UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario en Computación Grid y Paralelismo

Trabajo Fin de Máster

Paralelismo para la resolución de cuadrados mágicos formados por números primos

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario en Computación Grid y Paralelismo

Trabajo Fin de Máster

Paralelismo para la resolución de cuadrados mágicos formados por números primos

Autor: Jesús Redondo García

Tutor: Juan Antonio Gómez Pulido

Co-Tutor/es: Miguel Cárdenas Montes

1 ÍNDICE GENERAL DE CONTENIDOS

2 ÍNDICE DE FIGURAS

3 PALABRAS CLAVE

Paralelismo.

Cuadrado mágico.

Números primos.

OpenMP.

Anidación.

4 RESUMEN

Por la sencillez de su planteamiento y sus curiosas propiedades, los cuadrados mágicos han sido objeto de estudio y admiración por los matemáticos durante muchos siglos. En este documento describimos un estudio sobre la composición de cuadrados mágicos formados por números primos. La composición de este tipo de cuadrados es compleja porque no existe un método determinista, es decir, una heurística de formación de nuevos cuadrados. Por ejemplo, en un cuadrado relativamente pequeño, de tamaño 7x7, formado por 49 elementos tendríamos las siguientes combinaciones diferentes: 49! = 6x1062. Esto dificulta su tratamiento computacional.

Inspirado por el cuadrado mágico del prisionero (1), y aprovechando las capacidades de cálculo que ofrece la computación paralela, se formula un modelo denominado CCMSFP (Creador de Cuadrados Mágicos Superanidados Formados por Primos) para componer cuadrados mágicos de gran tamaño compuestos por primos. Asimismo, la inclusión de una nueva restricción, el anidamiento, simplifica el espacio de búsqueda al añadir una nueva restricción, permitiendo acelerar el proceso a la hora de ordenar los primos en el cuadrado. Los resultados obtenidos tras la implementación de CCMSFP reducen el número de combinaciones probadas para el 7x7 a 2x107.

En este trabajo se proponen dos implementaciones para solucionar cuadrados mágicos formados por primos. La primera variante del algoritmo permite componer cuadrados de gran orden en un tiempo reducido. Por ejemplo, un cuadrado de tamaño 17x17 toma un tiempo menor de 10 segundos. Además, encuentra una gran cantidad de cuadrados con las mismas características, pero de menor tamaño durante el proceso de ejecución.

La segunda variante del algoritmo funciona exclusivamente para los cuadrados de tamaño 7x7 y permite asegurar, para una combinación de pares, si existe una solución.

El algoritmo CCMSFP irrumpe como un método inédito para construir cuadrados mágicos compuestos por primos. Sus principales características involucran una eficacia temporal muy alta y el uso de la plataforma OpenMP como marco computacional. La utilización de este algoritmo podría tener un impacto en la ciencia de la encriptación e incluso abrir nuevas vías para la resolución de la teoría de los números primos.

5 INTRODUCCIÓN

Los enigmas numéricos han sido siempre objeto de admiración por matemáticos y estudiosos. En el caso de los cuadrados mágicos, se tiene constancia de su existencia desde el 2800 A.C. en la región de China. Tras casi 5000 años todavía siguen siendo objeto de estudio y misterio por muchos.

Debido a que el planteamiento de un cuadrado mágico es sencillo de entender y puede ser explicado en muy poco tiempo, su popularidad es notable entre los apasionados a los acertijos. Sin embargo, encontrar una solución para un cuadrado mágico requiere un análisis y un procesamiento concienzudos, lo que los hace candidatos perfectos para los libros de texto de aprendizaje de matemáticas.

Otro de los motivos por los cuáles los cuadrados mágicos parecen fascinar a los matemáticos es las maravillosas propiedades que estos pueden tomar. Algunos de los ejemplos más famosos de cuadrados mágicos con propiedades especiales son los siguientes:

- El cuadrado de Dürer (2), de tamaño 4x4. La suma de los números que forman cada una de sus filas, columnas y diagonales vale 34. Está compuesto por los números del 1 hasta el 16:

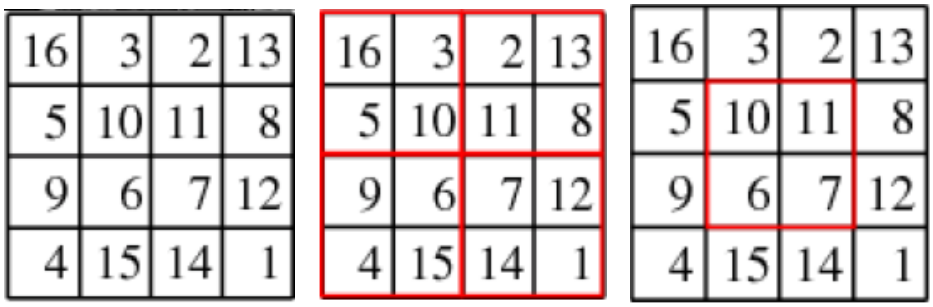

Ilustración 1 - El cuadrado de Dürer tiene la peculiaridad de que todos los cuadrados de 2x2 interiores que se pueden formar también suman 34.

- El cuadrado mágico de Benjamin Franklin, de tamaño 8x8. Compuso este cuadrado mágico cuyas filas y columnas suman 260 (no las diagonales):

14	3	62	51	46	35	30	19
52	61	4	13	20	29	36	45
11	6	59	54	43	38	27	22
53	60	5	12	21	28	37	44
55	58	7	10	23	26	39	42
9	8	57	56	41	40	25	24
50	63	2	15	18	31	34	47
16	1	64	49	48	33	32	17

Ilustración 2 El cuadrado de Benjamin Franklin esconde tantas curiosidades que es difícil responder cómo se construyó.

Algunas de las peculiaridades de este cuadrado son las siguientes: cada mitad de una fila y de una columna suma 130 (la mitad de lo que suman enteras), los cuatro números de las esquinas y los cuatro números centrales también suman 130, la suma de los elementos de cada cuadrado 2x2 que tomemos es 130 y los cuatro elementos de una diagonal ascendente junto con los cuatro de la correspondiente descendiente también suman 260.

- El cuadrado de la sagrada familia. La suma de los números que forman cada una de sus filas, columnas y diagonales vale 33. Para la composición de este cuadrado se han repetido los números 10 y 14 y se han eliminado el 12 y el 16.

Ilustración 3 Fragmento de "la sagrada familia", en Barcelona. Podemos apreciar cómo los cuadrados mágicos también han influido en el arte.

Hasta ahora hemos presentado muchas de las curiosidades que acompañan a los cuadrados mágicos y que los han hecho tan atractivos a lo largo de los años, quizás también sea necesario preguntarnos por la utilidad de los mismos. ¿Por qué componer y jugar con los cuadrados mágicos? Existen algunas aplicaciones prácticas estadísticas de los cuadrados mágicos, en campos como la colocación y ordenación de elementos o en campos estadísticos (3).

6 DESCRIPCIÓN DEL PROBLEMA

El enunciado del problema planteado es realmente simple: Crear cuadrados mágicos compuestos única y exclusivamente por números primos.

Para ello, nuestro planteamiento principal se centra en encontrar un método generador de este tipo de cuadrados del mismo modo que existen métodos para componer automáticamente cuadrados mágicos de tamaño tan grande como se desee siempre que estos estén formados por números enteros.

Las condiciones que deben cumplir los cuadrados son:

1. Que sean mágicos. Todas las filas, columnas y diagonales mayores del cuadrado deben sumar la misma cantidad. Este es el hecho que determina que un cuadrado sea mágico:

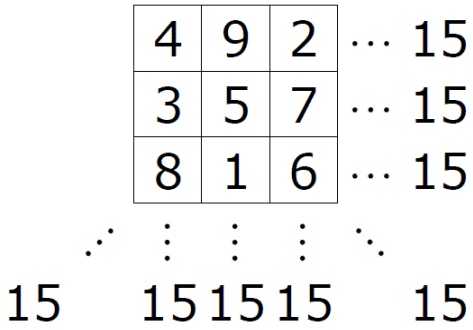

Ilustración 4 Cuadrado mágico de 3x3 formado por enteros

2. Todos los números que forman el cuadrado deben ser números primos. El número uno no forma parte de los números primos, por lo tanto, no puede ser utilizado.

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67
71	73	79	83	89	97	101	103	107	109	113	127	131	137	139	149	151	157	163
167	173	179	181	191	193	197	199	211	223	227	229	233	239	241	251	257	263	269
271	277	281	283	293	307	311	313	317	331	337	347	349	353	359	367	373	379	383
389	397	401	409	419	421	431	433	439	443	449	457	461	463	467	479	487	491	499
503	509	521	523	541	547	557	563	569	571	577	587	593	599	601	607	613	617	619
631	641	643	647	653	659	661	673	677	683	691	701	709	719	727	733	739	743	751
757	761	769	773	787	797	809	811	821	823	827	829	839	853	857	859	863	877	881
		883	887	907	911	919	929	937	941	947	953	967	971	977	983	991	997	

Ilustración 5 Listado de todos los números primos menores que 1000

3. No se pueden repetir ninguno de los números que componen el cuadrado.

Las restricciones presentadas anteriormente especifican el tipo de cuadrados que se tienen que componer. Algorítmicamente no sería muy complicado crear un programa que sea capaz de crear este tipo de cuadrados a base de fuerza bruta. Es decir, teniendo un tiempo ilimitado se pueden componer cuadrados de gran tamaño a base de ir colocando primos y comprobar si se cumplen las tres condiciones mencionadas anteriormente.

7 OBJETIVOS

Según hemos visto anteriormente el objetivo es generar un cuadrado mágico compuesto exclusivamente por números primos. En este cuadrado todas las columnas, filas y diagonales suman la misma cantidad. Por ejemplo:

101	47	31	79
73	61	71	53
43	67	59	89
41	83	97	37

Ilustración 6 Ejemplo de cuadrado mágico compuesto de primos de tamaño 4x4

Como se ha citado anteriormente, las bases del problema son muy sencillas. Es uno de esos acertijos que se pueden explicar a un compañero en 5 minutos mientras se toma un café, porque son sencillos de asimilar, y porque las operaciones que intervienen son básicas (sumas). Pero a su vez es un problema muy complejo de resolver cuando sólo se utilizan números primos.

Como fase inicial para abordar el problema hemos hecho un estudio de las soluciones ya propuestas y de los métodos que se han utilizado a lo largo de los años para componer cuadrados mágicos de distintos tipos.

8 ESTUDIO GENERATIVO DE CUADRADOS MÁGICOS DE PRIMOS

Si nos referimos exclusivamente a los cuadrados mágicos formados por números primos la documentación escrita hasta ahora es muy poca. A continuación mostramos algunas de las fuentes que han servido de inspiración:

A. El estudio del italiano Tognon Stefano. Toda la información que hemos encontrado de su investigación ha sido extraída de su página web personal (4). De su trabajo destaca un estudio sobre la generación de cuadrados mágicos formados por primos mediante ordenador. Para ello desarrolla un script que utiliza tanto heurística como fuerza bruta para componer los cuadrados.

   El algoritmo comienza identificando todos los números primos desde el 1 hasta el primo n2 -1, siendo n el lado del cuadrado. De este modo el autor es capaz de asegurar cuál debe ser la constante mágica para un determinado orden de cuadrado.

   El método que implementa consta de los dos siguientes pasos:

   a. Siendo el lado del cuadrado n, se rellena el cuadrado con el 1 más los n2-1 primeros números primos en un orden semi-aleatorio. Esta semi-aleatoriedad implica que la distancia entre la constante mágica y la suma de cada columna no es muy grande. Básicamente lo que hace es colocar los números primos de menor a mayor desde la esquina superior izquierda, primero rellenando la filas y luego las columnas.

1	3	5	7	11	13	17	19	23	29	31	37	41	43	47	9308
113	109	107	103	101	97	89	83	79	73	71	67	61	59	53	8370
127	131	137	139	149	151	157	163	167	173	179	181	191	193	197	7200
281	277	271	269	263	257	251	241	239	233	229	227	223	211	199	5964
283	293	307	311	313	317	331	337	347	349	353	359	367	373	379	4616
463	461	457	449	443	439	433	431	421	419	409	401	397	389	383	3240
467	479	487	491	499	503	509	521	523	541	547	557	563	569	571	1808
659	653	647	643	641	631	619	617	613	607	601	599	593	587	577	348
661	673	677	683	691	701	709	719	727	733	739	743	751	757	761	-1090
863	859	857	853	839	829	827	823	821	811	809	797	787	773	769	-2682
877	881	883	887	907	911	919	929	937	941	947	953	967	971	977	-4252
1069	1063	1061	1051	1049	1039	1033	1031	1021	1019	1013	1009	997	991	983	-5764
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151	1153	1163	1171	1181	1187	-7320
1291	1289	1283	1279	1277	1259	1249	1237	1231	1229	1223	1217	1213	1201	1193	-9306
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373	1381	1399	1409	1423	1427	-10680
24	96	72	60	66	30	58	48	0	-10	-46	-50	-74	-96	-86	-68	-52

Ilustración 7 Colocación inicial de los números. La distancia a la constante mágica en las columnas es muy pequeña con esta colocación.

b. Se ajustan columnas, filas y diagonales para que su suma tenga el valor de la constante mágica. Este proceso se realiza por fuerza bruta, pero teniendo en cuenta que para modificar el valor de una columna sólo hay que modificar el valor de otra, cambiando un par de números entre ellas (esto es aplicable con las filas). De este modo, se van ajustando las columnas y filas de dos en dos.

Hay ejemplos de cuadrados mágicos construidos con este algoritmo de orden muy grande (hasta 124). Incluso se encuentra el código fuente utilizado para generar estos cuadrados. Stefano indica a lo largo de su documentación que el algoritmo ha ido sufriendo modificaciones para refinar los cambios de pares, haciendo el proceso más rápido. Por desgracia falta documentación en este aspecto, por lo que sólo conocemos la aproximación inicial explicada anteriormente y que se han hecho refinamientos en el código para acelerar.

Este método nos ha sido útil para reconocer algunas propiedades interesantes de los cuadrados mágicos, pero no es un método válido para la generación de cuadrados mágicos compuestos exclusivamente por primos porque utiliza el número uno. Utilizar el uno facilita mucho los cálculos, pero no es un número primo y como tal no se debe utilizar en la generación del cuadrado mágico.

B. Existen numerosos algoritmos para generar cuadrados mágicos con números enteros. La necesidad de cálculo para componer uno de estos cuadrados es muy baja, pero es interesante revisar las metodologías que se han utilizado porque podrían ser extrapoladas a cuadrados mágicos formados exclusivamente por primos. Entre este tipo de algoritmos destacamos:

a. Método de Loubere, sólo es válido para cuadrados mágicos de orden impar. Veamos en qué consiste construyendo un cuadrado de orden 5:

Colocamos el 1 en la posición central de la fila superior y vamos rellenando en diagonal, es decir, el 2 se coloca en la posición (5,4) (fila 5, columna 4), el 3 en la posición (4,5), el 4 en la (3,1), y así sucesivamente. Si al intentar colocar un número en la posición que debe ocupar nos la encontramos ya ocupada, colocamos ese número justo debajo del último colocado y continuamos colocando en diagonal.

El cuadrado mágico de orden 5 obtenido con este procedimiento es:

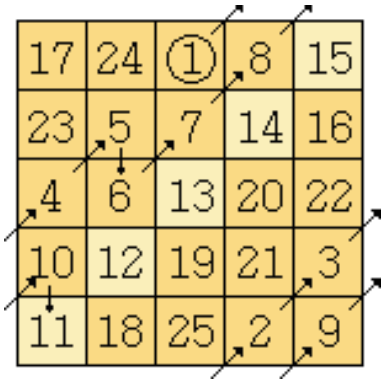

Ilustración 8 Esquema de generación del método de Loubere

b. Método de Bachet, sólo válido para cuadrados mágicos de orden impar.

Es un método parecido al de Loubere. Consiste en crear el cuadrado a partir de una construcción gráfica. Veamos en qué consiste construyendo también un cuadrado mágico de orden 5:

Dibujamos un cuadrado de 5x5. A partir de ahí disponemos los números del 1 al 25 como muestra la siguiente figura:

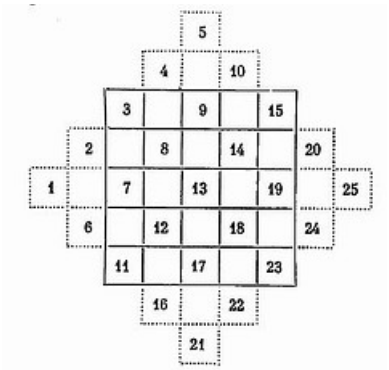

Ilustración 9 Esquema inicial del método de Bachet

Ahora colocamos los números que han quedado fuera del cuadrado en las posiciones opuestas que quedaron libres:

3	16	9	22	15
20	8	21	14	2
7	25	13	1	19
24	12	5	18	6
11	4	17	10	23

Ilustración 10 Resultado de colocar los números que están fuera del cuadrado en su casilla libre opuesta mediante el método de Bachet

c. Método para cuadrados mágicos de orden múltiplo de cuatro. Al igual que el método anterior es un método gráfico. Se empieza construyendo el cuadrado con los números dispuestos de forma consecutiva. Una vez hecho esto conservamos la submatriz central de orden n/2 y las cuatro submatrices de las esquinas de orden n/4. Los números restantes se giran 180º respecto del centro del cuadrado, o si se prefiere se recolocan en orden decreciente.

