)$
  $F(n) = 2F(n-1)$

  Como $F(n) = 2F(n-1)$ tenemos que:
  $F(n-1) = 2(F(n-2))$
  $F(n) = 2(2F(n-2)) = 4F(n-2)$

  Ahora veamos el patrón:
  $F(n-2) = 2F(n-3)$
  $F(n) = 2(2(2F(n-3))) = 8F(n-3)$
  $F(n) = 2^k F(n-k)$

  Como k<=n entonces el algoritmo está en $O(2^n)$

c) Pascal iterativo

```java
     * @throws IndexOutOfBoundsException si los indices <code>i</code> o
     * <code>j</code> son inválidos
     */
    @Override
    public int tPascalIt(int ren, int col){
      if( ren < 0 || col < 0) throw new IndexOutOfBoundsException();
      int [][] triangulo = new int[ren+1][col+1];
      contador=1;
      if(ren == col || col ==0 ) return 1;
      for(int i = 0;i <= ren;i++){
        triangulo[i] = new int  [i+1];
        triangulo[i][0] = 1; // j= 0
        triangulo[i][i] = 1; // j=i
        if(i >= 2){
          for (int j=1;j < i;j++){
            contador++;
            triangulo[i][j] = triangulo[i - 1][j - 1] + triangulo[ i-1 ][j];
          }
        }
      }
      //System.out.println(triangulo[ren ][col]);
      return triangulo[ren ][col ];
    }
```

Podemos ver que este metódo tiene dos ciclos anidados.
Analizando estos ciclos, veamos que, la iteración del ciclo exterior va desde 0 hasta, el parametro ingresado ren.Si nos encontramos En una posición triangulo[i][0] ó triangulo[i][i] entonces basta con regresar uno. Básicamente esto es seguir la norma donde si j=0 o j=i regresa 1.Entonces si se cumple esta condición, el numero de operaciones es el recorrido de este ciclo(lineal).Si no caemos en ese caso, entonces necesitamos hacer el recorrido de las columnas del triángulo, esto con un segundo ciclo que va desde j=1 hasta j<i.

Veamos que el ciclo interno tiene j iteraciones (desde que j =1 hasta que llega a i),por lo que el tiempo de ejecucion del ciclo interno es en general i.
El ciclo externo tiene k iteraciones,con k = ren, su cuerpo es el ciclo iterno , pero por cada valor de i = 0 hasta k el tiempo de ejecucion es 3j , osea cada que i itere , j tendrá i iteraciones.Podemos visualizar el tiempo de ejecución de la siguiente manera:

$$\sum_{i=}^{k} i = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} = \frac{3k^2+3k}{2} = \frac{3k^2}{2} + \frac{3k}{2} \tag{1}$$

EL tiempo de ejecución total es:

$$f(k) = 1 + \frac{3k^2}{2} + \frac{3k}{2} \tag{2}$$

Con K = ren.
Por lo que la complejidad es de orden $O(n^2)$

Observación: Esto sucede únicamente en el peor de los casos.Pues,como podemos observar en la gráfica, en los mejores casos podemos ver que la complejidad es de otro orden, bajando drasticamente.

Podemos ver que su complejidad aumenta, dependiendo que tan cerca del centro del triangulo estemos.pues si estamos en los extremos la complejidad es lineal, sin embargo, cuando tenemos la necesidad de recorrer hasta el punto medio de ls columnas la complejidad es $O(n2)$

d) Pascal recursivo

```java
    @Override
    public int tPascalRec(int ren, int col){
      if( ren < 0 || col < 0) throw new IndexOutOfBoundsException();
      contador = 1;
      return pascalAux(ren, col);
    }
    private int pascalAux(int ren, int col){
      contador ++;
      if( ren == col || col == 0) return  1; //Esta linea de codigo no
      return pascalAux(ren - 1,col - 1) + pascalAux(ren - 1,col);
    }

    /**
```

Primero observemos en la gráfica, pascal recursivo, la forma en que crece, en este algoritmo el peor de los casos es cuando llega al pico de la gráfica es decir, mientras mas centrados nos encontremos, respecto al triángulo. Al igual que ocurre con fibo recursivo, aqui Cada que se ejecuta a Pascal(), se llama a Pascal(i-1,j-1) + Pascal(i-1)(j), es decir, se generan otras dos llamadas para ejecutar y asi será hasta que cada parte de las llamadas llegue al caso base. En el peor de los casos, las llamas del "lado izquierdo y derecho"tiene la misma altura y misma cantidad de llamdas por lo que, su complejidad es $O(n^2)$

Observaciones y consideraciones

Para realizar las gráficas ejecuté mi programa con entradas más o menos grandes, pero en el código que subiré al repositorio, las dejaré en cantidades más pequeñas, de cualquier forma, los datos que obtuve de las ejecuciones grandes vienen anexados en la carpeta reporte, por si quisieran revisar las gráficas.