Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación

Fundamentos de Análisis y Diseño de Algoritmos
Jesus Alexander Aranda Bueno

Proyecto Final

Nombre	Código	Plan
Daniel Eduardo Gaviria Mazuera	1325535	2711
Daniel Felipe García Cárdenas	1430461	3743
Jorge Luis Gonzalez Pineda	1229411	2711
José Alexander Muñoz Delgado	1367014	2711

Introducción.

La rama de la computación en la Ingeniería es tal vez una de las más complejas, por la capacidad que se debe desarrollar para dar solución a problemas que en algunos casos pueden ser más complejos de lo esperado. Para una persona que no hace parte de este disciplina, y para quienes sí, tan pronto como iniciamos, se nos queda corta la idea de ¿A qué nos vamos a enfrentar?.

El análisis de un problema computacional, es tal vez una de las tareas más complejas a las cuales nos podemos enfrentar, ya que cualquier persona puede programar siguiendo pasos y viendo tutoriales. Y observamos que en numerosas ocasiones muchos de los Ingenieros en el mercado dan solución a los problemas, y no tienen en cuenta la forma o el camino para llegar a dicha solución, es decir desconocen la solución más óptima. Entonces, es cuando te das cuenta que en estos casos la solución que se le da a los problemas computacionales es tal vez la más **ingenua.**

En el este trabajo se plantea un problema al que un ingeniero podría enfrentarse, problema al que muchos podrían encontrar su forma **ingenua** de resolver. Sin embargo, abordaremos el problema empleando dos técnicas de programación llamadas **Programación Dinámica** y **Programación Voraz.**

PROBLEMA: PLANIFICACIÓN FORESTAL

Cierta región dedicada a la producción forestal se encuentra organizada en n zonas llamadas rodales. Cada rodal tiene un número estimado de árboles que se pueden talar, un costo estimado del proceso de tala, una fecha de inicio y una fecha final para talar, además de un número de personas que se requerirían en la tala de ese rodal.

Para realizar el proceso de tala de dichos rodales se requiere cierta máquina, lamentablemente sólo hay una máquina de estas a disposición y se le ha encargado a usted la labor de determinar la planificación de cuáles rodales van a ser talados por medio de dicha máquina.

Importante: No se pueden seleccionar rodales que se solapen en las fechas para talar, es decir las fechas comprendidas entre la fecha de inicio y final de un rodal no se puede solapar con la de ningún otro rodal que vaya a ser seleccionado. Esto aplica para todas las variantes del problema.

Teniendo en cuenta el problema planteado y que en concreto se nos ha solicitado diseñar algoritmos que resuelvan los siguientes problemas

1) Determinar cuáles rodales deben ser talados buscando maximizar el número de árboles talados.
2) Determinar cuáles rodales deben ser talados considerando maximizar el número de rodales seleccionados.
3) Determinar cuáles rodales deben ser talados buscando maximizar el número de árboles talados mientras el costo total estimado no supere cierto umbral.
4) Determinar cuáles rodales deben ser talados considerando maximizar el número de árboles talados mientras el número total de árboles talados no supere cierto umbral.
5) Determinar cuáles rodales deben ser talados considerando maximizar el número de árboles talados mientras el número total de árboles talados no supere cierto umbral y mientras el número total de personas empleadas supere cierto umbral.

Hemos implementado la solución del proyecto en JAVA con el IDE NetBeans, está dividido en 1 clase por cada punto solucionado, 1 clase para manejo de archivos, 1 para el rodal, 1 para mergesort, y 1 para el GUI la vista, que és un menú interactivo.

HARDWARE/SOFTWARE UTILIZADO

Procesador: Intel Core i7, cuarta generación.
Ram: 8 Gb.
IDE: Netbeans, JDK 1.8.
Sistema Operativo: Windows 7, 64 bit's.

