UNED

Memoria Práctica EPED 15/16

MISRRAIM SUÁREZ PÉREZ - misrraimsp@gmail.com

1 Escenario simplificado

*1.1 La clase **AcademiaS** implementa la interfaz **AcademiaIF** que, a su vez, extiende la interfaz **CollectionIF<DoctorIF>**. De los Tipos Abstractos de Datos estudiados en la asignatura, ¿cuáles se pueden utilizar para representar las relaciones de dirección de tesis doctorales? ¿Cómo deberían utilizarse esos TAD?*

De los tipos de datos abstractos vistos en la asignatura se podrían utilizar tanto las listas como los árboles generales. La complejidad de las implementaciones de los métodos dependería de los campos con los que cuente la clase DoctorS.

Así, se podría implementar la academiaS como una lista de doctores, en la que cada doctor contenga una referencia a su supervisor y a una lista con sus estudiantes.

```
public class AcademiaS {

        private List<DoctorIF> academia;

        ...

}

public class DoctorS {

        private DoctorIF supervisor;

        private CollectionIF<DoctorIF> students;

        ...

}
```

También podría concebirse la academiaS como un árbol general, en cuyo caso la clase DoctorS no necesariamente debería tener el campo supervisor, y no necesitaría para nada el campo students. No obstante, el campo supervisor aceleraría notablemente las consultas sobre el árbol.

```
public class AcademiaS {

        private Tree<DoctorIF> academia;

        ...

}

public class DoctorS {

        private DoctorIF supervisor;

        ...

}
```

1.2 Entre los TAD considerados en la pregunta anterior, escoja razonadamente uno y explique cómo funcionaría cada uno de los métodos detallados en la interfaz ***DoctorIF*** *y el método* ***getSupervisor()*** *de la clase* ***DoctorS****. ¿Cuál sería su coste asintótico temporal en el caso peor?*

El TAD escogido para representar la clase academiaS ha sido una lista de doctores, dotando a la clase doctorS de un campo *doctorIF supervisor* y un campo *CollectionIF<DoctorIF> students*, tal y como se explica en la primera opción presentada en la pregunta anterior.

De esta manera los métodos de la interfaz se implementarían como:

- getSupervisor();

  Sería inmediato a través de un getter. Su coste asintótico temporal en el caso peor sería constante.

- getAncestors(**int** generations);

  Se resuelve utilizando recursión. La salida es la unión de conjuntos entre el conjunto unitario formado por el supervisor del doctor llamante y el conjunto devuelto por una llamada recursiva con una generación menos.

  En cuanto al coste asintótico temporal en el caso peor, se utiliza las expresiones dadas por el ED para algoritmos recursivos, en este caso para sustracción y una sola llamada recursiva. El tamaño del problema es el número de generaciones. El coste viene determinado por el método *union* de la clase *Set*. El coste de este método es del orden del producto de los tamaños de los conjuntos llamante y parámetro. Como el conjunto llamante siempre tendrá tamaño unidad, el coste de este método queda lineal. Entonces, aplicando la expresión comentada se obtiene un coste cuadrático $O(n^2)$ con el número de generaciones.

- getStudents();

  Sería inmediato a través de un getter. Su coste asintótico temporal en el caso peor sería constante.

- getDescendants(**int** generations);

  Se resuelve utilizando recursión. La salida es la unión de conjuntos entre el conjunto formado por los estudiantes del doctor llamante y el conjunto devuelto por una llamada recursiva, con una generación menos, por cada estudiante.

  En cuanto al coste asintótico temporal en el caso peor, se utiliza las expresiones dadas por el ED para algoritmos recursivos, en este caso para sustracción y llamadas recursivas múltiples.

  $$O\left(K^{G\div 1}\cdot K^2\right)=O\left(K^{G+2}\right)$$

  El tamaño del problema es el número de generaciones (G) y el número de estudiantes por doctor (K). Aplicando la expresión comentada se obtiene el coste anterior, ya que el número de llamadas recursivas será K, y el coste de las operaciones no recursivas, que viene determinado por el método *union*, es cuadrático con K. Esto es, el algoritmo tiene complejidad exponencial respecto al número de generaciones G, y complejidad potencial respecto al número de hijos K.