   Para un cuadrado de lado 8 obtenemos el siguiente cuadrado mágico de orden 8:

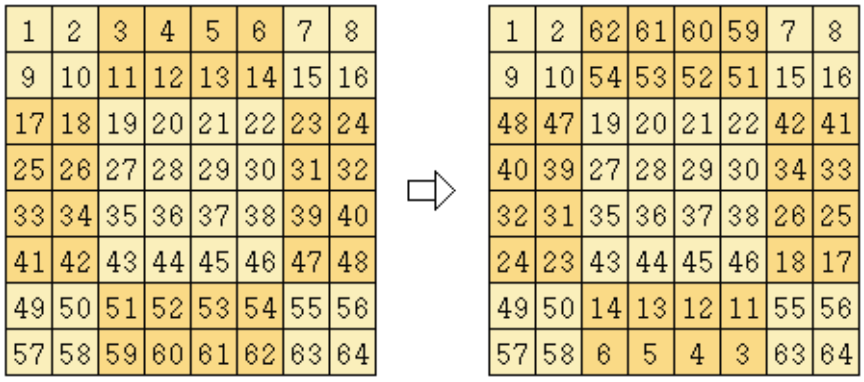

Ilustración 11 Ejemplo de plantilla para un cuadrado de orden múltiplo de 4

   Partimos del cuadrado con los números dispuestos consecutivamente y recolocamos cada uno de los cuadros en oscuro con su simétrico desde el centro del cuadrado.

   Este método funciona siempre que la plantilla de cuadrados oscuros sea simétrica con respecto del centro para cada uno de los cuadrados de 4x4.

C. Cuadrado del prisionero. Ha habido un cuadrado que nos ha servido de gran inspiración para el algoritmo que proponemos. El autor de este cuadrado es desconocido, se dice que fue hecho por una persona mientras estaba en la cárcel, de ahí su nombre del “cuadrado del prisionero”. Fue publicado por primera vez en 1979 en “Magic and antimagic squares” (Madachy, J. S. "Magic

and Antimagic Squares." Ch. 4 in Madachy's Mathematical Recreations. New York: Dover, pp. 85-113, 1979). El cuadrado en cuestión es el siguiente:

1153	8923	1093	9127	1327	9277	1063	9133	9661	1693	991	8887	8353
9967	8161	3253	2857	6823	2143	4447	8821	8713	8317	3001	3271	907
1831	8167	4093	7561	3631	3457	7573	3907	7411	3967	7333	2707	9043
9907	7687	7237	6367	4597	4723	6577	4513	4831	6451	3637	3187	967
1723	7753	2347	4603	5527	4993	5641	6073	4951	6271	8527	3121	9151
9421	2293	6763	4663	4657	9007	1861	5443	6217	6211	4111	8581	1453
2011	2683	6871	6547	5227	1873	5437	9001	5647	4327	4003	8191	8863
9403	8761	3877	4783	5851	5431	9013	1867	5023	6091	6997	2113	1471
1531	2137	7177	6673	5923	5881	5233	4801	5347	4201	3697	8737	9343
9643	2251	7027	4423	6277	6151	4297	6361	6043	4507	3847	8623	1231
1783	2311	3541	3313	7243	7417	3301	6967	3463	6907	6781	8563	9091
9787	7603	7621	8017	4051	8731	6427	2053	2161	2557	7873	2713	1087
2521	1951	9781	1747	9547	1597	9811	1741	1213	9181	9883	1987	9721

Ilustración 12 Cuadrado del prisionero

Es un cuadrado de tamaño 13x13 conformado exclusivamente por números primos. La principal peculiaridad del cuadrado del Prisionero es que cada uno de sus cuadrados interiores centrados en la celda central: 11x11, 9x9, 7x7, 5x5 y 3x3 son cuadrados mágicos.

La relación entre la constante mágica de cada cuadrado y su superior es siempre fija, con valor 10874.

Este tipo de cuadrados en el cual todos los subcuadrados centrales también son mágicos se denominan cuadrados mágicos anidados: nested magic squares.

Que todos los cuadrados estén anidados por capas, y que la variación en cada capa se mantenga fija nos proporcionó la idea de que tal vez se pudiera establecer un método para ir añadiendo capas en una progresión constante haciendo posible crear cuadrados cada vez más grandes.

9 BASES PARA LA GENERACIÓN DE CUADRADOS PRIMOS ANIDADOS

Como hemos revisado en los puntos anteriores, existen numerosos métodos deterministas para generar cuadrados mágicos. Así pues, para crear cuadrados de un tamaño determinado lo único que hay que hacer es seguir uno de esos métodos. En un número finito de pasos obtendremos el cuadrado deseado.

El problema a la hora de generar cuadrados mágicos de números primos es que no existe (o no se ha encontrado) un método determinista de generación. Hasta ahora las aproximaciones han estado focalizadas en la utilización de fuerza bruta, utilizando como herramientas la potencia de computación para realizar backtracking.

La solución expuesta en este documento trata de utilizar una heurística de anidamiento de cuadrados mágicos parecida a la que aparece en el cuadrado mágico del prisionero. Al utilizar anidamiento, el problema se puede atacar en pequeños fragmentos (cuadrados interiores), y se puede secuenciar la búsqueda del cuadrado mágico final, pese a que para cada tramo se necesite backtracking o fuerza bruta.

9.1 El Algoritmo teórico CCMSFP

Las siglas de CCMSFP hacen referencia a Creador de Cuadrados Mágicos Superanidados Formados por Primos.

Así como el cuadrado del prisionero se puede descomponer en subcuadrados concéntricos de tamaño más pequeño, la idea principal de CCMSFP consiste en generar cuadrados concéntricos cada vez de mayor tamaño.

El algoritmo que vamos a describir construye cuadrados mágicos de lado impar 1x1, 3x3, 5x5, 7x7… El fundamento metodológico está claramente influenciado por el cuadrado del prisionero: a partir de un primo base central se van a ir añadiendo capas (3x3, 5x5, 7x7…) para agrandar su tamaño.

El primo central es clave, porque dependiendo de sus características podemos asegurar que se van a cumplir una serie de condiciones que limitan el tamaño máximo del cuadrado que se puede alcanzar. Entraremos con más detalle en estas limitaciones a lo largo del documento. Para comenzar , presentamos los términos base que utilizaremos.

9.2 Términos importantes

Cuadrado mágico: Matriz cuadrada de números en el que la suma de cada una de sus columnas, filas y diagonales tiene el mismo valor.

Cuadrado mágico formado por primos: Cuadrado mágico en el que todos los números que lo conforman son números primos.

Cuadrado mágico anidado formado por primos (cmp): Cuadrado mágico formado por primos en el que cada uno de los cuadrados interiores centrados en la casilla central es también un cuadrado mágico. En numerosas partes de la documentación, para acortar, usaremos la expresión “cuadrados mágicos”, o simplemente “cuadrados” para denominar este tipo de cuadrados.

Orden o Tamaño (O): Número de filas o columnas que componen el cuadrado. Por ejemplo, el cuadrado mínimo con el que podemos trabajar tiene orden 3, es el de 3x3.

Centro del cuadrado (símbolo <>): Casilla central del cuadrado mágico. Todos los cuadrados mágicos que tratamos tienen orden impar, por lo tanto siempre va a haber una casilla central.

C =

67	1	43
13	37	61
31	73	7

Ilustración 13 Representación de cuadrado mágico 3x3

<C> = 37

Constante mágica (K): Sea C un cuadrado mágico, decimos que su constante mágica k tiene el valor que suma cada una de sus columnas, filas o diagonales mayores.

Anidación: Es una peculiaridad que se puede dar en los cuadrados mágicos. El ejemplo canónico de anidación aparece en el cuadrado del prisionero que hemos revisado anteriormente. Sea C un cuadrado mágico y C1, C2...Cn cada uno de sus cuadrados concéntricos desde la casilla central de C, decimos que ocurre la anidación si todos los cuadrados interiores C1, C2...Cn (3x3) también son cuadrados mágicos.

C =

5527	4993	5641	6073	4951
4657	9007	1861	5443	6217
5227	1873	5437	9001	5647
5851	5431	9013	1867	5023
5923	5881	5233	4801	5347

C1 =

9007	1861	5443
1873	5437	9001
5431	9013	1867

Ilustración 14 Cuadrados concéntricos en el cuadrado del prisionero

Decimos que C tiene anidación porque todos sus cuadrados interiores (sólo tiene uno, C1) también son cuadrados mágicos.

Anidar: Es un proceso que sirve para construir cuadrados mágicos de mayor tamaño a partir de uno de menor orden gracias a la propiedad de anidación. Sea C un cuadrado mágico anidado de orden O decimos que se ha producido anidamiento si somos capaces de crear un cuadrado mágico C'1 cuyo orden es O+2 y cuyo cuadrado interior más inmediato es C.

C =

5527	4993	5641	6073	4951
4657	9007	1861	5443	6217
5227	1873	5437	9001	5647
5851	5431	9013	1867	5023
5923	5881	5233	4801	5347

C'1 =

6367	4597	4723	6577	4513	4831	6451
4603	5527	4993	5641	6073	4951	6271
4663	4657	9007	1861	5443	6217	6211
6547	5227	1873	5437	9001	5647	4327
4783	5851	5431	9013	1867	5023	6091
6673	5923	5881	5233	4801	5347	4201
4423	6277	6151	4297	6361	6043	4507

Ilustración 15 Ejemplo de anidación en el cuadrado del prisionero

El cuadrado C'1 se ha producido por anidamiento a partir de C. C'1 se ha construido de tal manera de que sigue siendo mágico y su cuadrado interior más inmediato es C, por lo tanto, mantiene la anidación de C.

En este documento presentamos un método que permite construir cuadrados mágicos de orden cada vez mayor gracias a la anidación.

Cuadrado superanidado: Un cuadrado es superanidado si todos los cuadrados concéntricos se han producido por anidación. En nuestro caso siempre vamos a hablar de cuadrados mágicos superanidados, por lo que utilizaremos el término anidado y superanidado indistintamente, aunque en realidad no sea así.

Constante mágica de anidamiento (Ka): Sea C un cuadrado mágico con anidación. Decimos que Ka es igual a la diferencia entre la constante mágica de cualquiera de sus cuadrados interiores consecutivos. Todos los cuadrados interiores y los que se pueden anidar comparten la misma constante mágica de anidamiento.

En el ejemplo de la figura anterior podemos observar que K(C) = 27185 y K(C'1) = 38059 luego la constante de anidación Ka(C'1) = K(C'1) - K(C) = 38059 - 27185 =10874.

La constante mágica de anidamiento está directamente relacionada con el centro de un cuadrado de la siguiente manera:

Ka(C'1) = 2 <C'1>

Es decir, la constante de anidamiento es siempre dos veces el centro del cuadrado. La explicación a este fenómeno es fácil de entender si reflexionamos sobre la estructura que siguen los cuadrados mágicos anidados.

Realizaremos la comprobación por inducción matemática. Primero para el caso de anidación base, a partir de un sólo número construir la matriz de orden 3 (1 → 3). En segundo lugar demostrar que se cumple para el caso n → n+2 (recordemos que la matriz siempre crece de 2 en 2 con respecto al orden debido a la anidación).

Caso base, n=1:

Pensemos en la matriz más sencilla, la formada solamente por un elemento primo central y único:

C

Ilustración 16 Cuadrado de 1x1 con central

Para componer el cuadrado superior 3x3 habría que añadir 8 números primos con la siguiente distribución:

P_1	P_2	P_3
P'_4	C	P_4
P'_3	P'_2	P'_1

Ilustración 17 Colocación de primos para el 3x3

De tal modo que cada fila, columna y diagonal principal sume la misma cantidad (**K**).

A partir del gráfico podemos observar una serie de hechos:

1. La distribución en la que se colocan los dos primos opuestos con respecto al centro (no confundir con simétricos desde el centro) (los cuales llamaremos **pares, o complementarios**). De tal modo que la suma de cada primo (P1), junto con su complementario (P'1) y el central (C) es igual a **K**.

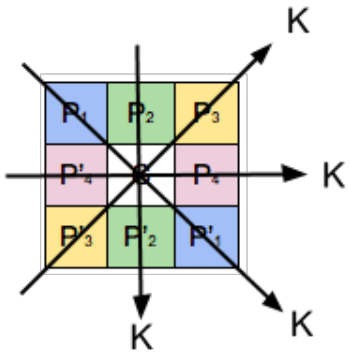

Ilustración 18 Los primos opuestos y el central suman K

Así pues podemos afirmar que la suma de cada par cumple: $P_x + P'_x = K - C$

NOTA: El subíndice "$_x$" hace referencia a cualquier par en el cuadrado

2. Fijándonos en la estructura del cuadrado...

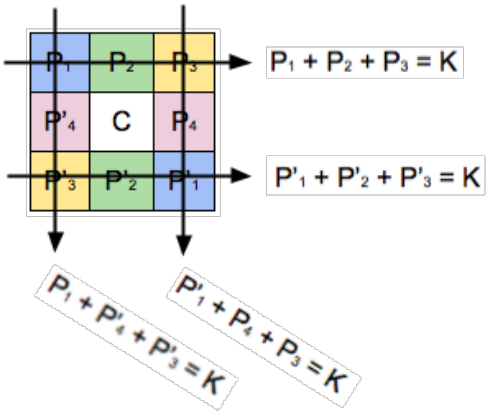

Ilustración 19 La suma de filas y columnas del 3x3 en el que no participa el central también son K

… y operando con las igualdades:

$$\begin{array}{l} P_1 + P_2 + P_3 = K \\ + \\ P'_1 + P'_2 + P'_3 = K \\ \hline P_1 + P'_1 + P_2 + P'_2 + P_3 + P'_3 = 2\,K \end{array}$$

Si sustituimos utilizando el hecho 1) → $P_x + P'_x = K - C$:

$(K - C) + (K - C) + (K - C) = 2K$

$3K - 3C = 2K$

$\boldsymbol{K = 3C}$

La constante mágica para cualquier cuadrado superanidado de orden tres es tres veces el lado del cuadrado. (Todos los cuadrados de orden 3 son superanidados).

Si consideramos $P_x + P'_x$ como la suma de un par (*Sp*) podemos determinar:

$P1 + P'1 + P2 + P'2 + P3 + P'3 = 2\,K$

$3\,Sp = 2K$

$\boldsymbol{Sp = \frac{2}{3}\,K}$

3. Teniendo en cuenta las dos igualdades que obtenemos en 2) podemos afirmar:

   $Sp = \frac{2}{3}\,K \rightarrow Sp = \frac{2}{3}\,3C \rightarrow$ **Sp= 2C = ka**

La suma de pares es igual a dos veces el centro del cuadrado. Como es obvio al fijarse en la estructura del cuadrado, Sp coincide siempre con la constante mágica de anidación. Así pues, si demostramos que **Sp = 2C** también se cumple para el cuadrado superanidado de orden n+2, demostraremos que la constante de anidación se mantiene fija y su valor es el mismo para cualquier anidación de un cuadrado superanidado.

El hecho de que **Sp = 2C** también acota superiormente la cantidad de cuadrados mágicos que se pueden superanidar para un determinado centro de cuadrado.

Caso n+2

Partimos de un cuadrado mágico superanidado de lado n al que denominamos $C_{interior}$:

$C_{interior}$ =

primo $_{1,1}$	primo $_{1,2}$	primo $_{1,3}$	primo $_{1,...}$	primo $_{1,n}$
primo $_{2,1}$	primo $_{2,2}$	primo $_{2,1}$	primo $_{2,...}$	primo $_{2,n}$
primo $_{3,1}$	primo $_{3,2}$	C	primo $_{3,...}$	primo $_{3,n}$
primo $_{...,1}$	primo $_{...,2}$	primo $_{...,3}$	primo $_{...,...}$	primo $_{...,n}$
primo $_{n,1}$	primo $_{n,2}$	primo $_{n,3}$	primo $_{n,...}$	primo $_{n,n}$

Ilustración 20 Cuadrado interior para caso n+2

Si se anidase, dicho cuadrado tomaría la forma:

$C_{anidado}$ =

P_1	P_2	P_3	$P_{...}$	P_n	P_{n+1}	P_{n+2}
P'_{n+3}	primo $_{1,1}$	primo $_{1,2}$	primo $_{1,3}$	primo $_{1,...}$	primo $_{1,n}$	P_{n+3}
$P'_{n+..}$	primo $_{2,1}$	primo $_{2,2}$	primo $_{2,3}$	primo $_{2,...}$	primo $_{2,n}$	$P_{n+..}$
P'_{2n+1}	primo $_{3,1}$	primo $_{3,2}$	C	primo $_{3,...}$	primo $_{3,n}$	P_{2n+1}
P'_{2n+2}	primo $_{...,1}$	primo $_{...,2}$	primo $_{...,3}$	primo $_{...,...}$	primo $_{...,n}$	P_{2n+2}
P'_{2n+3}	primo $_{n,1}$	primo $_{n,2}$	primo $_{n,3}$	primo $_{n,...}$	primo $_{n,n}$	P_{2n+3}
P'_{n+2}	P'_2	P'_3	$P'_{...}$	P'_n	P'_{n+1}	P'_1

Ilustración 21 Distribución de pares en la anidación n+2

Cada fila, columna y diagonal principal sumaría la misma cantidad, **k**:

Como consecuencia de la anidación, $<C_{interior}>$ == $<C_{anidado}>$ y $O(C_{anidado})$ == $O(C_{interior})$ + 2 == n + 2. Cualquiera suma de pares P - P', a la que venimos llamando Sp, es igual a la constante mágica de anidación:

P			C			P'

Ilustración 22 Ejemplo de un par en el caso n+2

Si nos fijamos en cualquier fila, columna o diagonal principal:

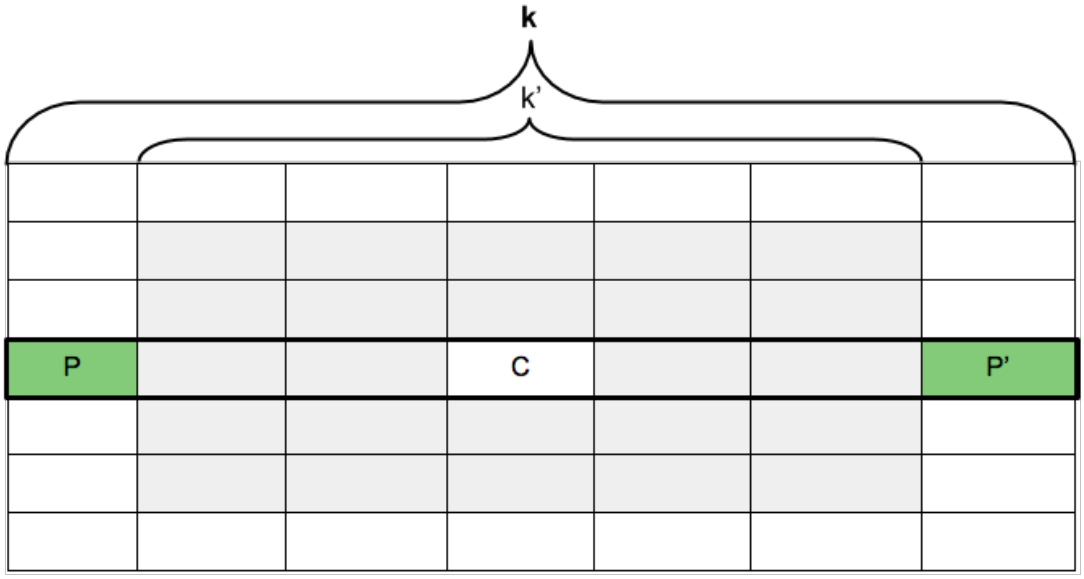

Ilustración 23 Relación entre la constante de anidación n y n+2

Podemos observar que **k** = k' + P + P' = k' + Sp

Luego Sp = **k** - k'

Podemos escribir k' en función del centro del cuadrado. Para ello tenemos que considerar que la suma cada uno los pares interiores del cuadrado anidado es igual a **2C**. Es algo que podemos considerar cierto porque lo hemos demostrado para el caso base n=1 (lo que queremos demostrar precisamente es que también se cumple para el caso n+2).