SOLUCIÓN PUNTO 1

Para la solución del primer punto en el cual se nos pide determinar cuáles rodales deben ser talados buscando maximizar el número de árboles talados, teniendo en cuenta las restricciones generales del problema. hemos usado un algoritmo dinámico mediante el cual primeramente se identifica si un rodal es válido, es decir que no se solapa, de ser así, se suma la cantidad de árboles de ese rodal y se verifica si esa suma de la cantidad de árboles es mayor que la cantidad de árboles de la última suma de árboles realizada, y si es verdad se toma este valor como última suma. De lo contrario se mantiene el valor anterior como última suma. Maximizando de esta manera el número de árboles a talar.

Método	Objetivo
mergesort	
`public int siguienteRodalValido(ArrayList<Rodal> rodales, int posicionActal)`	Identificar si un rodal se solapa con otros (mediante la verificación de las fechas). y retorna la posición del anterior rodal más próximo con el que no se solapa.
`public void buscarRodales(ArrayList<Rodal> rodales, int n)`	Llenar tabla con el mayor número de árboles a talar sin que se solapen sus rodales. maximizar el número de árboles a talar.

Posteriormente mediante la resta de los valores en la tabla con los valores de la cantidad de árboles en cada rodal, se identifica los rodales con los cuales se maximiza el número de árboles a talar.

Método	Objetivo
`public void RodalesSeleccionados()`	Listar un arraylist que contiene los rodales seleccionados para talar.
`public int findMayorMenorPosicion(int actual, int posicionActual)`	Identificar la posición del anterior rodal seleccionado. Retornando el último rodal seleccionado.

ANÁLISIS DE COMPLEJIDAD

Método	Complejidad
mergesort	**O(n log n)**
`public int siguienteRodalValido(ArrayList<Rodal> rodales, int posicionActal) { int posicion = -1; for (int i = posicionActal; i >= 0; i--) { if (rodales.get(i).getFechaFin()<= rodales.get(posicionActal).getFechaInicio()) { posicion = i; return posicion; } } return posicion; }`	un condicional if, de complejidad constante, el cual está dentro de un ciclo For de complejidad n. siendo la Complejidad **O(n)**
`public void buscarRodales(ArrayList<Rodal> rodales, int n) { tabla[0] = rodales.get(0).getNumerArboles(); for (int i = 1; i < n; i++) { int numeroArbolesInicial = rodales.get(i).getNumerArboles(); int siguienteRodal = siguienteRodalValido(rodales, i); if (siguienteRodal != -1) { numeroArbolesInicial += tabla[siguienteRodal]; } if (numeroArbolesInicial > tabla[i - 1]) { tabla[i] = numeroArbolesInicial; } else { tabla[i] = tabla[i - 1]; } } }`	Un ciclo **For**, dentro del cual hay un llamado a la función siguienteRodalValido() la cual tiene una complejidad O(n), más dos if de complejidad constante. Para una complejidad final de n + n + 1 +1 = **O(n)**
`public int findMayorMenorPosicion(int actual, int posicionActual) { int pos = posicionActual; for (int i = posicionActual; i >= 0; i--) { if (i == 0 && tabla[i] == actual) { pos = 0; return pos; } else { if (tabla[i] < actual) { pos = i + 1; return pos; } } } return pos; }`	dos condicional **If**, de complejidad constante, los cual están dentro de un ciclo For de complejidad n. siendo la Complejidad **O(n)**