- getSiblings();

  Realiza la diferencia de conjuntos entre los estudiantes del supervisor del doctor llamante y el propio doctor llamante.

  El coste asintótico temporal en el caso peor de este método viene determinado por el coste del método *difference* de la clase *Set*. Este método tiene un coste cuadrático resultado de multiplicar los tamaños de los conjuntos parámetro y llamante, ya que en el caso peor por cada elemento del conjunto llamante se recorrerá todo el conjunto parámetro. No obstante, en el método *getSiblings* el conjunto parámetro solo tendrá un elemento (el doctor llamante), luego el coste será lineal. En definitiva, el coste es $O(n)$.

*1.3 Ateniéndonos a este escenario, para cada uno de los cuatro métodos descritos en la interfaz **DoctorIF**, ¿pueden aparecer repeticiones? Justifique su respuesta.*

Dada la descripción del escenario NO podrían aparecer repeticiones. Cada doctor posee únicamente un supervisor.

2 Escenario completo

2.1 Explique cómo cambian las relaciones de supervisión en este segundo escenario. A la hora de representar el problema, ¿puede aplicarse directamente el mismo TAD utilizado en el primer escenario? ¿Por qué?

En este segundo escenario un mismo doctor puede tener varios supervisores, luego la opción planteada en la primera pregunta de implementar la Academia como un árbol general ya no es posible. Sin embargo, la implementación mediante lista, que fue la empleada en el escenario simplificado, es igualmente válida en este segundo contexto.

*2.2 Razone, sin realizar ningún tipo de implementación, cómo funcionarían en este escenario los métodos de la interfaz **DoctorIF** y el método **getSupervisors()** de la clase **DoctorC** suponiendo que la clase **AcademiaC** se implemente utilizando una lista de objetos de la clase **DoctorC**.*

La clase DoctorC tiene de partida tres atributos, a saber, el identificador del doctor, la academia a la que pertenece y una colección con los estudiantes a los que ha dirigido la tesis. Con estos atributos, e implementando AcademiaC como establece el enunciado, los métodos podrían ser resueltos de la siguiente manera:

getSupervisors(); Habría que recorrer dicha lista visitando cada DoctorC y almacenando aquellos en los que el doctor llamante se encuentra entre sus estudiantes.

getAncestors(**int** generations); Los ancestros de cada nivel del doctor llamante se obtendrían llamando a getSupervisors() sobre cada ancestro del nivel inferior. Después se tendría que recorrer la colección de salida para eliminar las repeticiones, ya que un doctor puede ser ancestro de otro en varios niveles. Las sucesivas llamadas a getSupervisors(), cada una con su recorrido de la lista, hace que el tiempo de ejecución sea exponencial con el número de generaciones que se busque.

getStudents(); Sería inmediato a través del getter correspondiente.

getDescendants(**int** generations); Los descendientes de cada nivel del doctor llamante se obtendrían llamando a getStudents() sobre cada descendiente del nivel superior. Después se tendría que recorrer la colección de salida para eliminar las repeticiones, ya que un doctor puede ser descendiente de otro en varios niveles.

getSiblings(); Primero se obtendría los supervisores del doctor llamante a través de getSupervisors(). Después se unirían los estudiantes de cada supervisor. Por último, se recorrería la colección de salida para eliminar al doctor llamante y las repeticiones, ya que diferentes supervisores pueden dirigir la tesis al mismo estudiante.

