Gráficos y tablas de rendimiento comparativo

Compare el tiempo de ejecución de todos los métodos implementados en los puntos 2.1 y 2.2 Genere todos los casos que considere necesarios para realizar el análisis.

Tiempo de Ejecución						
Polinomios	n=1	n=10	n=100	n=1000	n=10000	n=100000
evaluarMSucesivas()	0 ms	0 ms	0 ms	5 ms	217 ms	7233 ms
evaluarRecursiva()	0 ms	0 ms	1 ms	2 ms	150 ms	StackOverflowError
evaluarRecursivaPar()	0 ms	0 ms	0.7 ms	1 ms	3 ms	20 ms
evaluarProgDinamica()	0 ms	0 ms	0 ms	2ms	155 ms	StackOverflowError
evaluarMejorada()	0 ms	0 ms	0 ms	1ms	1ms	4ms
evaluarPow()	0 ms	0 ms	0 ms	0 ms	3 ms	38 ms
evaluarHorner()	0 ms	0 ms	0 ms	0 ms	0 ms	1 ms

Tiempo de Ejecución (en ms)						
Binomio de Newton	n=1	n=10	n=100	n=1000	n=10000	n=100000
potenciasJuntas()	0	0	1	1	18	OutOfMemory Error: Java heap space
cargarMatrizCoefPascal()	0	0	1	7	87	OutOfMemory Error: Java heap space
cargarMatrizCoeficientes()	0	0	4	810	847046	OutOfMemory Error: Java heap space
coeficienteTerminoK()	0	0	0	0	0	OutOfMemory Error: Java heap space
coefTerminoK()	0	0	0	0	0	OutOfMemory

						Error: Java heap space
coefKDesdeMatriz()	0	0	0	0	0	OutOfMemory Error: Java heap space
coeficientes()	0	0	0	8	257	OutOfMemory Error: Java heap space
coeficientesDesdeArray()	0	0	0	13	543	OutOfMemory Error: Java heap space
coeficientesRec()	0	0	0	6	Stack Overflow	OutOfMemory Error: Java heap space
resolverBinomio()	0	0	0	0	6	OutOfMemory Error: Java heap space
resolverBinomioRec()	0	0	0	0	Stack Overflow	OutOfMemory Error: Java heap space

Para los tiempos creo que era:

```
Long tInicial= System.currentTimeinMillis();
        //aca va la función
Long tFinal=System.currentTimeinMillis();

Long t = tFinal-tInicial;
System.out.println("t");
```

Análisis de complejidad computacional
Indique la función de complejidad computacional asociada a cada uno de los métodos implementados.

Conclusiones
A partir de los análisis comparativos extraiga conclusiones.

Polinomios	Complejidad Computacional	Explicación	Conclusión
evaluarMSucesivas()	O(n^2)	Esta función recorre el vector de coeficientes, calculando la potencia por medio de multiplicaciones sucesivas	Es óptimo en cuanto al uso de memoria pero al tener un n muy grande el tiempo de ejecución es demasiado.
evaluarRecursiva()	O(n^2)	Recorre el vector de coeficientes (n) y calcula para cada uno la potencia mediante una función recursiva.	No es eficiente ya que utiliza mucha memoria a causa de tener una función recursiva para el cálculo de potencia.
evaluarRecursivaPar()	O($n * log(n)$)	Recorre el vector de coeficientes, y para cada uno calcula su potencia teniendo en cuenta si es par o impar.	Mejor que la anterior ,al utilizar el cálculo de potencias pares, pero sigue sin ser una solución eficiente, a causa del uso excesivo de memoria.
evaluarProgDinamica()	O(n^2)	Recorre el vector de coeficientes, busca la potencia en un vector ya cargado con potencias, en caso de no encontrarlo lo calcula por medio de multiplicaciones sucesivas	Sera mas eficiente que las opciones anteriores pero solo en casos donde se acceda muchas veces a la misma potencia, ya que en esos casos la complejidad se aproxima a O(n)
evaluarMejorada()	O(n)	Utiliza el mismo metodo de programacion dinamica que	Mucho mas eficiente que las funciones

		la funcion anterior, pero este llena todo el vector antes de meterse en el ciclo de resolucion del polinomio, Entonces al momento de resolver el polinomio solo tiene que buscar el valor de la potencia en un vector, lo cual tiene complejidad O(1). Por regla de la suma entonces O(n) + O(n*1) = O(n)	anteriores en cuantoa uso del procesador, pero requiere un vector auxiliar que en algunos casos podria resultar exigente para la memoria
evaluarPow()	o(n log(b))	Como Math.pow() tiene una complejidad computacional de O(log(b)) (siendo b la potencia), y evaluarPow evalua cada uno de los terminos del polinomio, por regla del producto esa sera la complejidad computacional resultante	...
evaluarHorner()	o(n)	Siendo n la cantidad de coeficientes, el metodo de horner solo necesita ejecutar una instruccion por cada grado del polinomio	Es la solucion más eficiente