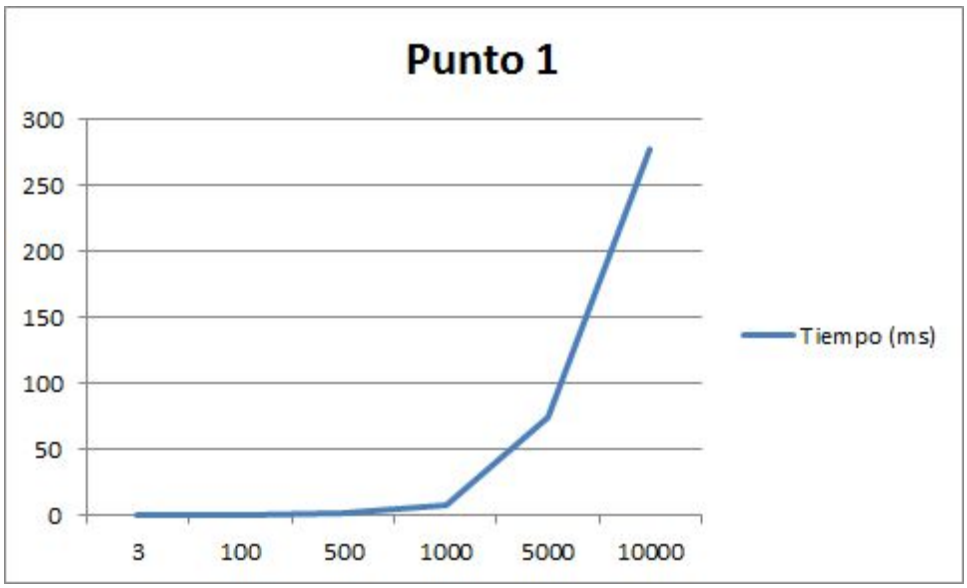
<pre>public void RodalesSeleccionados() {
    int i = tabla.length - 1;
    int actualAcomparar = tabla[i];
    int rodal = 0;
    rodalesSeleccionados = new ArrayList<Rodal>();
    while (i >= 0 && actualAcomparar > 0) {
        rodal = findMayorMenorPosicion(actualAcomparar, i);
        rodalesSeleccionados.add(rodalesAevaluar.get(rodal));
        numeroArboles += rodalesAevaluar.get(rodal).getNumerArboles();
        actualAcomparar = actualAcomparar - rodalesAevaluar.get(rodal).getNumerArboles();
        i = rodal - 1;
 	
    }
}</pre>
	While de complejidad **n**, el cual hace en su parte interna un llamado a la función findMayorMenorPosición () que tiene complejidad n. para una complejidad final de n + n = **O(n)**

En conclusión la solución del punto 1 es de complejidad **O(n log n)**.	

Pruebas realizadas

Cantidad Entradas	3	100	500	1000	5000	10000
Tiempo(MS)	0	0	2	7	74	277

Análisis de resultados

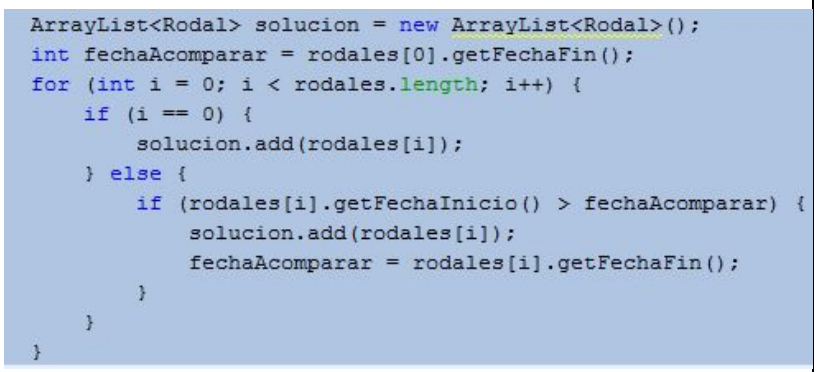


La tabla anterior nos muestra que se cumple la complejidad a cabalidad para cada una de las entradas, de acuerdo a la cantidad de las mismas. La complejidad **nlogn,** nos da un crecimiento no tan significativo entre las diferentes entradas. Sin embargo vemos la diferencia entre cada una de ellas.

SOLUCIÓN PUNTO 2

Determinar cuáles rodales deben ser talados considerando maximizar el número de rodales seleccionados.