*2.3 Basándose en la respuesta a la pregunta anterior, supongamos que se añade a la clase **DoctorC** un nuevo atributo **supervisors** con la colección de los doctores que dirigieron su tesis doctoral (y, en consecuencia, el método **getSupervisors()** se convierte en un getter de dicho atributo). ¿Cómo afectaría esto al espacio necesario para almacenar el Árbol Genealógico de la Academia en memoria? ¿Y cómo afectaría a la programación y al tiempo de ejecución de los métodos?*

La memoria necesaria para almacenar el árbol genealógico de la academia sería mayor, pero no mucho mayor. Suponiendo una academia con N doctores, se necesitaría espacio para N referencias a las colecciones de supervisores, más el espacio para dichas colecciones. Cada colección necesitaría espacio para M referencias a doctores.

Sin embargo, la programación y el tiempo de ejecución se verían alterados de manera notable. El hecho de que getSupervisors() fuese inmediato a través de un getter evita tener que recorrer la lista para encontrar los supervisores de un doctor, así que el método getAncestors() ya no tendría que recorrer la lista tantísimas veces, si no que sería implementado como getDescendants().

En conclusión, la memoria no se vería gravemente afectada, mientras que la programación se simplifica mucho y el tiempo de ejecución se reduce también bastante.

*2.4 Supongamos que la lista de objetos de la clase **DoctorC** referida en la pregunta 2.2 se encuentra siempre ordenada según el atributo **id**. ¿Cómo se podría aprovechar en la implementación de los métodos para mejorar el coste temporal de las operaciones de consulta? ¿Cómo afectaría a la operación de modificación?*

Si se supone que los id se van asignando de forma creciente según nuevos doctores van entrando en la academia, entonces se tendría que los supervisores de un doctor necesariamente tendían id's menores. Esto se puede aprovechar ya que no se tendría que recorrer la lista completa sino sólo los nodos previos al doctor llamante de getSupervisors(). Así, siguiendo el mismo razonamiento, en las sucesivas llamadas a getSupervisors() desde getAntecesors() los recorridos serían cada vez menores, ya que los doctores llamantes son cada vez más antiguos.

En cuanto a la operación de modificación, añadir un nuevo doctor sería añadir un nodo al final de la lista comprobando que el id es válido. Para dicha comprobación basta consultar el ultimo nodo de la lista. A la hora de añadir una relación de supervisión, hay que comprobar que el codirector tenga mayor antigüedad que el estudiante

2.5 Supongamos que levantamos la restricción de que en una Academia sólo exista un Doctor sin supervisores dentro de la Academia y permitimos que haya varios Doctores en esa situación. ¿Habría que hacer cambios en la implementación? ¿Cuáles y por qué?

El fundador ya no sería único, y el método para encontrarlo ya no puede estar basado, como en la implementación que se ha entregado, en que es el de más antigüedad y por tanto el primero de la lista. Se tendría que comprobar todos los elementos de la lista y acumular aquellos que no tuvieran supervisor.

*2.6 Ateniéndonos a este escenario, para cada uno de los cuatro métodos descritos en la interfaz **DoctorIF** y para el método **getSupervisors()** de la clase **DoctorC**, ¿pueden aparecer repeticiones? Justifique su respuesta.*

getSupervisors(); **No** habría repeticiones. Sólo se recorre la lista una vez.

getAncestors(**int** generations); **Sí** puede haber repeticiones. Un doctor puede ser ancestro de otro en varios niveles.

getStudents(); **No** habría repeticiones. Sería inmediato a través del getter correspondiente.

getDescendants(**int** generations); **Sí** puede haber repeticiones. Un doctor puede ser descendiente de otro en varios niveles.

getSiblings(); **Sí** puede haber repeticiones. Diferentes supervisores pueden dirigir la tesis al mismo estudiante.

3 Estudio empírico del coste

*3.1 Realice un estudio empírico del coste temporal de los métodos de consulta de la clase **DoctorS** en el que se mida cómo varía el coste según el número de doctores que haya en la Academia. Compárelo con el estudio teórico realizado en la pregunta 1.2.*

Para el estudio empírico se ha realizado por cada llamada 10^5 iteraciones. La siguiente tabla muestra los resultados obtenidos de tiempo de procesamiento del método getDescendants().