La cantidad de elementos en la fila resaltada de la imagen anterior que no son ni C, ni P, ni P' es n-1. Si el orden del cuadrado interior es n, entonces n-1 elementos son diferentes del centro. Llamemos a este conjunto como conjunto de elementos interiores.

Si seguimos trabajando sobre la figura anterior podemos observar que hay (n-1)/2 elementos a la izquierda y (n-1)/2 elementos a la derecha de C. Para cada elemento de la izquierda del centro debe haber otro en la derecha (colocado a la misma distancia con respecto del centro) que ha permitido formar cada una de las anidaciones interiores del cuadrado. En efecto, a estos pares de elementos los podemos denominar pares interiores y se tiene que cumplir que cada par interior es igual a **dos veces el centro del cuadrado** para que se haya producido anidación:

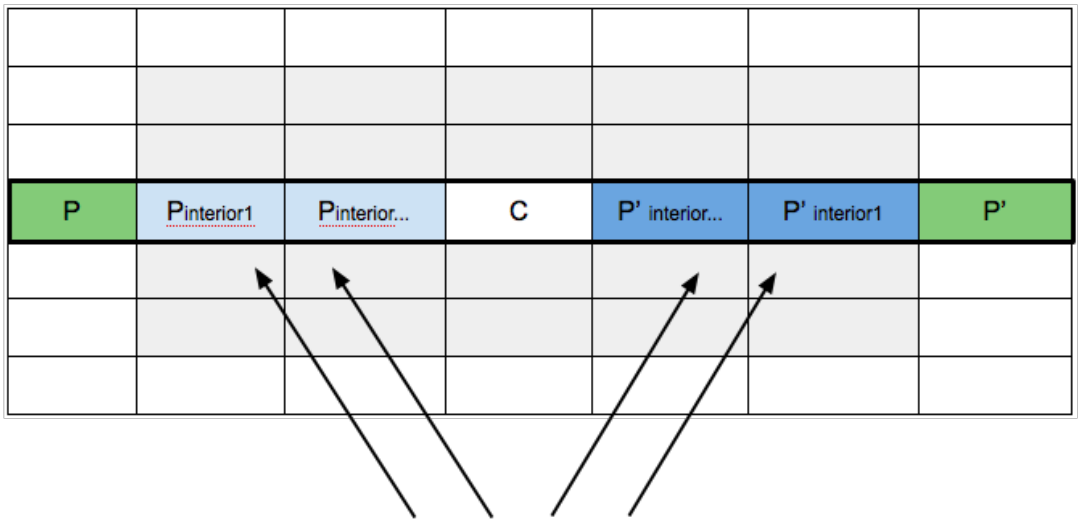

Ilustración 24 Fila completa interior de un n+2

Cada suma de par interior vale **2C**. Podemos afirmarlo porque lo demostramos en el caso base. Así pues **k'** = número de pares interiores x valor del par + C = (n-1)/2 x $Sp_{interiores}$ + C = (n-1)/2 x 2C + C = **nC**

Con las igualdades que hemos apuntado estamos en disposición de resolver el caso n+2. Comencemos resaltando de nuevo la estructura del cuadrado n+2:

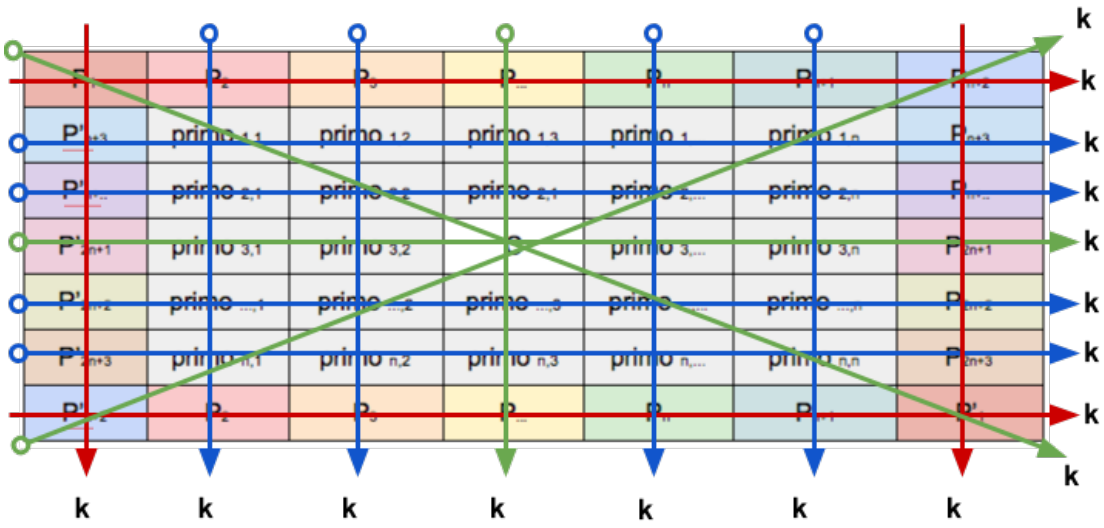

Ilustración 25 Estructura del cuadrado n+2

1. Teniendo en cuenta las flechas verdes y azules:

   Si nos fijamos, por ejemplo, en la flecha diagonal que parte desde la esquina superior izquierda hasta la esquina inferior derecha:

   $P1 + primo_{1,1} + primo_{2,2} + C + primo_{...,...} + primo_{n,n} + P'1 = k$

Sustituyendo:

$P1 + k' + P'1 = k$
$P1 + nC + P'1 = k$
$Sp + nC = k$

2. Teniendo en cuenta las flechas rojas:

Si nos fijamos, por ejemplo, en la primera flecha horizontal roja y en la última, podemos extraer el siguiente par de ecuaciones:

$$P1 + P2 + P3 + P... + P_n + P_{n+1} + P_{n+2} = k$$

$$P'1 + P'2 + P'3 + P'... + P'_n + P'_{n+1} + P'_{n+2} = k$$

Sumándolas obtenemos:

$$(P1 + P'1) + (P2 + P'2) + (P3 + P'3) + (P... + P'...) + (P_n + P'_n) + (P_{n+1} + P'_{n+1}) + (P_{n+2} + P'_{n+2}) = 2\mathbf{k}$$

$$Sp1 + Sp2 + Sp3 + Sp... + Spn + Spn+1 + Spn+2 = 2\mathbf{k}$$

Teniendo en cuenta que → $Sp1 = Sp2 = Sp3 = Sp... = Spn = Spn+1 = Spn+2$ llegamos a la siguiente igualdad:

$$(n+2)\,Sp = 2k$$

Así pues si sabemos que:

- Por flechas verdes y azules) → $Sp + nC = \mathbf{k}$
- Por flechas rojas) → $(n+2)\,Sp = 2\mathbf{k}$

sustituyendo **k** :

$$(n+2)\,Sp = 2(Sp + nC)$$

$$nSp + 2Sp = 2Sp + 2nC$$

$$nSp = 2nC$$

$$\mathbf{Sp = 2C}$$

Así pues, hemos demostrado:

1. La suma de cualquier par de primos que se sitúen en lados simétricos con respecto al centro del cuadrado formando anidación siempre tiene que sumar dos veces el centro del cuadrado.

2. No importa el nivel de superanidación que se esté llevando a cabo. La constante mágica de anidación siempre aumenta en el mismo orden, 2C.
3. La constante mágica para un cuadrado mágico superanidado de orden *O* es igual a O veces el centro de dicho cuadrado.

Estas tres propiedades son básicas para acelerar el proceso de composición de cuadrados mágicos. **Las restricciones mencionadas serán aplicables al algoritmo de generación porque delimitan el espacio de búsqueda de soluciones.**

Par: Consideramos un par con respecto a un cuadrado C, como una tupla de dos primos (p, p') tal que la distancia cada uno de sus componentes con respecto al centro del cuadrado es igual. Es decir:

Sea un cuadrado C, cuyo centro del cuadrado es <C>, consideramos que (p,p') es un par si : (p + p') / 2 = <C>

El concepto de par ya ha sido introducido cuando explicamos la constante de anidamiento **Ka**. Es importante comprender la importancia que tiene el **Par**, ya que sólo aquellos números que sean pares se pueden colocar en un cuadrado mágico superanidado (ya sea de primos o no).

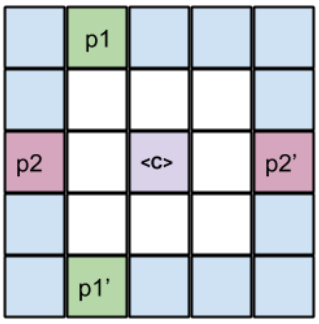

Ilustración 26 Pares en una anidación de 5x5 sobre 3x3

Como ya demostramos con anterioridad, *Ka = 2<C>*, por lo que cualesquiera dos primos que se utilicen para anidar y que estén situados en posiciones opuestas del cuadrado cumplen la peculiaridad de conformar un **Par**. Es decir, es un requisito necesario e indispensable que los números colocados en posiciones opuestas desde el centro (que no simétricas) formen un par.

A partir del estudio de las relaciones que hemos expuesto hasta ahora podemos plantear una serie de premisas que se tienen que cumplir para generar un cuadrado mágico superanidado formado por primos.

9.3 Premisas para generar cuadrados superanidados formado por primos.

La generación de cuadrados anidados parte desde un elemento primigenio al que llamamos elemento central de cuadrado. Este elemento central, compone por sí mismo el cuadrado 1x1 desde el que se producirá la anidación de todos los demás.

Si consideramos un cmp *C* de orden *O(C)*, entonces ese cuadrado tendrá $O(C)_xO(C)$ primos que lo componen. Uno de ellos es el primo central y los otros $O(C)_xO(C) - 1$ se agruparán en pares. *Es decir para todo cmp C ocurre que tiene* $(O(C)^2 - 1) / 2$ *pares*.

Estos pares vienen delimitados con respecto al elemento central de dos maneras:

1. Por una parte todos los pares de la forma P = (p,p') cumplen que (p+p')/2 = <C>
2. Por 1) podemos afirmar que hay un número limitado de pares dependiendo del primo central. La demostración es simple, para ello tenemos en cuenta:

   Si p y p' son primos y por lo tanto naturales, y **consideramos p como el elemento más pequeño del par**. Entonces p < <C> < p' porque:

   Si: p < p' y (p' + p) /2 = <C> → p' + p = 2<C>

   Es obvio que si p es menor que <C> , p' tiene que ser mayor que <C> para que su suma sea igual a 2C.

Luego todos los *p* de los **Pares** son primos menores que <C> por lo tanto están acotados superiormente por <C>.

Para que un número primo pueda ser candidato para ser central del cuadrado tiene que tener al menos los suficientes pares para rellenar ese cuadrado. Por ejemplo, si queremos construir un cmp de 5x5 se necesitan $(5_x5 -1) / 2 = 12$ pares.

El primo 13 nunca podría ser centro de un cuadrado de 5x5 porque sólo tiene los pares: (3,23) y (7,19). Le faltan 10 pares para poder ser candidato. El primo mínimo que tiene pares suficientes es el 347, que tiene 13 pares, uno más de los necesarios. El primo mínimo con los pares justos para formar un cuadrado de 5x5 es 409.

Que un primo tenga los pares necesarios lo convierte en candidato, pero no asegura que se pueda formar un cuadrado con esos pares. En el caso de los primos anteriores 347 y 409 sí que ocurre:

311	293	263	227	641
191	251	233	557	503
587	653	**347**	41	107
593	137	461	443	101
53	401	431	467	383

Es posible formar un cmp de 5x5 con el menor primo con menor pares suficientes, 347.

397	379	271	241	757
157	277	331	619	661
691	751	**409**	67	127
739	199	487	541	79
61	439	547	577	421

Es posible formar un cmp de 5x5 con el menor primo con los pares justos, 409.

Para empezar a plantear un algoritmo generador de cuadrados primos debemos primeramente pues, realizar un estudio de los primos que pueden colocarse en el centro, de los candidatos.

La segunda parte para formar el cuadrado consiste en intentar colocar el central con sus pares en la disposición correcta. Nada nos asegura que aunque tengamos los pares suficientes, estos se puedan colocar correctamente, sólo mediante prueba y error podremos colocarlos, con el coste temporal que ello conlleva.

9.4 Estudio de los candidatos

Para el studio de los primos candidatos, imaginemos un cuadrado de lado muy alto, de 101x101. Para intentar encontrar un cuadrado de lado 101 necesitamos, al menos (101x101-1)/2 = 5100 pares. Como podemos imaginar, el orden de los primos que estamos tratando es muy grande.

Para hacernos una idea del orden del primo que necesitamos para la posición central habría que determinar un primo **C** tal que tenga 5100 primos (**p**) menores que él, relacionados con otros 5100 primos (**p'**) mayores que él (las dos partes de los pares) y que cuya distancia entre cada primo del par sea igual:

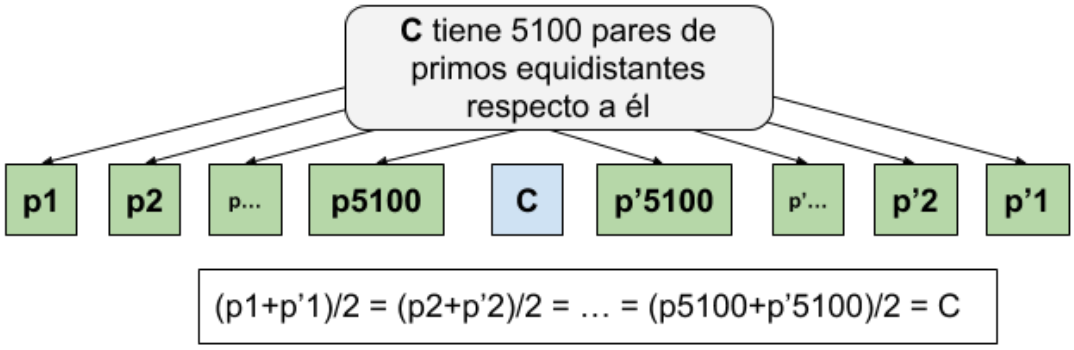

Ilustración 27 La fila central del 101x101

Para poder encontrar un primo candidato, primero debemos saber cuántos pares hay centrados en cada uno de los primos. Esta tarea la hemos dividido en dos partes:

1. Generar los primeros números primos de modo que tengamos una base suficiente de primos para trabajar durante todo el algoritmo. Llamaremos a este conjunto PrimosBase, *PB.*
2. Sea *Pmayor* el mayor primo de PB, podemos calcular todos los pares siempre y cuando su p' (primo mayor del par) sea menor o igual que *Pmayor*.

Por lo tanto sólo podremos calcular el número total de pares centrados en un primo para aquellos primos que son menores o iguales que *Pmayor*/2.

1) Generar los primeros números primos:

Existen muchos métodos para encontrar números primos. Sin embargo hay dos enfoques principales: las cribas y los algoritmos para discriminar primos.

Los primeros parten de todos los números enteros y a partir de cribas como puede ser la de Eratóstenes o versiones más optimizadas como la criba de Sundaram y la de Atkin ([5]), se consiguen filtrar aquellos números que son primos. Lo positivo de este tipo de implementaciones es que encuentran muchos primos de una sola barrida, que es precisamente lo que necesitamos. Lo negativo es que en términos de memoria son terriblemente ineficaces pues necesitan almacenar todos los números que se van a cribar (dependiendo de la criba es necesario tener todos los números o no, pero no vamos a entrar en ese detalle, ya que no es el problema que estamos solventando).

Por otra parte están los algoritmos para discriminar primos que funcionan a nivel individual. Sirven para determinar si un número es primo o no. Son más rápidos y

eficientes en términos de memoria que las cribas. Un algoritmo de este tipo puede ser, por ejemplo, aquel que realiza la descomposición en primos de un número y comprueba que el resultado es solamente el 1 y el propio número.

Existen aproximaciones que aceleran el proceso, como por ejemplo el test de Miller-Rabin ([6]) y son seguras para primos relativamente grandes. La parte negativa de este enfoque es que son temporalmente más ineficientes cuando se necesitan generar muchos primos, como es nuestro caso.