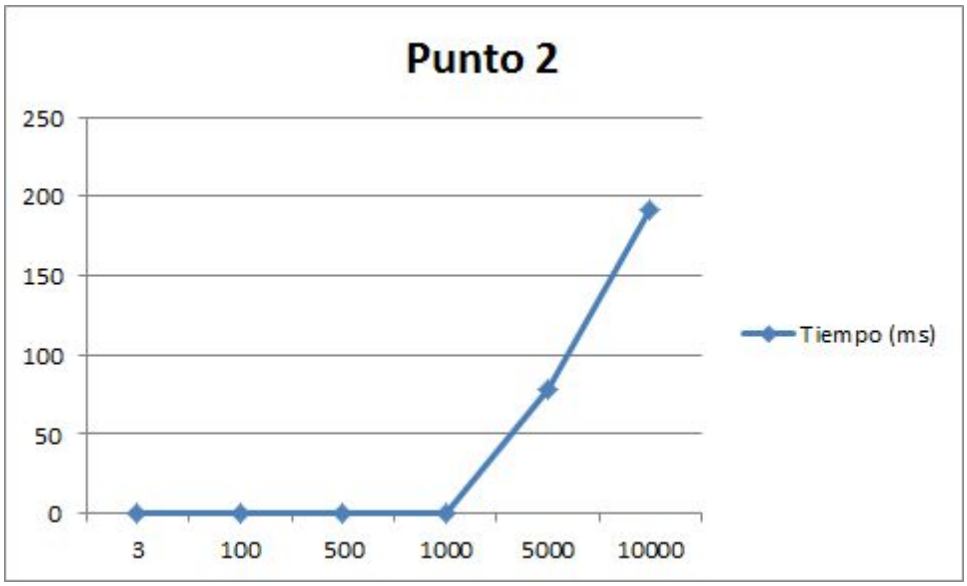
Se utilizó la técnica de Programación Voraz para dar solución a este problema, comenzaremos por ordenar los rodales de acuerdo a su fecha de finalización en orden ascendente por medio del algoritmo de ordenamiento MergeSort, es decir, el primer rodal ordenado será el rodal que más rápido finaliza su proceso de tala. Después, se asume que el primer rodal será talado como principio de la programación voraz, a partir de allí, se toma cada uno de los rodales que no se solapan entre ellos. Teniendo en cuenta que la técnica de programación voraz, no siempre encuentra la solución más óptima, habremos encontrado una solución que es muy próxima a ella.

Método	**Complejidad**
mergesort	**O(n log n)**
	Un ciclo **For** y dentro de este un condicional **If** dentro de otro If, para una complejidad constante, más la complejidad n del ciclo For. Finalmente una complejidad de 1 + n = **O(n)**.
En conclusión la solución del punto 2 es de complejidad **O(n log n)**.	

Pruebas realizadas

Cantidad Entradas	3	100	500	1000	5000	10000
Tiempo (ms)	0	0	0	0	78	192

Análisis de resultados



La programación voraz implementada en el segundo punto, el cual nos dá como complejidad **n,** se ve reflejada en el crecimiento lineal que está directamente relacionados con la cantidad de entradas.

SOLUCIÓN PUNTO 3

Para la solución del tercer punto en el cual se nos pide determinar cuáles rodales deben ser talados buscando maximizar el número de árboles talados mientras el costo total estimado no supere cierto umbral., teniendo en cuenta las restricciones generales del problema. hemos usado el algoritmo mergesort para ordenar ascendentemente por fecha de finalización los rodales, con un algoritmo dinámico el cual calcula el mayor número de árboles que se puede obtener de un conjunto de rodales que no se solapen entre sí, a medida que se va generando ese conjunto se valida que la suma del costo de cada rodal no supere el umbral del costo establecido. este proceso se ejecuta varias veces para obtener el mayor beneficio y almacenando en un arreglo los id de los rodales seleccionados.