G	Descendientes				
K	1	2	3	4	5
1	26	25	21	21	24
	20	25	19	20	22
2	196	325	542	803	1064
	20	21	30	24	23
	182	318	465	793	1011
3	150	402	1424	4128	10066
	18	24	19	26	17
	173	353	485	832	1015
	160	500	1470	4204	9512
4	121	1410	7912	35837	163421
	25	32	17	22	14
	198	274	479	708	884
	174	428	1326	3797	8923
	128	1131	7792	33724	164766
5	127	3439	41501	521183	6613066

Las filas representan el número de generaciones *G* que hay en la academiaS. Las columnas representan el número de estudiantes *K* que tiene cada doctor, salvo los de la última generación.

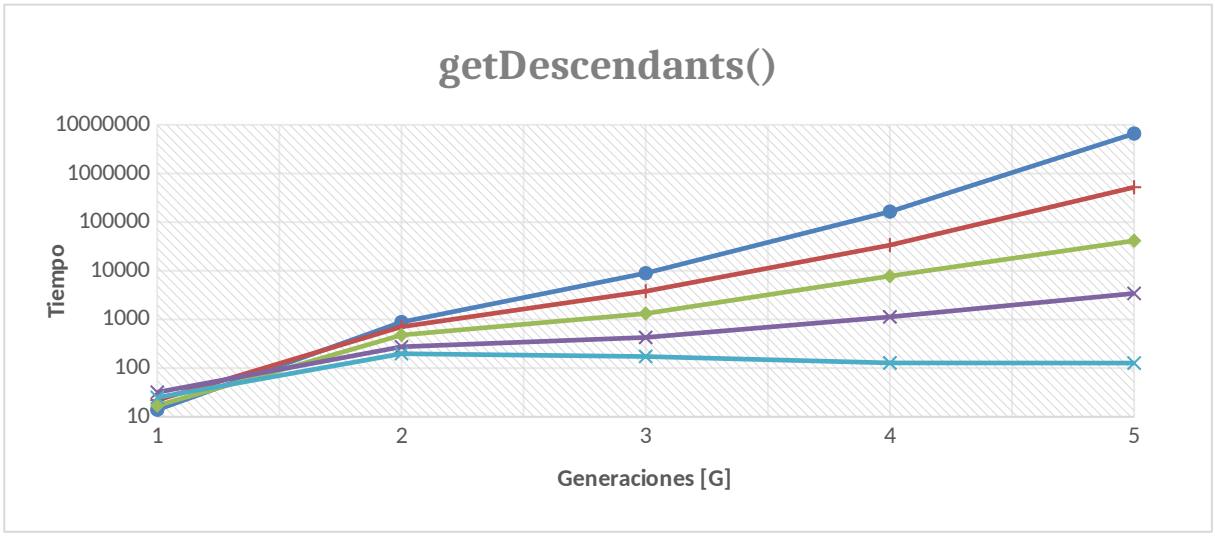
Para cada valor de G se hace un recorrido llamando al método con argumentos crecientes desde 1 hasta G. De ahí que para cada valor de G exista un número igual de filas.