Sólo necesitamos generar *PB* una vez y guardarlo en memoria. Las ejecuciones de CCMSFP pueden reutilizar ese *PB* generado. Por este motivo vamos a utilizar la criba de Eratóstenes para generar *PB*, ya que su implementación es simple, las cribas se ajustan perfectamente para encontrar un rango grande de primos y el coste temporal va a ser mínimo, puesto que sólo va a ser necesaria una ejecución para calcular *PB* y después guardaremos el resultado en memoria.

Implementación de la criba de Eratóstenes:

El algoritmo de criba de Eratóstenes es sencillo y conocido. Los pasos en pseudocódigo son los siguientes:

Si queremos encontrar los números primos hasta *n*:

1. Crear una lista de los enteros consecutivamente a partir de 2 hasta *n*: (2, 3, 4, …, *n*).
2. Inicialmente, seleccionar un índice, al que denominamos *p*. Inicialmente tiene el valor 2, que es el primer número primo.
3. Comenzar desde *p*, marcando todos los múltiplos de *p* en la lista hasta que finalicemos con un valor mayor que *n*, (2*p*, 3*p*, 4*p*, …; **nota:** *p* no debe ser marcado, sólo los múltiplos de p).
4. Encontrar el siguiente número mayor que *p* que no esté marcado. Si no existe ese número, el algoritmo ha finalizado; si existe, tomar ese número como el siguiente índice *p* y repetir los pasos desde el punto 3.

Resultado: Al final del algoritmo, todos aquellos números que no han sido marcados en la lista de enteros son primos. Este conjunto lo denominamos *Cribados*.

El algoritmo se puede mejorar sensiblemente símplemente teniendo en cuenta que el valor máximo que debe tomar p en es $\sqrt{n}$. Es decir, para cada nuevo p todos los números menores que p ya han sido probados y por tanto tachados. El siguiente número que se podría llegar a tachar para cada p es p^2.

2) Implementación para calcular los pares centrados en cada uno de los primos:

Partimos de los primos que habíamos desarrollado en el apartado anterior. El algoritmo para marcar pares es muy simple:

Sólo podemos obtener los pares centrados para los primos **p** que cumplan:

$Mayor = \max\{Cribados\}$

$\boldsymbol{p} <= Mayor/2$

Comprobar para todos los primos **m**enores de p si existe un primo **s**uperior cuya distancia al elemento p sea la misma:

$\forall\, m : m<p \qquad \exists\, s: (m+s)/2==p \quad \rightarrow$ en caso afirmativo apuntar $[m, s]$ en el conjunto Pares para el primo p.

Resultado: Quedan apuntados todos los pares calculables para *Cribados* en Pares. Los pares se apuntan para cada uno de los cribados.

A tener en cuenta: dado el orden de los números primos que estamos manejando, tenemos que emplear un tipo lo suficientemente grande para poder representarlo. Para evitar problemas he decidido emplear **unsigned long long** en una estructura en C denominada **BigInt** para almacenar los primos (y correspondientemente, sus pares).

El tamaño de este tipo de datos es 8 bytes, pudiendo almacenar números tan grandes como 2^{64}.

Aunque a priori puede parecer que emplear tantos bits para representar la cifra es ineficiente, a la hora de realizar el algoritmo no va a ser muy relevante porque en casi todos los cálculos se van a emplear los índices o posiciones de memoria en lugar de los propios valores en sí.

Al final de este algoritmo tenemos almacenados los primos y sus pares en un fichero binario denominado **primosConPares.dat** que mantiene la siguiente información:

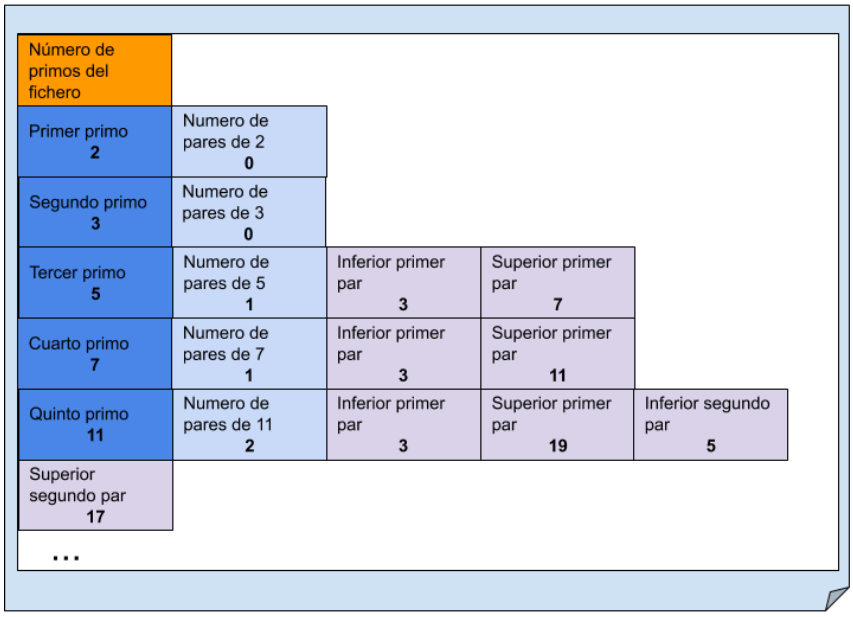

Ilustración 28 Estructura del fichero PrimosConPares.dat

Dado el coste espacial tan alto que tiene almacenar cada número del gráfico anterior (8 bytes) he creado varios ficheros con los primos (menores de 1000, menores de 10000…), de tal manera que no hace falta cargar y manejar un fichero que tiene los primos hasta 1000000 si vamos a resolver un cuadrado de 15x15, por ejemplo.

9.5 Un ejemplo para plantear el algoritmo, anidación para crear el 3x3

Para componer un cuadrado mágico compuesto por primos. Vamos a empezar explicando el algoritmo utilizando un ejemplo base: componer un cuadrado mágico de primos de 3x3.

Lo primero que vamos a determinar es el primo central que podemos utilizar como candidato. Teniendo el orden del cuadrado, el número de pares necesarios es 4.

Siendo *n* el orden del cuadrado, necesitamos los siguientes pares: *$((n^2)-1)/2$. En el caso de un cuadrado de 3x3 → $(3^2-1)/2 = 4$ pares.*

Podemos encontrar un candidato consultando la lista de primos con pares, que es la siguiente:

2: 0 pares 3: 0 pares 5: 1 pares 7: 1 pares 11: 2 pares 13: 2 pares 17: 3 pares 19: 1 pares 23: 3 pares 29: 3 pares 31: 2 pares **37: 4 pares** **41: 4 pares**	**43: 4 pares** **47: 4 pares** **53: 5 pares** **59: 5 pares** 61: 3 pares **67: 5 pares** **71: 7 pares** **73: 5 pares** **79: 4 pares** **83: 5 pares** **89: 6 pares** **97: 6 pares** [...]

Ilustración 29 Primos con el número de pares centrados en cada uno

El primer número candidato para ser colocado en el centro del cuadrado es 37 ya que tiene 4 pares.

Sabemos que si tomamos el 37 como centro del cuadrado, la constante mágica va a ser 3x37 = 111

Los pares de 37 son los siguientes:

37: 4 pares →

{31 - 43}
{13 - 61}
{7 - 67}
{3 - 71}

Una segunda comprobación que podemos realizar es verificar que haya n (siendo n el orden del cuadrado) primos entre sus pares (sin repetir un par) cuya suma sea igual a la constante mágica (111).

P_1	P_2	P_3
P'_4	37	P_4
P'_3	P'_2	P'_1

Ilustración 30 Pares de la primera fila del ejemplo de 7x7

P1 + P2 + P3 = 111

Las posibles combinaciones que sumen 111 son las siguientes:

31+13+67 = 111 → (Como consecuencia, es trivial) → 43+61+7 = 111

37: 4 pares

{**31** - 43}
{**13** - 61}
{7 - **67**}
{3 - 71}

Por simetría, cuando una combinación suma la constante mágica, entonces la suma de los pares complementarios también suma la constante mágica. Gracias a este hecho podemos dividir la ejecución del algoritmo a la mitad (sólo comprobamos una mitad de las posibilidades).

Podemos colocar los valores encontrados en la primera fila con todas las posibles ordenaciones posibles siempre y cuando los pares asociados los cambiemos en la fila inferior:

Cuadrado 1:

31	13	67
P'_4	37	P_4
7	61	43

Cuadrado 2:

13	31	67
P'_4	37	P_4
7	43	61

Cuadrado 3:

13	67	31
P'_4	37	P_4
43	7	61

No hay más combinaciones, las demás combinaciones posibles ya son simétricas de la mencionada anteriormente.

Si tenemos el **cuadrado1**, podemos colocar los pares restantes (sólo un par, el {3-71}) de las siguientes maneras. **Sólo es necesario fijarse en la primera columna**:

31	13	67
71	37	3
7	61	43

Por primera columna → 31+71+7 = 109, no es un cuadrado mágico, no suma 111.

31	13	67
3	37	71
7	61	43

Por primera columna → 31+3+7 = 41, no es un cuadrado mágico, no suma 111.

Si tenemos el **cuadrado2**, podemos colocar los pares restantes (sólo un par, el {3-71}) de las siguientes maneras. **Sólo es necesario fijarse en la primera columna**:

13	31	67
71	37	3
7	43	61

Por primera columna → 13+71+7 = 91, no es un cuadrado mágico, no suma 111.

13	31	67
3	37	71
7	43	61

Por primera columna → 13+3+7 = 23, no es un cuadrado mágico, no suma 111.

Si tenemos el **tercer cuadrado**, podemos colocar los pares restantes (sólo un par, el {3-71}) de las siguientes maneras. **Sólo es necesario fijarse en la primera columna**:

13	67	31
3	37	71
43	7	61

Por primera columna → 13+3+43 = 59, no es un cuadrado mágico, no suma 111.

13	67	31
71	37	3
43	7	61

Por primera columna → 13+3+43 = 127, no es un cuadrado mágico, no suma 111.

En el ejemplo dado, tras probar todas las combinaciones, podemos determinar que utilizando los 3 primos (13,67,31) que suman 111, no se puede generar un cuadrado mágico. O lo que es lo mismo, no se puede anidar, si entendemos que estamos anidando sobre el cuadrado de 1x1 que conforma el primo 37.

Partiendo de este ejemplo simple, hemos conformado la primera aproximación al algoritmo que nos permite resolver cuadrado mágicos anidados.

Se destacan las siguientes particularidades que serán clave a la hora de elaborar un algoritmo:

- El primo que se coloca en el centro restringe el orden máximo del cuadrado que se puede encontrar. A este apartado lo hemos determinado **búsqueda de candidatos para un cuadrado de lado n**.
- Sabiendo el valor de la constante mágica, **se puede ir construyendo el cuadrado por partes, primero colocando la fila superior, luego un lateral e ir anidando capas**. Por distancia al centro, los pares quedan automáticamente colocados, sólo hay que colocar la mitad de todos los huecos disponibles por cada anidación.

10 ALTERNATIVAS PARA LA IMPLEMENTACIÓN DEL ESTUDIO GENERATIVO

Estas premisas han servido para generar una aproximación sobre la generación de cuadrados primos empleando anidación. A continuación, en el punto 11 y en el punto 12 de este documento vamos a ver dos distintas técnicas de aplicar este algoritmo para la generación de soluciones.

Los planteamientos han surgido según nos hemos ido encontrando con casuísticas al realizar pruebas y al mejorar el algoritmo. A grandes rasgos estas son sus características:

El primer planteamiento, el algoritmo secuencial recursivo permite generar cuadrados de cualquier orden. Sus características más positivas son:

1. Se adapta al tamaño del cuadrado a generar, el mismo algoritmo se emplea para generar un orden cualquiera. La solución es elegante y simple gracias a la recursividad y el backtracking (al menos comparándolo con la segunda variante).
2. Encuentra soluciones de órdenes de cuadrados muy grandes, hasta de 101x101.

Pero tiene dos inconvenientes:

1. No es paralelizable (al menos no mediante OpenMP) por culpa de la recursividad. Además, es muy difícil de optimizar por ese mismo hecho.
2. Si hay muchos más pares de los suficientes, encuentra una solución muy rápida, incluso para cuadrados de un tamaño grande. Pero tiene problemas para determinar si NO se puede hacer un cuadrado mágico con una entrada de pares dada, ya que es difícilmente optimizable y el número de combinaciones no es computable en un tiempo razonable.

El segundo planteamiento, el algoritmo paralelo con bucles *for*, surge para resolver las limitaciones que tiene el primer planteamiento. Las principales mejoras son:

1. Se puede determinar si no existe una solución con los primos que se emplean como entrada del algoritmo.
2. Es paralelizable, por lo que el código se ejecuta mucho más rápido.

3. Es optimizable. Se recortan muchas posibilidades mediante simetría, descarte de posibilidades y almacenamiento de posibilidades ya exploradas.

Los inconvenientes que presenta:

1. Excesiva complejidad. El algoritmo se conforma mediante una secuencia de bucles *for* para cada búsqueda, en caso de que avancemos en la solución, pasamos a bucles interiores con los pares que nos quedan disponibles.
2. No se pueden optimizar todas las ideas que se plantean. O bien porque el mero hecho de comprobar la optimización consume más recursos que el ahorro que conlleva, o bien porque no se puede optimizar dos casuísticas a la vez. Esto lo explicaremos con más detenimiento en el punto 9.
3. No es adaptable a distintos tamaños. Sólamente soluciona cuadrados de 7x7.
4. Solamente funciona con el número de primos mínimo, es decir con 24 pares.

El segundo planteamiento, surge como una visión con aun más restricciones (Ya habíamos delimitado cuando aplicamos como principio la anidación). Con esto lo que se busca es reducir el amplio espacio de búsqueda para confirmar si un cuadrado mágico NO tiene solución. Es decir, si con los 24 pares seleccionados, realmente se puede o no generar un cuadrado mágico de 7x7.

Veamos con más detalle cada uno de estos planteamientos.

11 ALGORITMO 1: GENERADOR RECURSIVO MULTIDIMENSIONAL

Esta primera fase del algoritmo consiste en solucionar un cuadrado mágico de primos anidados teniendo exclusivamente como entrada el tamaño deseado del cuadrado mágico. El planteamiento de dicho algoritmo viene representado por el siguiente esquema en sus rasgos más significativos:

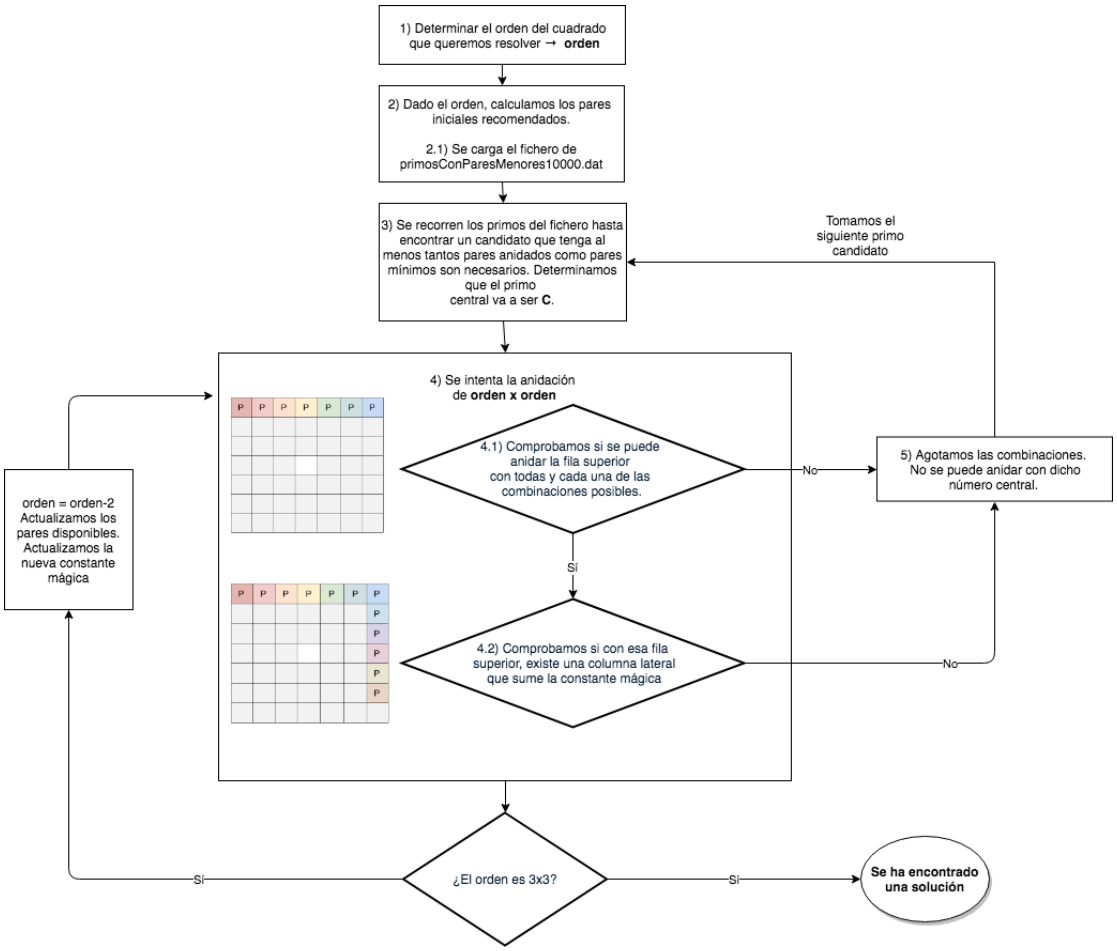

Ilustración 31 Esquema de funcionamiento del algoritmo recursivo

Uno de los problemas que nos plantea esta aproximación es cómo podemos codificar la solución para que funcione independientemente del número de veces que se anide (ya que el tamaño es variable). Además, dentro de una misma ejecución hay que ir cambiando el orden seguń se van añadiendo los distintos anillos del cuadrado.