Método	Complejidad
mergesort	**O(n log n)**

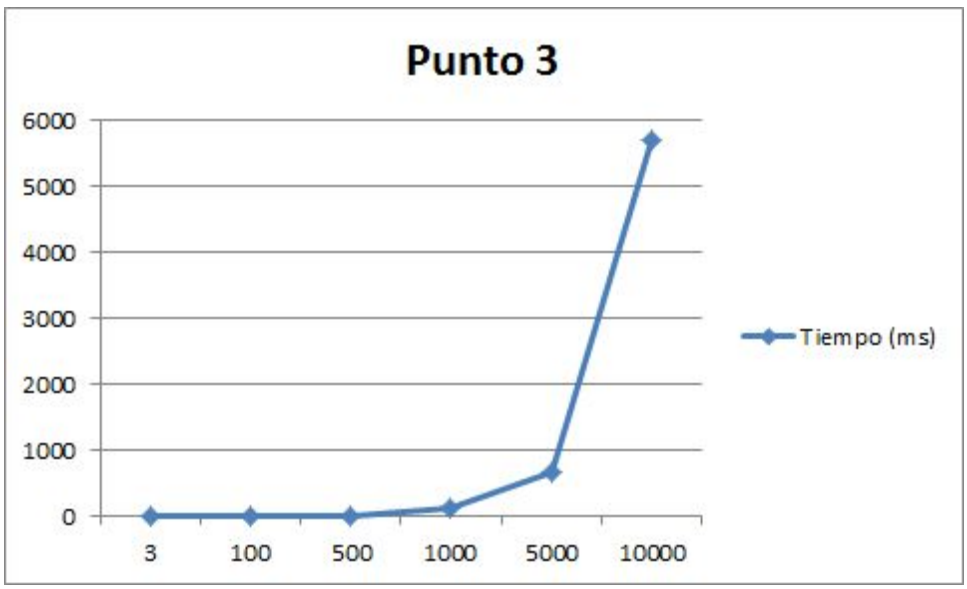
`for (int i = 0; i < rodalesEntrada.length; i++) {`
`  Arreglo_Soluciones_Temporales.clear();`
`  CalcularRodalesUp(i); //Calculo es CostoAlmac`
`  CalcularRodalesDown(i);`
`  Comparar();`
`}`
	Ciclo **For** dentro del cual se ejecuta el método **CaulcularRodalesUp()** y **CalcularRodalesDown()** de complejidad **n**, y el método **Comparar()** de complejidad **constante**, para una complejidad final (n+1)*n = **O(n^2)**

El ciclo mostrado es la parte del código que representa mayor complejidad en la solución del punto 3, con complejidad **O(n^2)**.	

Pruebas realizadas

Cantidad Entradas	3	100	500	1000	5000	10000
Tiempo (ms)	0	0	16	109	671	5700

Análisis de resultados



La complejidad **n^2**, implementada durante la solución del punto 3, es bastante compleja, es claro que una complejidad exponencial no es recomendada para dar solución a un ejercicio. Sin embargo para resolver este problema utilizando la técnica de Programación Dinámica, podemos ver en la gráfica anterior como se incrementa exageradamente el tiempo de acuerdo a la cantidad de entradas.

SOLUCIÓN PUNTO 4

Para la solución del cuarto punto en el cual se pide determinar cuáles rodales deben ser talados considerando maximizar el número de árboles talados mientras el número total de árboles talados no supere cierto umbral, teniendo en cuenta las restricciones generales del problema. hemos usado el algoritmo mergesort para ordenar ascendentemente por fecha de finalización los rodales, con un algoritmo dinámico el cual calcula el mayor número de árboles que se puede obtener de un conjunto de rodales que no se solapen entre sí, a medida que se va generando ese conjunto se valida que la suma de las cantidades de árboles a talar de cada rodal no supere el umbral de árboles establecido. Este proceso se ejecuta varias veces para obtener el mayor beneficio, es decir maximizando el número de árboles talados sin superar un número determinado de árboles talados. Almacenando en un arreglo los id de los rodales seleccionados .