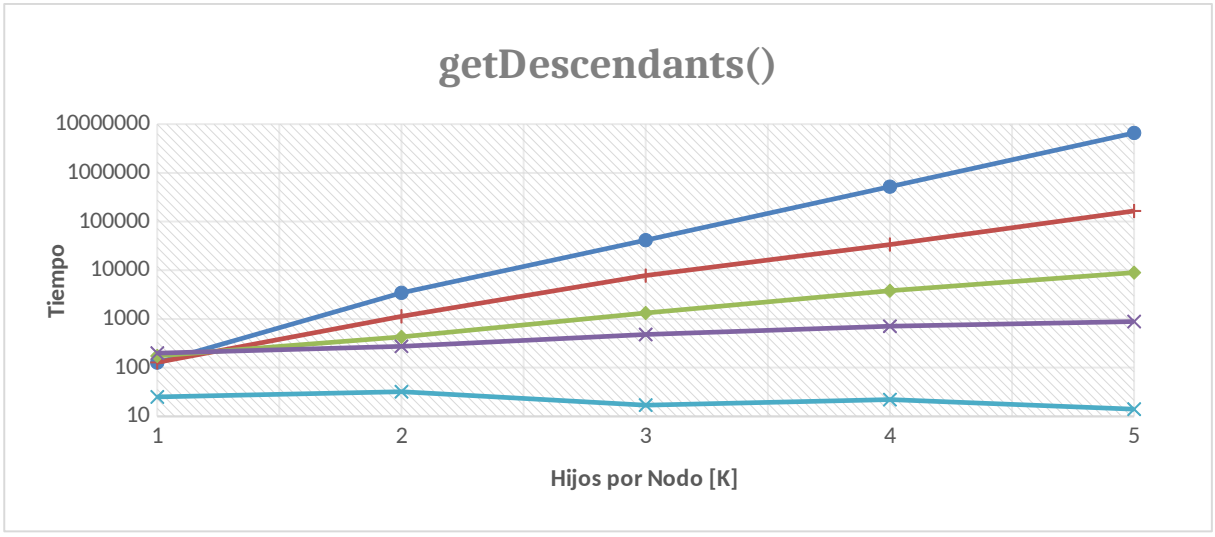
La primera conclusión que se obtiene es que el tiempo de ejecución de este método no depende de la cantidad total de doctores contenidos en la academia, sino de la combinación concreta de los valores G y K.

Esto se ve en que los valores de tiempo para una K determinada, por ejemplo K = 2, y un valor de generaciones pasadas como parámetro, por ejemplo 2, no depende del número total de generaciones G que tenga la academia, esto es, no depende del número total de doctores. Véase los valores resaltados en rojo.

De la anterior conclusión puede deducirse que la información relevante para el estudio de coste computacional se encuentra contenida en la región sombreada en azul.

A continuación, se muestra esta información condensada en dos gráficas. En la primera se puede apreciar el carácter exponencial (cóncavo en gráfico logarítmico) del tiempo respecto del número de generaciones G (para una K constante).





En la segunda se puede ver el carácter potencial (lineal en gráfico logarítmico) del tiempo respecto del número de hijos K (para un número G constante).

Lo anterior concuerda razonablemente con las predicciones teóricas para este método de consulta. Sin embargo, para los métodos de consulta getAncestors() y getSiblings() la práctica no parece seguir a la teoría. Como se muestra en las dos siguientes tablas, el comportamiento del tiempo de ejecución parece ser completamente independiente del número de doctores de la academia, así como de la distribución concreta de los valores G y K. Estos métodos presentan un coste constante.

G	**Ancestros**				
K	1	2	3	4	5
1	35	41	26	32	12
	23	13	24	44	34
2	68	47	54	68	76
	13	37	10	22	8
	74	72	89	94	113
3	61	43	29	28	121
	14	15	9	12	11
	87	189	116	33	31
	37	166	101	23	28
4	98	195	121	47	36
	37	10	10	12	11
	78	50	46	85	196
	77	65	24	87	102
	69	45	30	102	175
5	128	115	42	123	182