La anidación para encontrar la solución se va a llevar a cabo desde el orden mayor del cuadrado hasta la última anidación de 3x3. Empezar desde órdenes mayores a menores se debe a las siguientes razones:

- **Hay menos combinaciones válidas para un orden alto que para un orden bajo**. Para un número de pares cualesquiera, por regla general hay muchas más combinaciones de anidaciones de orden inferior que de orden superior, pero sólo habrá solución si efectivamente existe solución en el orden superior y no viceversa.
- **No empleamos primos en una anidación inferior que son necesarios para una anidación superior**. Imaginemos una solución completa de un cuadrado mágico. Es muy posible que con los primos del orden superior seamos capaces de componer anidaciones inferiores, por ejemplo, con los pares que componemos la anidación 13x13 podríamos componer la 3x3. Pero hemos desperdiciado esos pares, ya no podríamos encontrar la anidación 13x13 sin hacer backtracking. Esto es especialmente importante para esta primera aproximación, que funciona como un algoritmo para encontrar soluciones en lugar de como un algoritmo para **descartar** soluciones.

Vamos a ir explicando con detenimiento cada uno de los puntos que aparecen en el esquema anterior:

1) Determinar el orden del cuadrado que queremos resolver. Este bloque funciona como preparación del algoritmo**.** Indicamos el orden del cuadrado mágico que queremos encontrar mediante la siguiente constante:

```
const int ordenNuevo = 7; //Lo cambiamos por el compilador de linux.
```

Decidimos qué fichero de los ParesConPrimosMenoresQue… vamos a utilizar. El fichero con primos menores que 100000 nos ofrece valores suficientes para todas las pruebas iniciales con cuadrados con orden <=17.

```
char * FICHERO_PARES = "primosConParesMenores";
BigInt  RANGO_CRIBA = 100000; //Los primos que se cargan, ya cribados
BigInt RANGO_CANDIDATOS = 20; //Los primeros candidatos que se van a comprobar.
```

RANGO_CANDIDATOS es una constante que determina cuántos candidatos se van a tomar como primos centrales. Es decir, cuántas veces como máximo podría ejecutarse el algoritmo con distintos centros pasando por el punto 5 del esquema.

2) Dado el orden, calculamos los pares iniciales recomendados. Sabemos que los pares mínimos para generar un cuadrado mágico de orden n es $(n^2 -1)/2$. Hasta ahora no hemos tenido en cuenta el número de **pares iniciales recomendados**, que es distinto de los pares necesarios. Hemos acotado el número de pares necesarios inferiormente mediante el orden del cuadrado. Pero si tenemos más pares que los mínimos necesarios, obviamente va a ser más fácil encontrar un cuadrado formado por primos, ya que vamos a tener más números para poder combinar.

```
BigInt paresRecomendados = (ordenNuevo * ordenNuevo) * 2;
```

Los pares recomendados son más del cuádruple de los mínimos necesitados. Este valor ofrece unos resultados buenos ya que casi nos garantiza encontrar el cuadrado sin que entremos en backtracking. Si queremos calcular cuadrados muy grandes (orden >50) puede que se quede pequeño, podemos ir modificando esta cantidad según lo deseemos.

Cargar el fichero de primos necesitamos la siguiente estructura:

```
//Una estructura que guarda un primo y sus pares asociados.
typedef struct _S_primoConPares{
    BigInt primo;
    BigInt numPares;
    BigInt ** pares;
}PrimoConPares;
```

Esta estructura almacena los siguientes valores:

- primo → Un primo que puede ser colocado en el centro del cuadrado.
- numPares → Cuántos pares hay centrados en este primo. Fundamental para descartar rápidamente un número como central del cuadrado.
- pares → Una estructura bidimensional que almacena los pares centrados en *primo*. En el primer índice de la estructura se almacena la posición dentro del par. Si es un cero significa que es el primer primo. En la segunda dimensión se almacena el valor del primo par.

Inicialmente cargamos todos los primos junto con sus pares como un array de *PrimoConPares* denominado primosCandidatos.

```
//Posición de memoria desde donde se almacenan todos
//los primos con sus pares que son candidatos como centro del cuadrado.
extern PrimoConPares* primosCandidatos;
```

Se carga con la siguiente llamada (recortado, el método continúa):

```
//Método para rellenar los candidatos.
//Recorre el fichero de primos con pares y nos devuelve los RANGO_CANDIDATOS primeros.
void rellenarCandidatos(){

    primosCandidatos = malloc(sizeof(PrimoConPares)*RANGO_CANDIDATOS);
```

El método rellenarCandidatos, carga sólamente *PrimoConPares* que cumplan que tienen centrados el número necesario de pares que indique *paresRecomendados*.

3) Se recorren los primos del fichero hasta encontrar un candidato que tenga al menos tantos pares anidados como pares iniciales son recomendados. Una vez tenemos la estructura de *primosCandidatos* en memoria, consiste en indicar cuál de ellos vamos a emplear, partimos desde la primera posición de la estructura y cada vez que pasemos por el bloque 5 del esquema de ejecución vamos avanzando una posición.

```
//Guarda el índice de primosCandidatos con el que se trabaja
extern int indiceCandidato;
```

4) Se intenta la anidación de orden x orden. El grueso del algoritmo. Consiste en colocar una anidación empezando por la fila superior y pasando por la columna derecha. Por simetría de pares se coloca la fila inferior y la columna izquierda.

11.1 Estructuras de datos

```
extern PrimoConPares primoRecursivo;
```

-*primoRecursivo* almacena el primo central que se está empleando actualmente, el número de pares que tiene centrados y un array bidimensional que guarda los pares de primos. Tanto los datos del número de pares como los del array bidimensional de pares se actualizan con cada anidación.

```
extern int ordenEstaAnidacion; //El orden de la anidación actual que se está resolviendo.
```

- *ordenEstaAnidacion* va almacenando el orden de la anidación actual. Inicialmente es el mismo que el cuadrado mágico que queremos generar y va reduciéndose de 2 en 2 hasta que llegamos al orden 3 y hemos encontrado una solución completa.

```
extern int * indicesRecursivo;
```

- indicesRecursivo es un array que almacena los índices de los pares que se van a emplear para intentar anidar la fila superior.

```
extern int * indicesFilaRecursivo;
```

- *indicesFilaRecursivo* es un array que almacena, para cada uno de los primos de la fila superior, si se trata del primero del par o del segundo del par.

```
extern int * indicesNoUtilizadosRecursivo;
```

- *indicesNoUtilizadosRecursivo* es un array que almacena los índices de los pares que no han sido utilizados para la fila superior.

```
extern int * indicesLateralRecursivo;
```

- *indicesLateralRecursivo* es un array que almacena los índices de los pares que se seleccionan para la columna del lateral. El entero que almacena es la posición de *indicesNoUtilizadosRecursivo* del par.

```
extern int * indicesFilaLateralRecursivo;
```

- *indicesFilaLateralRecursivo* es un array que almacena para cada par seleccionado en el lateral si es el primero del par (almacenaría un 0 en dicha posición) o si es el segundo (almacenaría un 1).

```
BigInt ** solucion;
```

- *solucion* es una matriz bidimensional que se emplea para guardar la solución. Sirve como almacenamiento temporal del estado del cuadrado hasta que se encuentra la solución completa.

11.2 Métodos y funciones

- **procesarCombinaciones:**

```
void procesarCombinaciones(int numeroIndice, int porCualAndo){...}
```

Es la llamada principal para lograr una solución del algoritmo. Permite seleccionar combinaciones de pares tomadas de *ordenEstaAnidacion* en *ordenEstaAnidación* sin repetir. El primer parámetro indica el número de pares que ya hemos seleccionado para una combinación y el segundo indica la posición del último par que hemos utilizado. Los pares que se han utilizado quedan almacenados en la estructura *indicesRecursivo*.

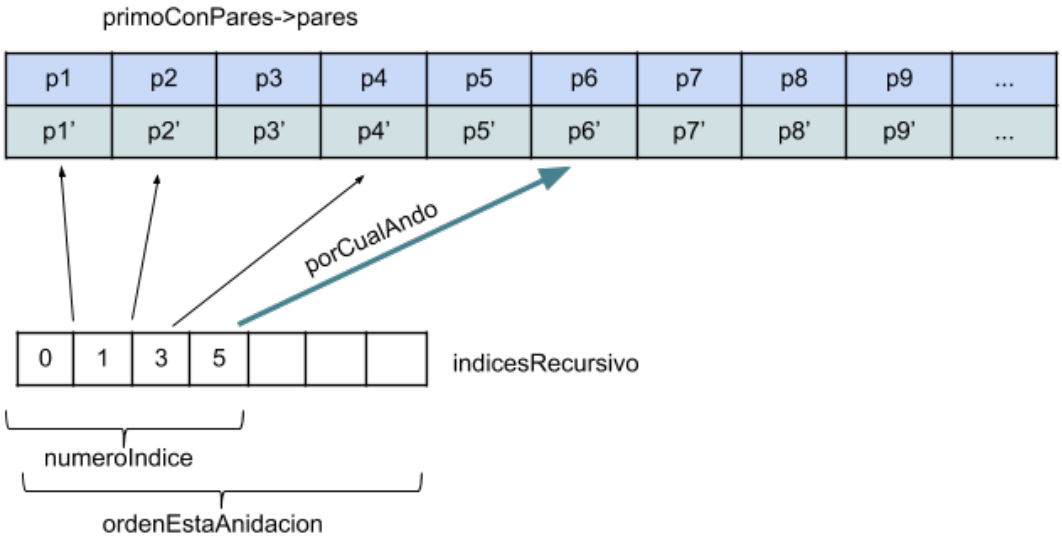

Ilustración 32 Esquema de funcionamiento de procesarCombinaciones

Los siguientes valores de indicesRecursivo siempre se van a obtener desde las posiciones siguientes a *porCualAndo*. De esta manera evitamos repeticiones por distinto orden (el orden no importa, lo único que importa es que la suma sea la constante mágica). Cuando el *numeroIndice* es igual al *ordenEstaAnidacion*, entonces podemos procesar si la fila es candidata y con ella podemos seguir avanzando mediante la función *procesarCombinacionesRecursiva*.

- ***procesarCombinacionesRecursiva**:*

```
//Procesa todas las posibles combinaciones de primos una vez seleccionados
// los pares para colocar la fila superior.
void procesarCombinacionesRecursiva(int profundidad){...}
```

Por cada uno de las combinaciones de pares seleccionadas, debemos comprobar todas las combinaciones posibles de primos mayor y menor de cada uno de ellos. Esta

función recursiva se encarga de ello. Se llama sobre sí misma, eligiendo por cada par sus dos posibles primos, el mayor y el menor y avanzando en el índice de selección *profundidad* en cada llamada.

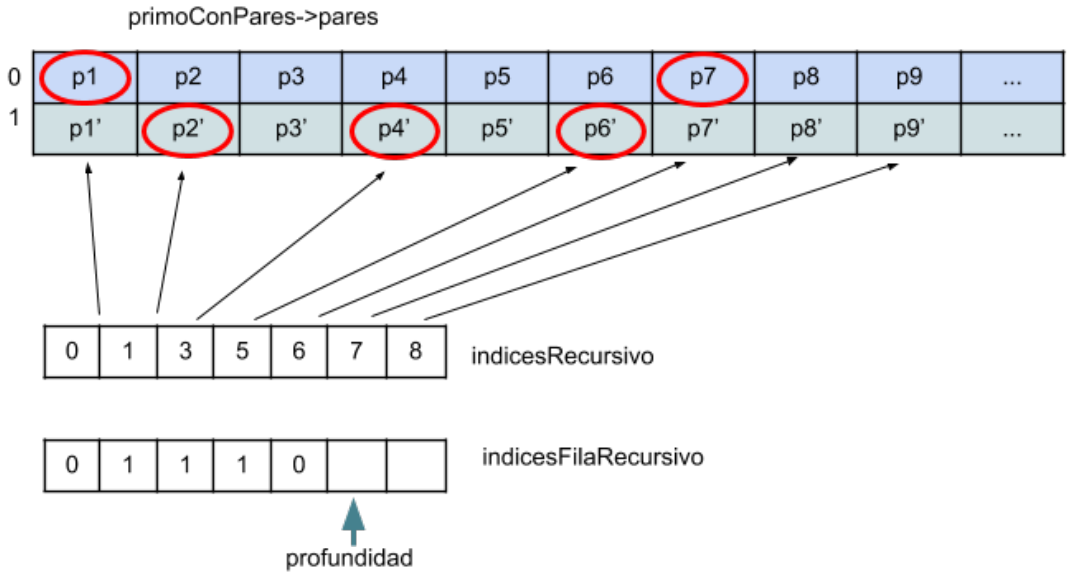

Ilustración 33 Esquema de funcionamiento de procesarCombinacionesRecursiva

Una vez hemos seleccionado una combinación completa para la fila superior (pares y primos dentro de los pares) podemos comprobar si la fila es candidata mediante la función *comprobarFilaCandidata*.

- ***comprobarFilaCandidata***:

```
//Comprueba si la fila superior de la anidación es válida.
int comprobarFilaCandidata(){...}
```

Devuelve verdadero si los pares definidos por *indicesRecursivo* y *indicesFilaRecursivo* suman la constante mágica. Si es así, continuamos con el algoritmo con la función *procesarLaterales*.

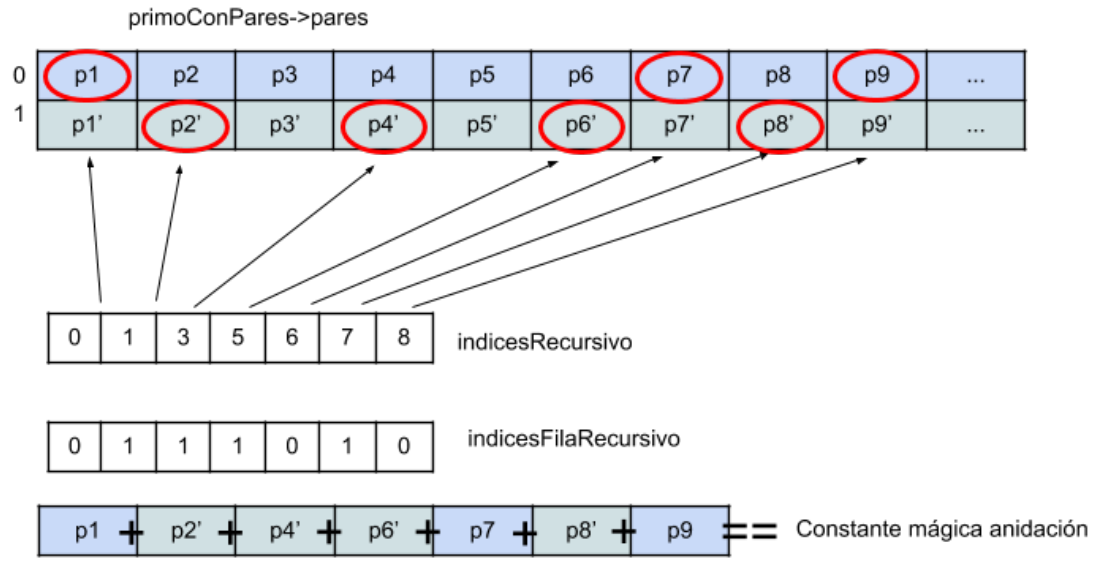

Ilustración 34Esquema de funcionamiento de comprobarFilaCandidata

- ***procesarLaterales***

```
//Método que se encarga de inicializar la resolución de la parte derecha de una anidación (lateral).
//Se utiliza una vez hemos encontrado la fila superior.
void procesarLaterales(){
```

Prepara las estructuras necesarias para buscar una combinación para el lateral. Realiza dos subtareas distintas:

1. Rellena el array *indicesNoUtilizadosRecursivo* con los índices que no se han utilizado. Nos servirá como estructura auxiliar para poder ir eligiendo los índices de los pares restantes que quedan sin utilizar. Ojo, que para llegar al índice de un par ahora tendremos que pasar por esta estructura que indexa los libres.