Método	**Complejidad**
mergesort	**O(n log n)**

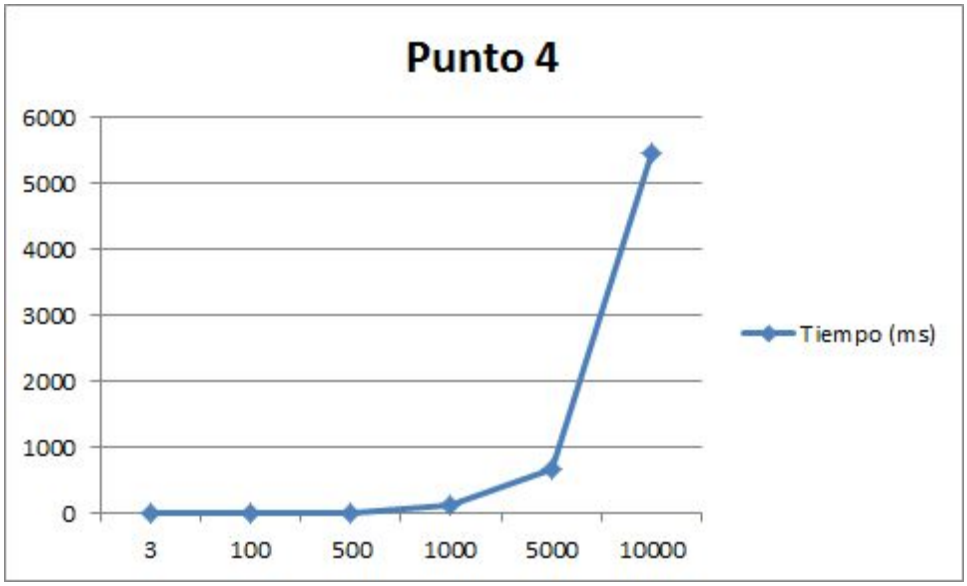
`for (int i = 0; i < rodalesEntrada.length; i++) {`
`  Arreglo_Soluciones_Temporales.clear();`
`  CalcularRodalesUp(i); //Calculo es CostoAlmac`
`  CalcularRodalesDown(i);`
`  Comparar();`
`}`
	Ciclo **For** dentro del cual se ejecuta el método **CaulcularRodalesUp()** y **CalcularRodalesDown()** de complejidad **n**, y el método **Comparar()** de complejidad **constante**, para una complejidad final (n+1)*n = **O(n^2)**

El ciclo mostrado es la parte del código que representa mayor complejidad en la solución del punto 4, con complejidad **O(n^2)**.	

Pruebas realizadas

Cantidad Entradas	3	100	500	1000	5000	10000
Tiempo (ms)	0	16	16	110	666	5458

Análisis de resultados



Vemos la misma relación existente entre la cantidad de entradas y el tiempo, para los puntos 3 y 4 de acuerdo su complejidad, ratificamos el gran costo computacional que le lleva a un equipo ejecutar un algoritmo de complejidad exponencial, sin importar el hardware en el cual sea ejecutado.

SOLUCIÓN PUNTO 5

Para la solución del quinto punto en el cual se nos pide determinar cuáles rodales deben ser talados considerando maximizar el número de árboles talados mientras el número total de árboles talados no supere cierto umbral y mientras el número total de personas empleadas supere cierto umbral, teniendo en cuenta las restricciones generales del problema. hemos usado el algoritmo mergesort para ordenar ascendentemente por fecha de finalización los rodales, con un algoritmo dinámico el cual calcula el mayor número de árboles que se puede obtener de un conjunto de rodales que no se solapen entre sí, a medida que se va generando ese conjunto se valida que la suma de las cantidades de árboles a talar de cada rodal no supere el umbral de árboles establecido y que la suma de la cantidad de empleados no sea inferior al umbral de empleados. Este proceso se ejecuta varias veces para obtener el mayor beneficio, es decir maximizando el número de árboles talados sin superar un número determinado de árboles a talar y con un mínimo de empleados. Y almacenando en un arreglo los id de los rodales seleccionados