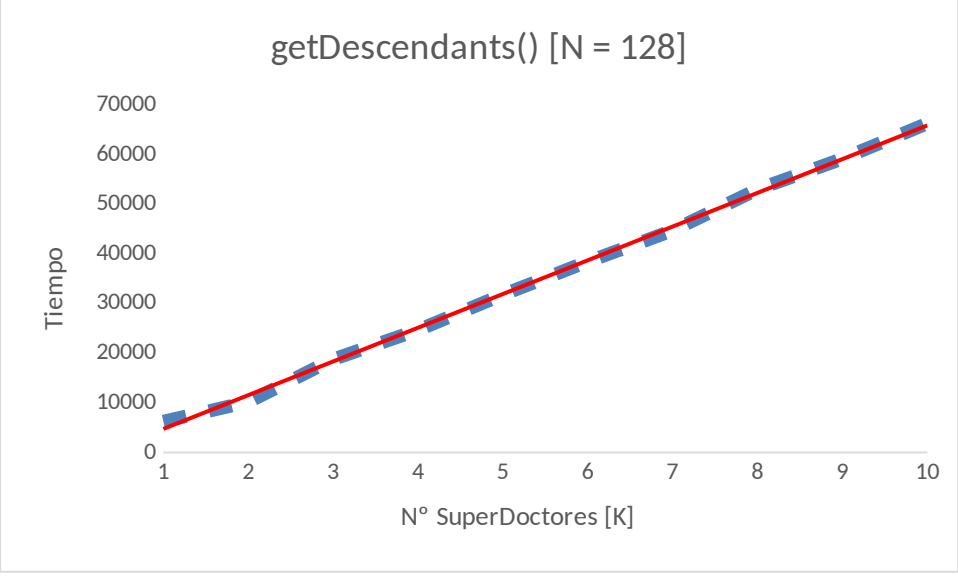
G	**Hermanos**				
K	1	2	3	4	5
1	163	132	137	123	156
2	88	80	91	57	87
3	40	61	52	48	63
4	35	61	41	35	35
5	31	65	52	37	81

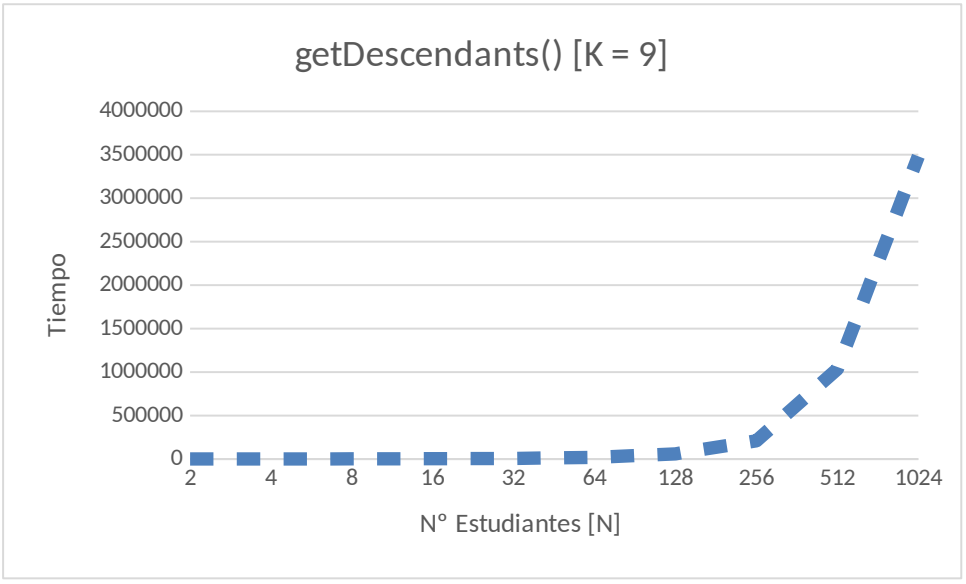
Un posible motivo es que el estudio teórico no sea correcto, mientras que otro podría ser que las pruebas no se han hecho con valores adecuados para resaltar correctamente el comportamiento asintótico buscado.

*3.2 Realice un estudio empírico del coste temporal de los métodos de consulta de la clase **DoctorC** en el que se mida cómo varía el coste según el número de doctores que haya en la Academia. Compárelo con el estudio teórico realizado en la pregunta 2.4.*