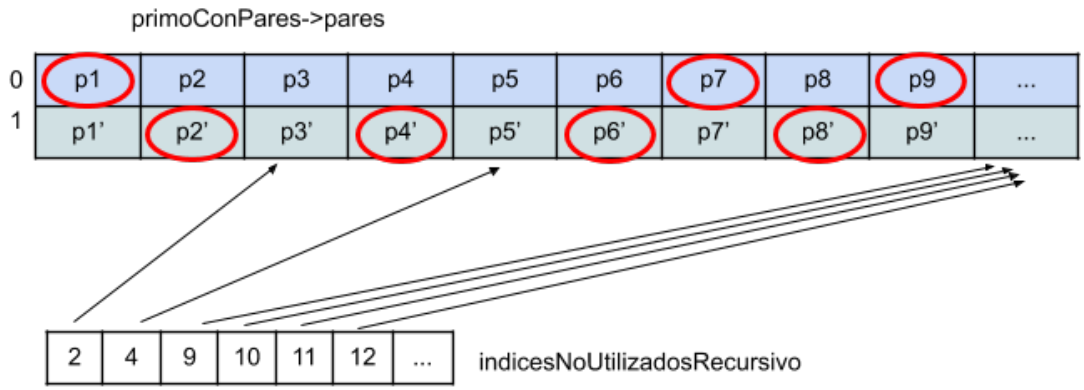

Ilustración 35 Detalle de indicesNoUtilizadosRecursivo

2. Tomamos todas las combinaciones posibles de esquinas con los pares empleados en la fila superior. Para ello hay que coger todas las combinaciones posibles de pares tomados de 2 en 2 sin repetición y sin orden. No nos importa las posiciones intermedias de la fila superior, ya que no condicionan el anidamiento. Sólo nos interesan las esquinas porque condicionan la fila lateral:

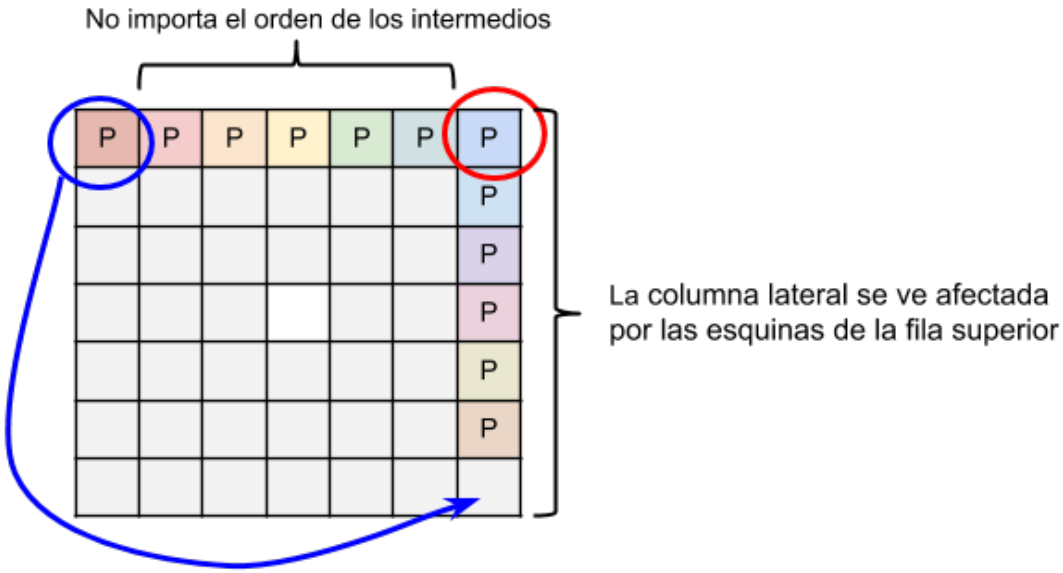

Ilustración 36 Los intermedios de la fila superior no condicionan la columna

Para cada fila superior que suma la constante mágica, para cada posible posicionamiento de esquinas (da igual el orden), intentamos colocar la fila derecha mediante la función recursiva *combinacionesRecursivaLateral*.

- ***combinacionesRecursivaLateral***

```
//Selecciona todas las posibles combinaciones de pares restantes (sin los usados en la fila)
void combinacionesRecursivaLateral(int numeroIndice, int porCualAndo){...}
```

Funciona de la misma manera que *procesarCombinaciones* pero empleando como estructura para ir seleccionando los pares *indiceLateralRecursivo* que hace referencia al índice de *indicesNoUtilizadoRecursivo*. El siguiente gráfico muestra la relación de índices:

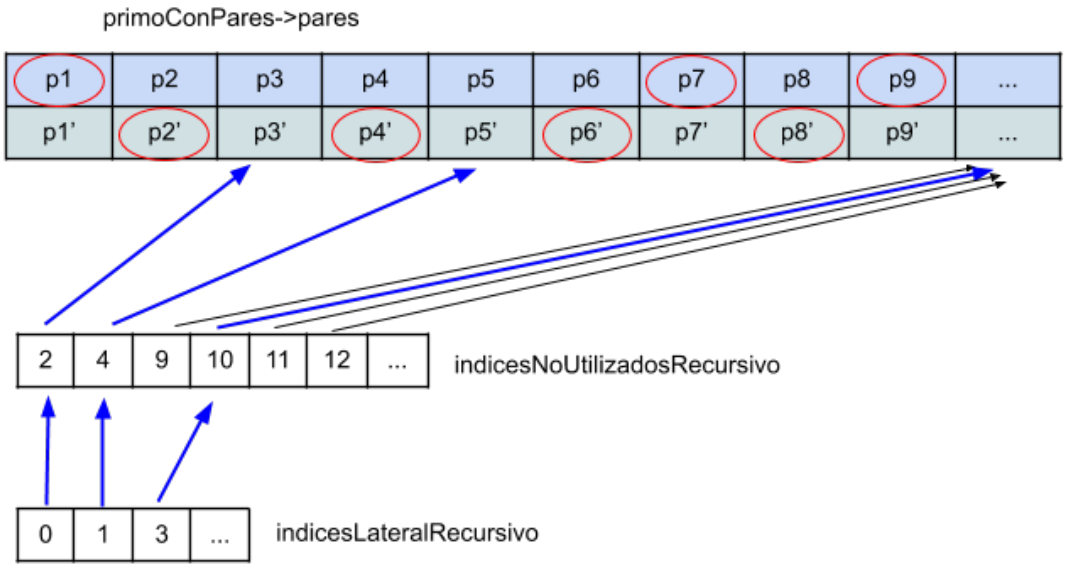

Ilustración 37 Relación entre índices

indicesLateralRecursivo toma todas las combinaciones posibles de *indicesNoUtilizadosRecursivo* tomadas de orden -2 en orden-2 (los dos que restamos son las esquinas de la fila superior que hemos prefijado anteriormente). Cada vez que se rellene llamamos a *procesarCombinacionesRecursivaLateralFila.*

- ***procesarCombinacionesRecursivaLateralFila***

```
//Selecciona todas las combinaciones de primos con los pares seleccionados para la columna lateral.
void procesarCombinacionesRecursivaLateralFila(int profundidad){...}
```

Es capaz de generar todas las combinaciones de primos para los pares del lateral, sin repetir primos del mismo par, y sin que importe el orden de los primos seleccionados. Si es el primero o segundo del par queda apuntado en la estructura *indicesFilaLateralRecursivo*.

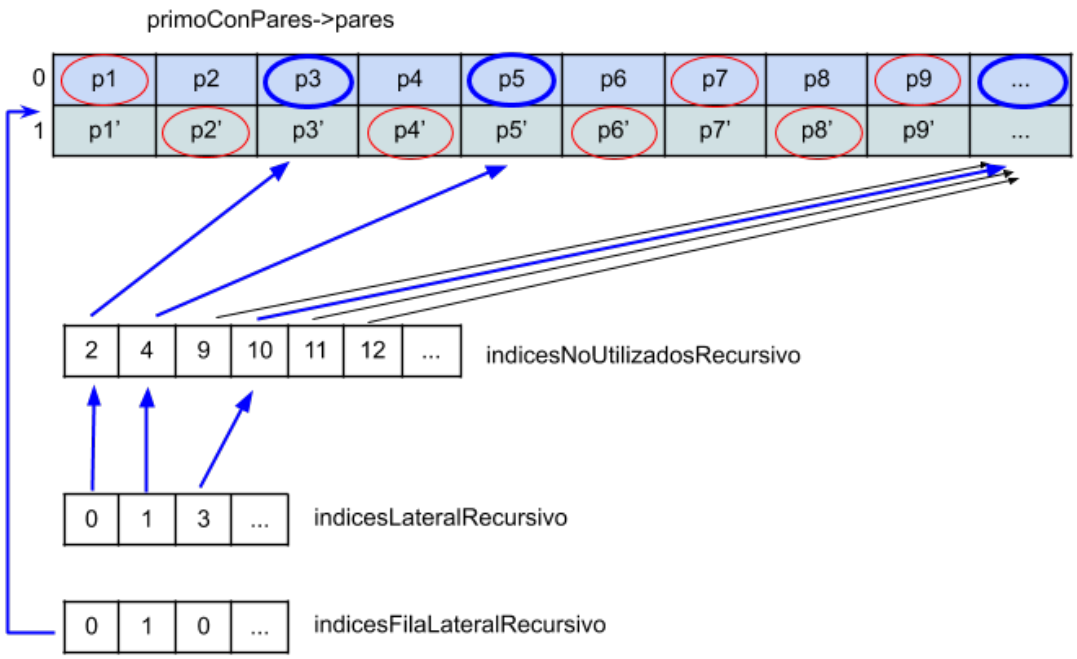

Ilustración 38 Índices empleados para la anidación lateral

Para todas las combinaciones generadas se llama a *comprobarLateralCandidato para determinar si la combinación es buena* y en caso de que sea afirmativo el resultado, proceder con su resolución mediante *resolverAnidacion*.

- ***resolverAnidacion***

```
//Se ha encontrado una anidación válida. Hay que preparar para anidar de nuevo o para dar la solución
void resolverAnidacion(){...}
```

Llegamos a este método cuando hemos encontrado una fila superior y una columna lateral formada por pares, eso quiere decir que por simetría hemos encontrado un conjunto de números que permiten anidar una solución del problema.

Empleamos el método *guardarAnidacion* para almacenar el anillo nuevo sobre la matriz que guarda la solución, denominada *solucion*. No importa el orden de los primos en la fila superior y la columna que no sean esquinas, podríamos intercambiar las posiciones de todos ellos, por lo que se van rellenando por el orden que ocupan los índices en las estructuras que los albergan.

Pueden ocurrir dos cosas:

1. Que sea la anidación de 3x3, con lo cual ya tenemos el cuadrado mágico formado por primos completo. En dicho caso. Podemos dar por concluido el algoritmo. Paramos el tiempo de ejecución, pintamos por pantalla los resultados y liberamos los recursos necesarios para la ejecución del programa.

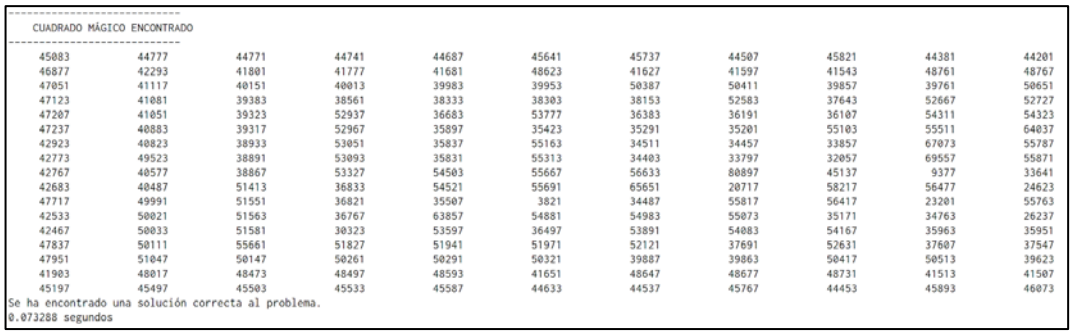

Ilustración 39 Solución del cuadrado 11x11

2. No hemos completado el cuadrado, debemos seguir anidando. Hay que realizar los siguientes pasos antes de continuar:

    1. Empleamos la pila (stack) de la llamada a este método para almacenar punteros nuevos a las estructuras que hemos empleado en esta anidación. Guardamos copias que pueden ser restauradas si no encontramos soluciones en las anidaciones interiores y tenemos que volver a buscar por backtracking.
    2. También aprovechamos para calcular el nuevo orden para la siguiente anidación, que es el mismo que el anterior menos dos.
    3. Volvemos a reservar memoria para las estructuras empleadas.
    4. Actualizamos la estructura de *primoRecursivo* con el número de pares que quedan disponibles y los que todavía no se han utilizado. También guardamos una referencia de los valores anteriores en pila por si fuese necesario restaurarlo por backtracking.
    5. **Hacer una llamada de nuevo a *procesarCombinaciones***. Volvemos al punto 4 del esquema de ejecución del algoritmo, pero con menos pares y un orden menor.
    6. **Backtracking**. Si no se encuentra solución en la llamada anterior a *procesarCombinaciones*, debemos restaurar los valores anteriores que habíamos guardado y también eliminar la memoria auxiliar que reservamos, que ya no utilizamos. Es muy importante que no haya pérdidas de memoria, porque podríamos agotar toda la memoria disponible tras unas ejecuciones de backtracking y hacer que el algoritmo dejase de funcionar inesperadamente.

11.3 Estructuración del proyecto

A continuación, se hace un breve repaso de los ficheros que componen el proyecto.

Librería globales.h/c: Almacena todos los valores que se pueden modificar para alterar la ejecución de la resolución. A destacar los dos valores *ordenNuevo* y *paresRecomendados* que tendremos que ir modificando para definir qué cuadrado queremos encontrar y cuántos pares vamos a necesitar.

Librería cargaGuarda.h/c: Carga la estructura de primos candidatos y permite visualizar los primos candidatos si quisiéramos depurar o mostrar información al usuario.

Librería solucion.h/c: Define la estructura matricial para guardar la solución y tiene los métodos necesarios para manejarla: Guardar una anidación, Reservar y liberar memoria, pintarla por pantalla…

Librería estructuras.h: Define las estructuras de datos empleadas para la ejecución del algoritmo, especialmente aquellas que almacenan los índices de los pares empleados en la anidación.

Librería funciones.h/c: Contiene funciones relacionadas con:

- Gestión de memoria. Reservar y liberar la memoria necesaria para cada ejecución del algoritmo.
- Tiempo de ejecución. Inicializar la cuenta de tiempo, pausar la cuenta de tiempo y mostrarlo por pantalla.
- Algunas funciones auxiliares generales. Como por ejemplo comprobar si un elemento está en un array.

Main.c: Es el código ejecutable principal. Las funciones descritas dentro del punto 4 del esquema del algoritmo se encuentran en este fichero. Todas las llamadas recursivas, la inicialización de valores y la muestra de resultados se engloban aquí.

CMakeLists.txt: Para la gestión del build del proyecto se emplea CMake, que facilita la tarea de compilación en distintos entornos distintos. En mi caso he estado compilando el proyecto tanto para Linux como para Mac. Las dependencias son librerías propias del proyecto por lo que el build es muy sencillo.

11.4 Posibilidades de paralelización y optimización.

El planteamiento del algoritmo emplea las mismas estructuras de índices una y otra vez para cada anidación. Precisamente emplear las mismas estructuras es lo que nos habilita la posibilidad de volver a llamar a los mismos métodos recursivamente para ir seleccionando todas las combinaciones. Desafortunadamente, esta misma estructura recursiva que hace que el algoritmo pueda adaptarse a cualquier orden hace imposible, o poco eficiente la paralelización, ya que, al emplear la misma memoria, no se pueden lanzar tareas amplias que se ejecuten a la vez en distintos cores y que realmente hagan valor del coste que implica comenzar a paralelizar.

En cuanto a la optimización, podríamos hacer un descarte temprano al ir seleccionando combinaciones de pares si la suma parcial de estos pares nos arrojase valores que sobrepasan la constante mágica o que bien son tan bajos que ya no se puede alcanzar la constante mágica. Si recordamos cómo funciona el algoritmo, primero genera todas las combinaciones de índices de los pares posibles y después genera qué primos se seleccionan dentro de los pares. Hasta este segundo paso, no tenemos valores con los que ir realizando optimizaciones parciales, ya es demasiado tarde, pues todas las posibles combinaciones de índices de pares ya están hechas. Este es uno de los puntos clave que se han trabajado en el segundo algoritmo.

11.5 Resultados

La siguiente gráfica (Ilustración 40) muestra la relación entre el tiempo de procesamiento y el orden del cuadrado que se quiere completar. Todas las ejecuciones se han llevado a cabo en el mismo equipo, un AMD Ryzen 5 2600.

El tiempo crece con el orden de una forma exponencial. Pese a ello, el algoritmo es muy rápido, la medida de tiempo empleada es ms.

Tampoco podemos obviar los problemas que surgen al almacenar y calcular primos tan grandes. Para más de orden 19 es necesario haber cribado previamente más de los 100000 primeros primos.

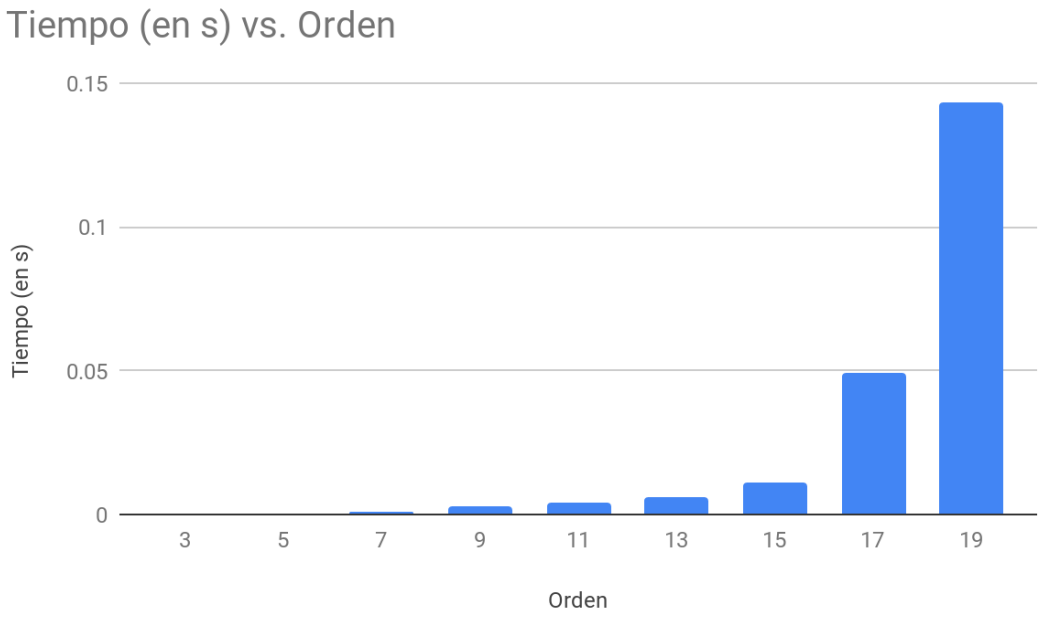

Ilustración 40 Rendimiento del algoritmo 1 Tiempo vs Orden

11.6 Conclusiones

Este planteamiento está totalmente enfocado a encontrar un cuadrado de un orden dado. Cuantos más pares disponibles haya más sencillo será encontrar una solución. Al final encontrar un cuadrado se resume a encajar los diferentes pares posibles según van apareciendo, si tenemos muchos pares disponibles no nos preocupamos de tener que probar distintas combinaciones y tener que deshacer parte de la solución para encontrar otros caminos, simplemente, seguimos probando, porque habrá algún primo que encaje.