Método	Complejidad
mergesort	**O(n log n)**

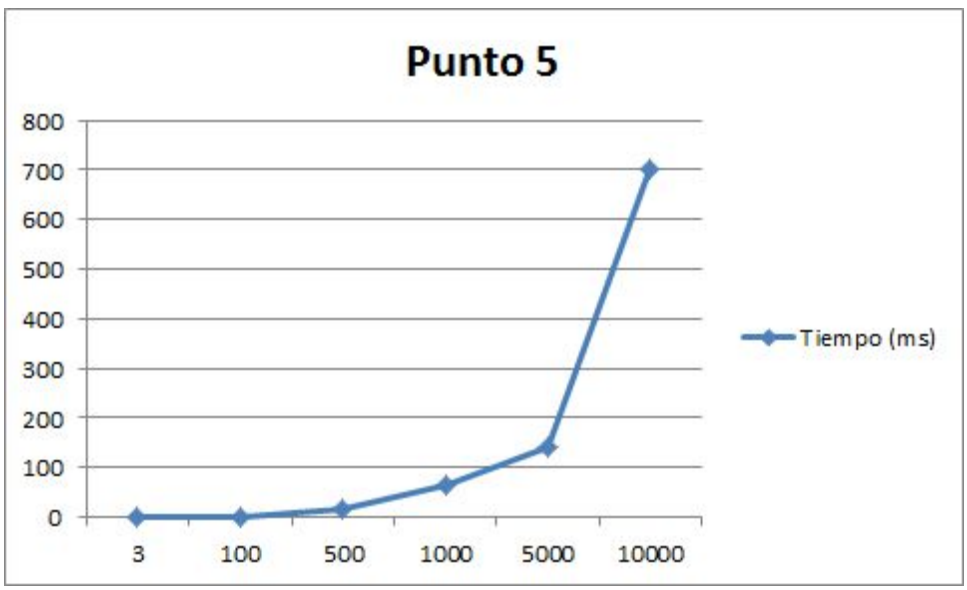
`for (int i = 0; i < rodalesAevaluar.length; i++) {`
`  Solucion.clear();`
`  CalcularRodalesUp(i);`
`  CalcularRodalesDown(i);`
`  Comparar();`
`}`
	Ciclo **For** dentro del cual se ejecuta el método **CalcularRodalesUp()** y **CalcularRodalesDown()** de complejidad **n**, y el método **Comparar()** de complejidad **constante**, para una complejidad final (n+1)*n = **O(n^2)**

El ciclo mostrado es la parte del código que representa mayor complejidad en la solución del punto 5, con complejidad **O(n^2)**.	

Pruebas realizadas

Cantidad Entradas	3	100	500	1000	5000	10000
Tiempo (ms)	0	0	16	63	141	703

Análisis de resultados



Del punto cinco podemos obtener las mismas conclusiones del punto 3 y 4, ya que se cumple la misma complejidad.

Conclusiones Generales

Para finalizar, después de la construcción de los algoritmos que dan solución a este problema, teniendo en cuenta la complejidad con la que fueron resueltos y su tiempo de ejecución, podemos ver la importancia que tiene dar solución a un problema de la manera más óptima posible, esto ahorra tiempo y recursos del ordenador.

A continuación vemos una tabla donde se visualiza sobre el eje X la cantidad de entradas, sobre el eje Y el tiempo de ejecución en cada una de ellas. es clara la diferencia de tiempo de ejecución en las diferentes complejidades, explicando de manera grafica lo dicho anteriormente.