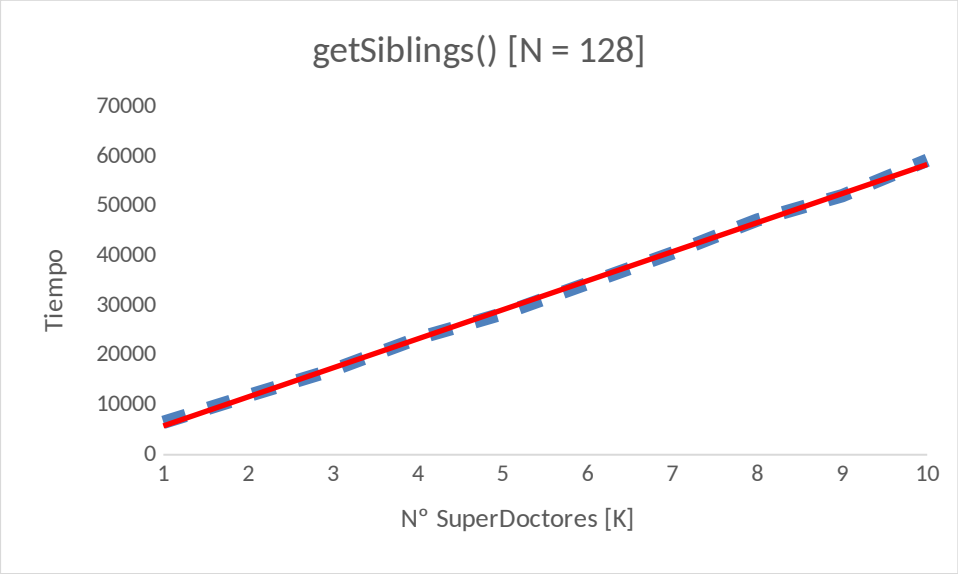
En cuanto a los métodos consultores de la clase DoctorC se ha obtenido los siguientes resultados, utilizando 10^5 iteraciones por cada llamada.

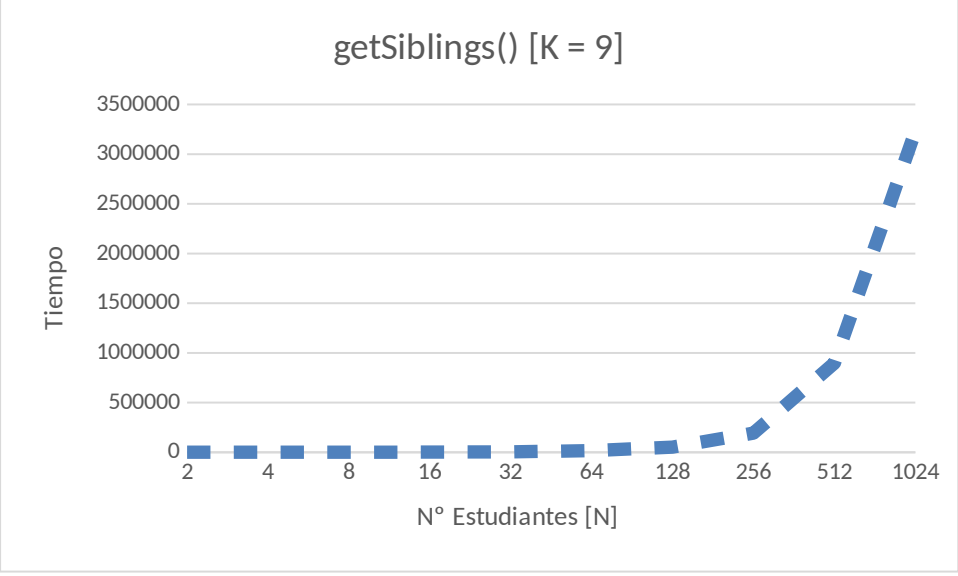
		getDescendants()									
	K										
N		2	4	8	16	32	64	128	256	512	1024
	1	112	134	160	290	635	1664	6238	20328	101837	327963
	2	163	182	264	535	1192	3282	10157	44311	165683	688599
	3	230	223	383	729	1678	4947	18739	74164	246764	913667
	4	263	369	501	1084	2631	6986	24622	93226	302136	1200313
	5	307	396	669	1333	2875	8356	31756	106581	573112	1537316
	6	397	454	834	1606	4113	11584	38415	137305	617550	2247976
	7	469	648	1074	1927	4901	13129	44652	175341	791780	2580653
	8	554	650	1192	2476	5901	16087	52950	184237	889170	3268617
	9	554	770	1571	3005	6369	18349	59119	209413	1035646	3482665
	10	695	920	1622	2975	6607	20844	66208	234199	1157311	4293729