Este algoritmo se ha construido para encontrar cuadrados mágicos formados con primos, pero no es paralelizable (código recursivo). Si quisiéramos acelerar el proceso de encontrar el cuadrado, la mejor estrategia es añadir más pares, para que evitemos totalmente emplear el backtracking. Además es muy lento a la hora de determinar si con un conjunto de primos dados centrados en otro primo NO se puede construir un cuadrado mágico. Todo el proceso de backtracking, si bien está implementado, no está optimizado, hay que realizar muchas operaciones de restauración de memoria muy costosas cada vez que queremos probar una nueva combinación, algo que va a a ocurrir muy frecuentemente una vez empezamos a realizar backtracking.

Aun así, gran esfuerzo dentro del desarrollo total del proyecto fin de máster ha sido destinado a construir este algoritmo, que efectivamente devuelve una solución correcta para cuadrados muy grandes. En el siguiente punto de la documentación describiremos cómo lo hemos transformado para satisfacer los requisitos de paralelismo del planteamiento del proyecto fin de máster.

12 ALGORITMO 2: GENERADOR SECUENCIAL DE CUADRADOS 7X7

Este algoritmo surge para solucionar las complicaciones encontradas con el planteamiento anterior. El esquema de ejecución es muy parecido al anterior, consiste en ir anidando capas desde los órdenes superiores a los inferiores mediante la búsqueda de fila superior y columnas a la derecha. En la siguiente lista se enumeran, de mayor a menor importancia las características de este nuevo planteamiento:

- El algoritmo es secuencial y no recursivo. De esta manera podremos paralelizar completamente la ejecución del algoritmo, cumpliendo con los requisitos del proyecto y acelerando la ejecución.
- El algoritmo es capaz de determinar si no existe solución dado un conjunto de pares dados en un tiempo de ejecución razonable (menos de 12 horas en el clúster de computación).
- Gracias a la nueva organización del código, es posible hacer optimizaciones parciales que mejoran el rendimiento del algoritmo. Las optimizaciones consisten en descartar posibilidades en el momento en el que ya no puedan formar un cuadrado mágico, en lugar de generar todas las combinaciones de filas y columnas.

Por contra, también nos encontramos algunos inconvenientes que se numeran a continuación:

- El código del algoritmo es secuencial, es muchísimo más complejo y difícil de mantener que el del algoritmo anterior. Para la solución del cuadrado existen más de 45 *bucles for* anidados. No un código fácil de entender, de mantener o de modificar, pero es el precio que pagamos por poder optimizar el algoritmo. En el anterior planteamiento teníamos un código muy elegante que se autoajusta al tamaño del algoritmo, en este caso tenemos un código muy ofuscado pero que nos da mucha libertad de optimización.
- Sólo se pueden solucionar cuadrados de tamaño 7x7. El algoritmo no se adapta a otros órdenes, cada distinto anidamiento está hecho a mano.
- No se han podido llevar a cabo todas las optimizaciones que teóricamente se pueden plantear. Entraremos en más detalle más adelante, pero hay

optimizaciones que no son compatibles entre sí por el orden de ejecución de las operaciones. También se plantean optimizaciones que se podrían llevar a cabo empleando memoria para reservar las posibilidades ya vistas, pero el gasto espacial es demasiado grande para ser asumible.

- Existen muchas soluciones que podríamos considerar equivalentes cuando las esquinas de las anidaciones rotan o se cambian la posición de los primos intermedios en filas y columnas. No se ha encontrado una manera eficaz de poder determinar estos casos para evitar computar soluciones equivalentes.

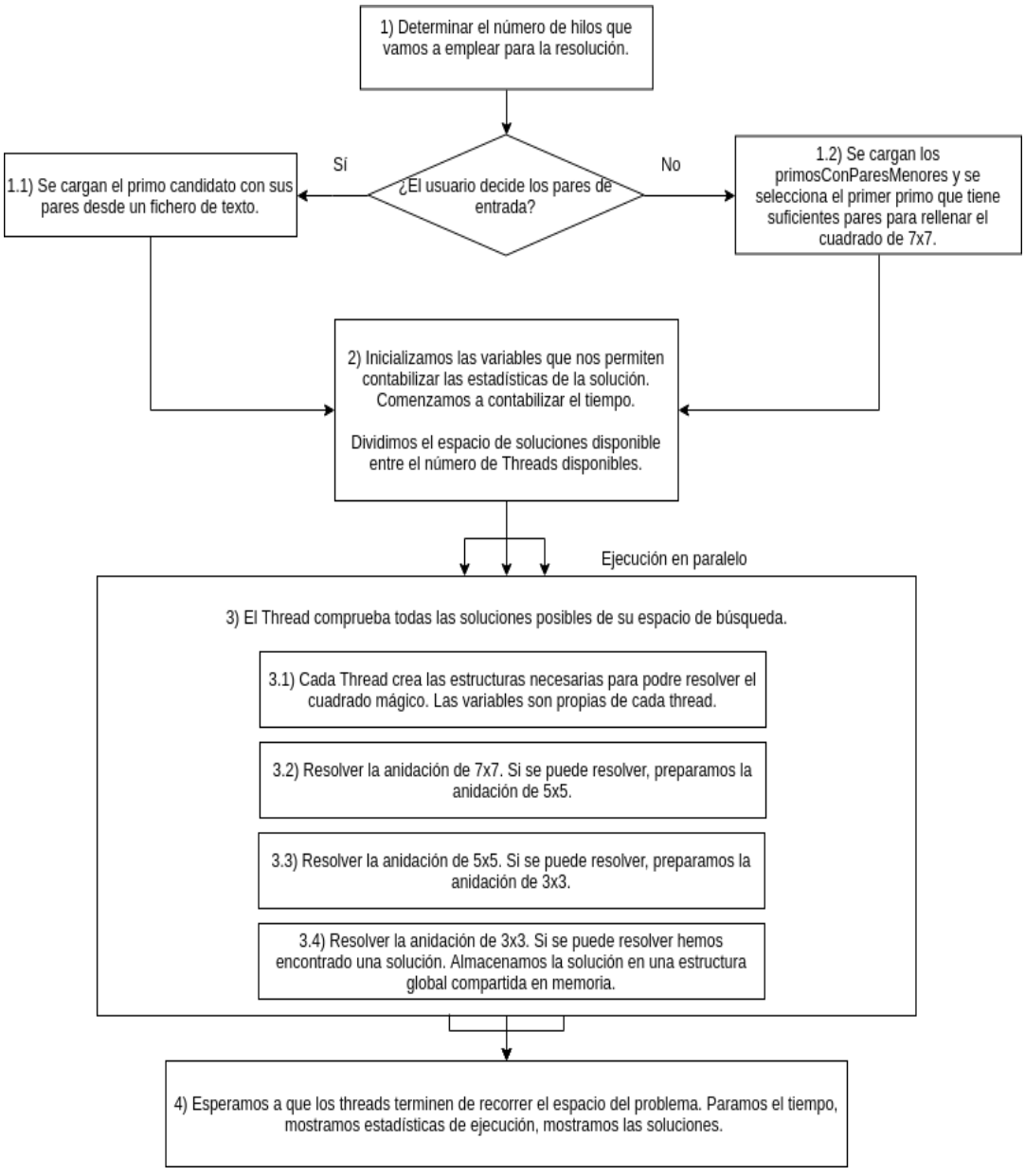

Ilustración 41 Esquema de funcionamiento del segundo algoritmo

12.1 Estructuras, métodos y organización

Las estructuras de datos empleadas son las mismas que las que se han descrito en el algoritmo anterior. La única estructura nueva que se añade es la siguiente:

```
//Estructura para guardar una solución y un puntero a la siguiente:
typedef struct _S_solucion{
    BigInt cuadrado[7][7];
    struct _S_solucion* siguiente;
} ESolucion;
```

Es una estructura recursiva para almacenar las soluciones. Cada solución está dentro de la matriz dimensional de 7x7 y existe un puntero para almacenar la siguiente solución encontrada.

Como en el algoritmo se está empleando paralelismo a nivel de hilo, es conveniente identificar aquellas estructuras que se encuentran a nivel global y son accesibles por cualquier hilo:

Las estructuras globales, en su mayor parte se emplean para almacenar información estadística sobre la búsqueda de la solución. Las he empleado preferentemente para comprobar cómo el algoritmo reduce su espacio de búsqueda cuando realizaba optimizaciones.

- ***EPrimoConPares*** **ePrimoCentral* → Estructura que almacena el primo central, el número de pares anidados en dicho primo y una matriz bidimensional que guarda los primos de los pares centrados en el primo central.
- ***long*** *totalCombinacionesFila* → Número de combinaciones totales que se comprueban para la fila superior del 7x7.
- ***long*** *totalCombinacionesOK1Fila* → Número de combinaciones que son solución para la fila superior del 7x7.
- ***long*** *totalCombinacionesTotalCorona* → Número de combinaciones para la anidación de 7x7 completa. Aquí el número de combinaciones se dispara exponencialmente. El problema es que para cada combinación correcta de Fila superior hay que volver a comprobar todos los pares laterales.
- ***long*** *combinacionesOKCorona* → Aquellas anidaciones de 7x7 que cumplen la constante mágica.

- ***long*** *numeroOK7x7y5x5Superior* → Todas las combinaciones correctas de la anidación completa de 7x7 y la superior de 5x5
- ***long*** *numeroOK7x7y5x5Coronas* → Todas las combinaciones correctas que han llegado hasta el 5x5 pasando previamente por el 7x7.
- ***long*** *numeroOK7x7y5x5OKy3x3Superior* → Todas las combinaciones correctas que resuelven el 7x7, el 5x5 y la fila superior del 3x3.
- ***long*** *numSoluciones* → Todas las combinaciones correctas que resuelven el 7x7, el 5x5 y el 3x3. Es decir, aquellas que son solución para los cuadrados mágicos de 3x3.

Y aquellas estructuras que son propias de cada hilo:

Existen las mismas **estructuras interiores** para cada anidación. Su función es la misma, pero su tamaño varía según el orden. Para evitar la repetición, indicamos para qué se utilizan dichas estructuras sin repetir a los 3 niveles, sólo en el superior:

- ***int*** **filaSuperiorIndiceSelePar* → Es la estructura que almacena los índices de los pares para la fila superior de 7x7. Ocurre como en el algoritmo anterior, sólamente se almacena el índice del par.
- ***int*** **filaSuperiorIndiceSeleParAA* → Para cada par selecionado con *filaSuperiorIndiceSelePar* se guarda un entero para identificar si seleccionamos el primer o segundo primo dentro del par.
- ***int*** * *pareslibres* → Almacena los índices de los pares libres para la columna lateral.
- ***int*** **columnaDerIndiceSelePar* → Almacena los índices de *paresLibres* empleados para seleccionar los pares candidatos de la columna lateral derecha.
- ***int*** **columnaDerIndiceSeleParAA* → Almacena un 0 o un 1 para cada *columnaDerIndiceSelePar*. Un 0 si en la combinación participa el primer primo del par y un 1 en caso contrario.

Además, para cada anidación, tenemos que volver a recalcular los pares que tenemos disponibles, de tal manera que se emplea una estructura extra:

- ***int*** **indicesLibres5x5*→ Es un array que guarda los índices libres que han quedado tras la anidación anterior 7x7 (hay una estructura similar para el 3x3).

Con este algoritmo, siempre tenemos en memoria, y sin emplear otras estructuras que se almacenen en pila, todos los índices que se han empleado a lo largo de la anidación. Eso nos va a permitir descartar antes muchas posibilidades y acelerar el algoritmo. Por contra, se complica mucho la nomenclatura de los primos en los pares. Por ejemplo, para seleccionar un par en la anidación lateral 5x5 tenemos que emplear la siguiente instrucción:

ePrimoCentral->pares[columnaDerIndiceSeleParAA5x5[i]][indicesLibres5x5[pareslibres5x5 [columnaDerIndiceSelePar5x5[i]]]];

Donde i es el índice del par que empleamos.

Como podemos observar al repasar el código fuente, el algoritmo es básicamente el mismo pero transformado en secuencial. Todas las llamadas recursivas se transforman en una cantidad de *fors* anidados unos dentro de otros y de estructuras que hacen las veces de la pila en las llamadas anteriores.

12.2 Optimizaciones

Un detalle de la estructuración del algoritmo que es especialmente importante es cómo en cada una de las anidaciones de *for* para solucionar filas y columnas se intercambian los fors para seleccionar los índices de los pares y los fors para seleccionar los primos dentro de los pares.

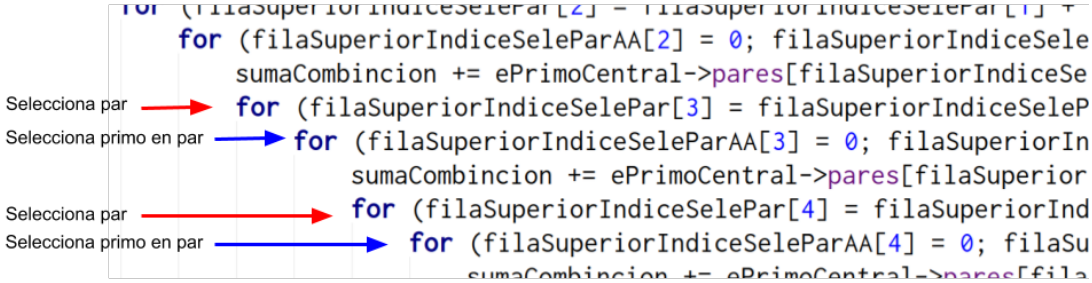

Ilustración 42 Detalle de estructuración para la optimización

Este detalle hace que sea posible realizar varias optimizaciones. Antes de explicar cada una de ellas, vamos a ver cuántas combinaciones existen si fuese necesario recorrer todo el espacio de búsqueda:

Número de combinaciones teórico para las combinaciones de 7x7 en su fila superior

Tenemos 24 pares. Debemos extraer combinaciones de 24 números tomados de 7 en 7. Que sean combinaciones implica que el orden no importa. Esto se puede expresar de la siguiente manera:

$C(7\ 24) = \frac{24!}{7!(24-7)!}$ = 346.104 combinaciones distintas de pares.

Ahora, para cada selección de pares, puede que se elija el primero o el segundo del par, los cálculos son muy sencillos:

$346.104 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 346.104 \times 2^7 = 44.301.312$ de combinaciones posibles sólo para la anidación de 7x7.

Los tiempos de ejecución para anidar la primera fila de 7x7 son los siguientes:

Número Ejecución	Tiempo (s)
1	0.54598802
2	0.52016997
3	0.53438598
4	0.51713800
Media de las ejecuciones	0.5294204925

Ilustración 43 Tiempo de la fila 7x7 sin optimizaciones

De todas las combinaciones posibles, 44.301.312, sólo 64.924 filas suman la constante mágica, aproximadamente un 0,15% de las combinaciones son válidas para poder continuar con la solución del cuadrado.

De todas las combinaciones posibles, 44.301.312, sólo 64.924 filas suman la constante mágica, aproximadamente un 0,15% de las combinaciones son válidas para poder continuar con la solución del cuadrado.

Con estas premisas, veamos las optimizaciones que se han llevado a cabo. **Para explicar las optimizaciones, no hace falta tener en cuenta todo el cuadrado,** las mismas optimizaciones que se aplican para la fila superior del 7x7 se aplican para su

lateral, para la fila superior del 5x5, para el lateral del 5x5, para la fila superior del 3x3 y para el lateral del 3x3.

12.2.1 Optimizacion 1) Parar tras solución parcial.

Para seleccionar los primos de la fila superior, empleamos dos estructuras de índices: *filaSuperiorIndiceSelePar* que selecciona los índices de los pares y *filaSuperiorIndiceSeleParAA* que selecciona los primos dentro de cada par. Si tenemos la combinación entera de *filaSuperiorIndiceSelePar*, y vamos ahora combinando las *filaSuperiorIndiceSeleParAA,* podemos dejar de buscar en cuanto encontremos un *filaSuperiorIndiceSeleParAA* que nos ofrezca una solución. Es decir, no nos interesa sacar más de una solución con la misma selección de pares, ya que los pares restantes que nos quedan son los mismos y la ejecución del algoritmo después sería exactamente idéntica.

Elegir el primero o el segundo del par para cada uno de los pares no tienen ninguna relevancia a efectos de continuar con el algoritmo.

La primera optimización consistiría en una vez encontrada una combinación de primos en pares que cumpla con la constante mágica, descartar todas las posibles combinaciones restantes.

La ganancia, pese a que puede parecer pequeña, es muy grande, pues cada pequeña ejecución más se multiplica en tiempo en las ejecuciones interiores del cuadrado.

Esta optimización no ha sido empleada en la versión final del algoritmo por la estructuración que se lleva a cabo. Como hemos dicho anteriormente, se mezcla el avance de los índices de los pares con la selección de primos en los pares, por lo que no es posible realizar un descarte, no podemos cortar ninguno de esos fors. **Esta optimización sólo sería posible si primero se calculasen todos los índices de los pares y después todos los índices de los primos dentro de los pares** (como ocurría en el algoritmo anterior).

12.2.2 Optimización 2) Evitar combinaciones simétricas

Al resolver una fila o columna, no tiene sentido probar una combinación de pares que tengan los primos seleccionados de forma simétrica. Por ejemplo, si tenemos 3 pares:

A	B	C
A’	B’	C’

Podemos seleccionar, entre otras, la combinación:

A	B	C
A’	**B’**	**C’**

Si suponemos que dicha combinación suma la constante mágica k → **a+b’+c’ = k**

Entonces su combinación simétrica también suma la constante mágica, como demostramos en la parte teórica del algoritmo:

A	**B**	**C**
A’	B’	C’

a’+b+c = k

Si pintamos las posibles combinaciones de 3 elementos seleccionando con un 0 el primero del par o con un 1 el segundo del par, tenemos los siguientes resultados:

Ilustración 44 Combinaciones de pares de 3 en 3

En la imagen anterior, todas las combinaciones de la izquierda empiezan por cero y todas sus simétricas empiezan por uno, a su derecha.