Binomio de Newton	Complejidad Computacional	Explicación	Conclusión
Aca van los métodos		Explicación de la complejidad computacional calculada	
potenciasJuntas()	O(n.log(n))	Esta función recorre un array n veces, calcula y almacena en el mismo la potencia de los términos del binomio (a y b).	Es una forma útil de disponer de esta información al momento de calcular del desarrollo del binomio.
cargarMatrizCoefPascal()	O(n^2) / n^2	La función recorre una matriz de n+1 x n+1, y va	Es una de las formas más óptimas de

		generando los valores correspondientes a los coeficientes para un n puntual. Usa la propiedad de construcción del triángulo de Pascal, la cual implica que el coeficiente para un [i][j] es la suma de los términos que se encuentran sobre él, en la fila anterior ([i-1][j-1] y [i-1][j] respectivamente).	obtener los coeficientes, debido a la propiedad del triángulo. El cálculo más complejo que realiza es una suma de valores que ya tiene calculados (sólo tiene que pedirlos a la misma matriz), y la multiplicación correspondiente del término con sus potencias.
cargarMatrizCoeficientes()	O(n^3)	La función recorre una matriz de n+1 x n+1, y genera los coeficientes del binomio mediante el cálculo de combinatoria.	No es muy óptimo al momento de contar con un n demasiado grande, ya que realiza los cálculos de combinatoria por cada n.
coeficientesDesdeArray()	O(n^2)	La función utiliza dos arrays de tamaño n+1. Va creando el triángulo por propiedad: en un array almacena la primer fila, y en el segundo va creando la siguiente basándose en los resultados de la fila anterior.	Calcula todo el triángulo, pero sólo almacena en el array la fila actual en la que está trabajando. Sólo utiliza un array auxiliar, y uno para los coeficientes, ambos con tamaño n+1. Ocupa menos memoria que la matriz de n+1 x n+1. Trabaja “sin memoria”, ya que le basta con tener la fila anterior de la matriz para generar la siguiente.
coeficientes()	O(n^2)	Recorre un array de tamaño n+1 y calcula los coeficientes mediante combinatoria.	Es óptimo, ya que usa sólo un array de tamaño n+1, a pesar de que calcule las combinatorias.

coeficientesRec()	O(n)	Calcula y almacena los coeficientes en un array mediante combinatoria de manera recursiva.	Es eficiente en cuanto a la cantidad de cálculos que realiza (sólo la combinatoria), pero no lo es para recursividad, mucho menos cuando se trata de un n muy elevado.
coefKDesdeMatriz()	O(1)	Retorna el coeficiente desde la matriz, dado un i, j específicos.	Es muy eficiente, ya que sólo pide, mediante índices, por un valor en la matriz de coeficientes que ya está cargada.
coefTerminoK()	O(1)	Retorna el coeficiente desde el array, dado un i específico.	Es muy eficiente, ya que sólo pide, mediante un índice, por un valor en el array de coeficientes que ya está cargado.
coeficienteTerminoK( )	O(n.log(n)) // O(n)	Retorna el cálculo del coeficiente especificado: calcula combinatoria y potencias de los términos del binomio.	Es bastante eficiente, ya que sólo calcula el valor del coeficiente en el momento y lo retorna, sin necesidad de ocupar memoria. Pero si se quiere volver a usar su resultado, tendría que volver a calcularlo.
resolverBinomio()	O(n.log(n))	Retorna el valor del desarrollo del binomio, evaluado en un valor dado de x.	Es eficiente, ya que sólo recorre un array de n+1 posiciones, y multiplica cada coeficiente del array por la potencia de x que corresponda. Al ser (ax+b)^n el binomio, se sabe que x está en el término

			de a, y por propiedad, el término de a es elevado desde n a 0, a medida que se desarrolla el binomio.
resolverBinomioRec()	O(1)	Retorna el valor del desarrollo del binomio, evaluado en un valor de x dado, de manera recursiva.	Es efectivo en cuanto a los cálculos que realiza: recorre un array y acumula los cálculos para luego retornar el valor total. Al ser recursiva, deja de funcionar para valores de n muy grandes.