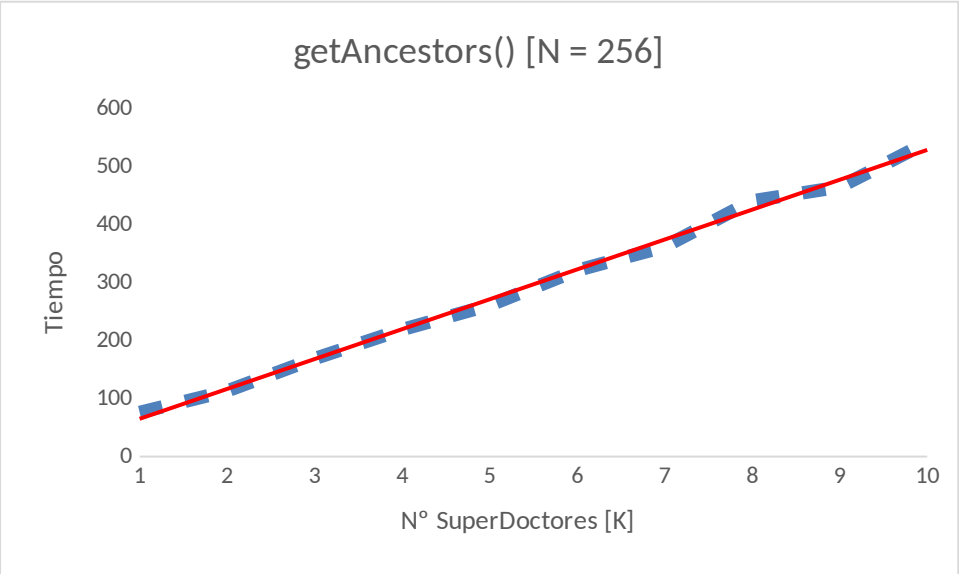

getDescendants() [K = 9]
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Tiempo
2
4
8
16
32
64
128
256
512
1024
N° Estudiantes [N]

	getSiblings()									
K										
N	2	4	8	16	32	64	128	256	512	1024
1	137	128	173	273	689	1957	6500	24186	91275	398895
2	139	122	163	468	1197	3276	11874	44571	151292	655372
3	149	163	216	611	1572	4884	17066	71649	256364	925449
4	138	197	285	714	2121	6649	23427	84992	336055	1425176
5	106	208	305	920	2594	8489	28371	105184	427512	1513137
6	168	168	389	1118	2978	10019	34487	131160	764230	2237433
7	138	219	404	1188	3452	11814	40583	163309	499373	2544980
8	185	216	416	1298	4044	12879	47238	172882	910351	3185219
9	201	300	462	1335	4427	15170	52134	191995	897380	3233863
10	212	245	541	1706	4794	16381	59281	223427	959497	4107469





		getAncestors()									
	K										
N		2	4	8	16	32	64	128	256	512	1024
	1	71	57	82	60	101	86	102	76	199	85
	2	133	115	106	86	115	132	138	113	114	136
	3	178	174	147	313	179	216	209	169	175	154
	4	236	240	263	238	248	253	241	219	137	141
	5	277	263	305	316	322	248	285	261	216	278
	6	372	307	319	318	305	297	285	321	458	317
	7	392	416	421	393	387	385	314	362	283	342
	8	574	397	421	324	375	365	386	441	563	342
	9	517	501	486	415	455	464	445	465	643	443
	10	572	524	570	438	570	519	501	542	543	748



Cada método presenta carácter lineal $O(n)$ respecto del número de SuperDoctores. Además, el método getAncestors() tiene un coste constante $O(1)$ respecto del número de estudiantes, mientras que getDescendants() y getSiblings() aproximadamente multiplican por 4 el tiempo cada vez que se dobla el número de estudiantes, lo cual representa un coste cuadrático $O(n^2)$.