Esta característica se puede aprovechar a nuestro favor para no tener que procesar combinaciones equivalentes cada vez que resolvamos filas o columnas. En el caso de la fila de 7x7 superior, tenemos la estructura *filaSuperiorIndiceSeleParAA* que nos va almacenando los índices posibles que se pueden ir seleccionando para los pares empleados en dicha fila. Guarda exactamente 0 y 1 como se han representado en la figura anterior dependiendo de la combinación seleccionada.

Aprovecharse de esta optimización es tan sencillo como que para el primer primo de cada par seleccionado, ya sean filas o columnas,su valor siempre va a ser 0. Es decir el *filaSuperiorIndiceSeleParAA[0]* siempre va a valer 0, todas las combinaciones si valiese 1 se prueban por simetría.

Esta optimización es fundamental para el algoritmo, pues reducimos a la mitad el espacio de búsqueda total. Veamos los tiempos para la fila superior empleando esta optimización:

Optimización 2	Tiempo (s)
1	0.26207200
2	0.25320700
3	0.25661501
4	0.25949204
Media de las ejecuciones	0.2578465125

Ilustración 45 Tiempo de la fila 7x7 con optimización 2

Ahora el número de combinaciones posibles se reduce a la mitad, es de 22.150.656. De las cuales son solución 32.462. Este dato es muy importante, pues no sólo probamos la mitad de combinaciones, sino que profundizamos en la mitad de combinaciones. **La ganancia de esta implementación es mucho mayor que el 50% de una pasada, como aparece en la tabla**, pues reduce el número de llamadas interiores al 50% en cada paso. **Es decir, a nivel de una fila o columna reduce a la mitad el tiempo necesario, pero como hay la mitad de soluciones, la próxima pasada se hará sobre un 50% de la mitad.**

Esta optimización pone de manifiesto cómo un bucle for extra puede optimizar o ralentizar mucho la ejecución. Llevar a cabo una anidación más… el 9x9 es exponencialmente más costoso.

12.2.3 Optimización 3) Operaciones parciales en cada iteración

En un primer momento, los cálculos para comprobar si una combinación es mágica o no se llevaban a cabo al final de cada combinación. Es decir, se van generando los índices de los pares, a su vez se generan los primos seleccionados en esos índices, y finalmente, cuando ya tenemos todos los índices seleccionados, sumamos los valores de todos los primos y comprobamos si coinciden con la constante mágica o no.

Este proceder hace que cada vez que queramos validar una combinación, tengamos que sumar n cantidades, siendo n el orden del cuadrado.

Es posible optimizar el código si a la vez que vamos generando índices vamos sumando parcialmente el resultado:

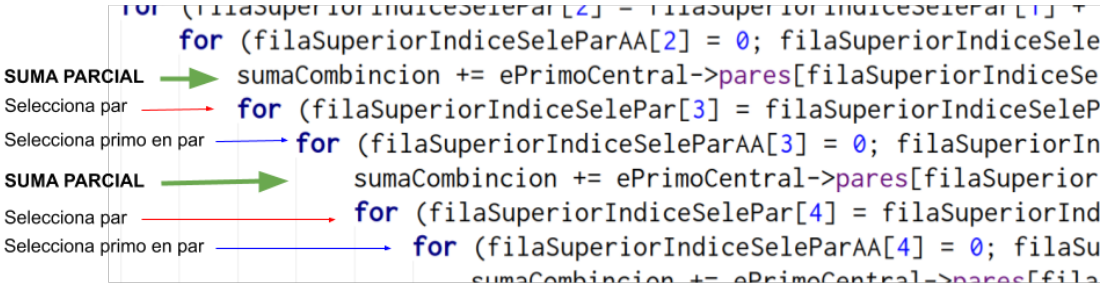

Ilustración 46 Optimizando con la suma parcial

Y a la vez, cada vez que volvemos por backtracking, restamos ese valor sumado:

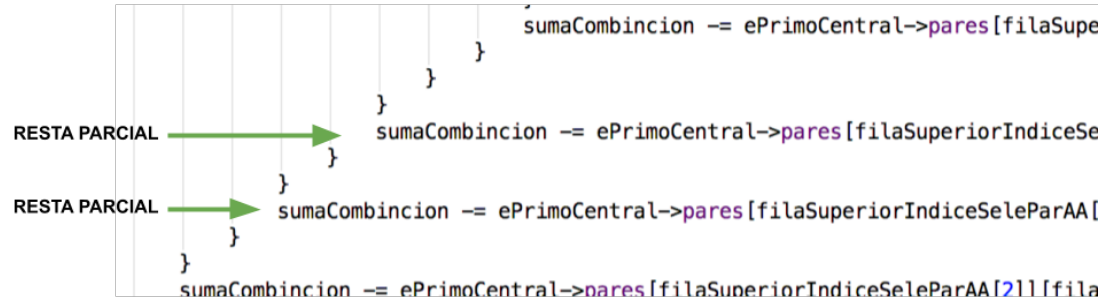

Ilustración 47 Restaurando la suma parcial

El número de sumas totales se ve reducido. Cuanto más grande sea el orden a resolver más ganancia vamos a tener con esta aproximación. Por ejemplo para el cuadrado de 7x7 sólo es necesario sumar y restar la primera cantidad del cuadrado 17 x 2 = 34 veces (una por cada combinación distinta posible, al inicio hay 24 pares - 7 posiciones = 17 x 2 (sumas inicial y resta final)). De la otra manera habría que sumar esos primeros números 22.150.656 veces (una por cada combinación posible). Obviamente la ganancia en este primer caso es muy importante. Según vamos profundizando, cada vez la ganancia es menor.

La siguiente tabla muestra los tiempos tras haber aplicado esta optimización:

Optimización 3	Tiempo (s)
1	0.20105600
2	0.19578300
3	0.22579300
4	0.19592100
Media de las ejecuciones	0.20463825

Ilustración 48 Tiempo de la fila 7x7 con optimización 3

12.2.4 Optimización 4) Descarte por superar la constante mágica

Una vez hemos implementado la suma parcial según vamos generando la combinación podemos empezar a descartar conjuntos grandes de combinaciones cuando sabemos que su suma nunca va a poder valer la constante mágica.

Esta optimización se lleva a cabo en cada uno de los bucles for, siembre que la suma parcial sea mayor que la constante mágica ya podemos descartar todas las posibilidades que empiecen con esa combinación de primos:

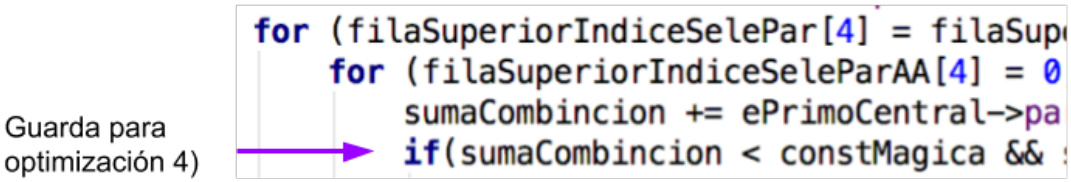

Ilustración 49 Detalle de la optimización 4

Con esta guarda los tiempos pasan a ser los siguientes:

Optimización 4	Tiempo (s)
1	0.17848399
2	0.17964400
3	0.17912900
4	0.17605500
Media de las ejecuciones	0.1783279975

Ilustración 50 Tiempo de la fila 7x7 con optimización 4

Mejoramos ligeramente el tiempo porque ahora el número de combinaciones que se comprueban es bastante menor 16.918.346, manteniendo el mismo número de soluciones 32.462.

12.2.5 Optimización 5) Descarte por no poder alcanzar la constante mágica

De manera similar a la optimización 4, podemos descartar combinaciones porque la suma parcial es tan baja que ya no es posible alcanzar la constante mágica. El valor máximo que podría tener un primo es dos veces el valor del primo central, luego por cada primo que falte por colocar nos podemos distanciar como mucho 2 veces el valor del central. Esos valores se encuentran como guardas a lo largo del código:

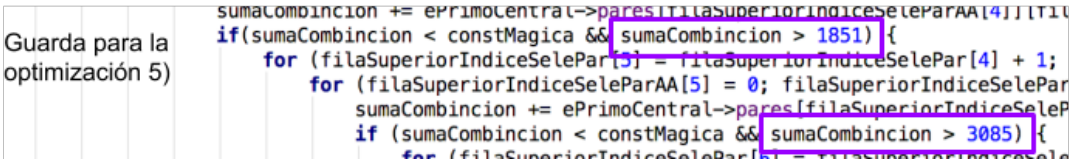

Ilustración 51 Detalle de la optimización 5

Con esta guarda los tiempos pasan a ser los siguientes:

Optimización 5	Tiempo (s)
1	0.14691600
2	0.14899500
3	0.15324900
4	0.14734800
Media de las ejecuciones	0.149127

Ilustración 52 Tiempo de la fila 7x7 con optimización 5

Mejoramos ligeramente el tiempo porque ahora el número de combinaciones que se comprueban es bastante menor 10.191.502, manteniendo el mismo número de soluciones 32.462.

En la siguiente gráfica podemos ver cómo se han mejorado los tiempos medios tras cada optimización para componer la fila superior de 7x7.

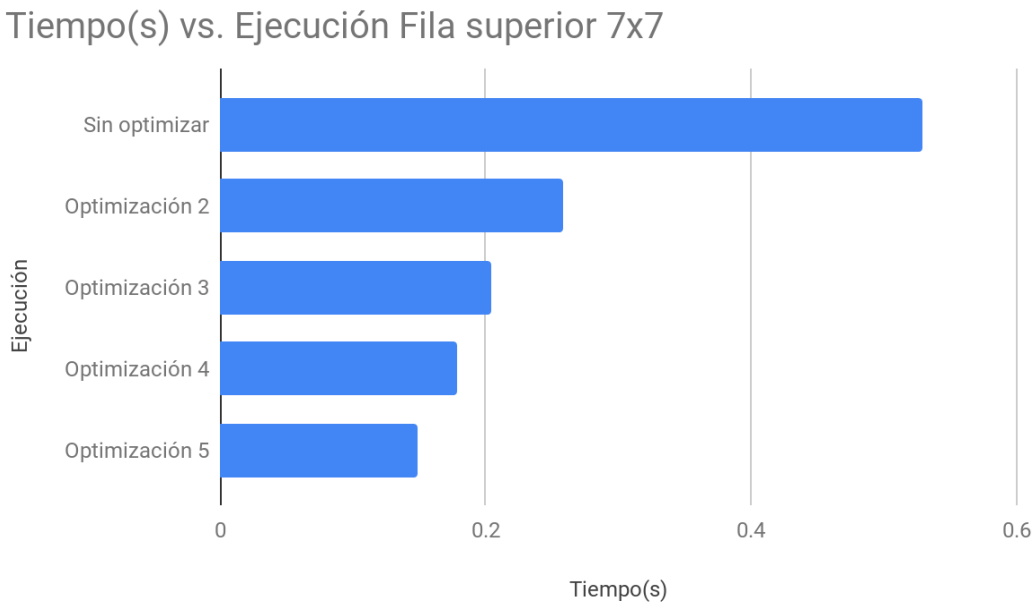

Ilustración 53 Comparativa de la mejora temporal tras aplicar optimizaciónes

Estas optimizaciones se han llevado a cabo por cada fila y columna del algoritmo.

12.3 Paralelización del algoritmo. OpenMP

Para llevar a cabo la paralelización del algoritmo utilizamos OpenMP,, que es una especificación creada por Intel para paralelizar código escrito en C, C++ ó Fortran.

Dentro de las librerías del sistema accesibles con C tenemos *pthread*, que nos permitiría crear hilos para especificar la programación paralela de nuestros algoritmos. OpenMP nos ofrece una serie de directivas que nos aíslan de la complejidad de crear a bajo nivel los hilos y sincronizarlos.

En el caso de nuestro algoritmo, nos interesa utilizar una aproximación SPMD, *Single Program Multiple Data*. Es muy común en computación paralela plantear un problema y correr varias copias del algoritmo que dividen el espacio de búsqueda del problema. Con OpenMP solicitamos varios hilos de ejecución al sistema operativo. Así, teniendo en cuenta el ID de cada *Thread* y sabiendo cuántos *threads* hay, ajustamos la ejecución de cada uno para repartir el trabajo.

En este tipo de problemas tenemos que tener en cuenta *false sharing*: Sucede cuando estamos compartiendo memoria, especialmente con arrays. Aunque nunca se solape la misma memoria que estamos tocando por distintos hilos, si esta se encuentra contigua y ocupa el mismo bloque, entonces cada vez que hacemos una operación en un hilo tenemos que pasar el bloque de un procesador a otro, destruyendo todas las ventajas de la paralelización.

El único punto en el cuál podría ocurrir algo así, sería que estemos tocando memoria que debe ser empleada por varios hilos. Esto podría ocurrir con los pares del primo central, pero como sólo los estamos empleando para leer, no para escribir, no habría problemas de actualizaciones entre los distintos cores. Por otra parte, el tamaño de un *BigInt* coincide con el tamaño del bloque de caché en las máquinas en las que estamos ejecutando el algoritmo, eliminando el problema del *False Sharing* de raíz.

Los *threads* que se crean con OpenMP comparten la memoria y el código del programa principal, pero cada uno tiene su pila independiente, su contador de programa y sus registros individuales. Las **estructuras globales** son generales a nivel de programa, y por lo tanto accesibles por cada uno de los *threads*, mientras que las **estructuras internas** son individuales a nivel de hilo porque se crean en el *stack* de cada uno.

El siguiente esquema muestra la estrategia de paralelismo aplicada al algoritmo.

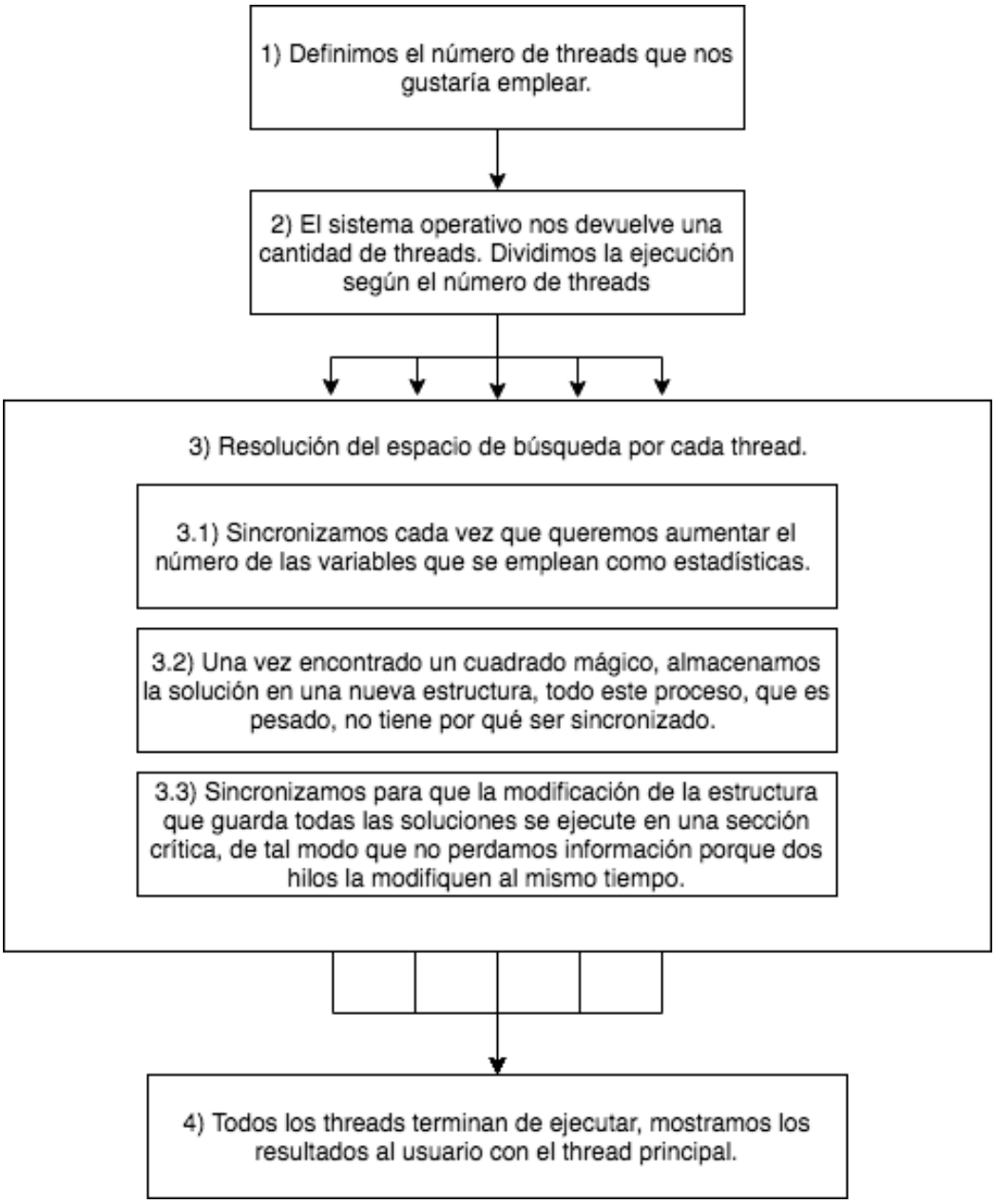

Ilustración 54 Esquema de paralelización del algoritmo 2

La distribución del espacio de búsqueda por los diferentes *threads* se denomina **simple distribution of a loop operation**. Este esquema de paralelismo se basa en incrementar la variable índice del bucle con respecto al número de *threads* creados.

Por cómo funciona el algoritmo, las primeras ejecuciones del algoritmo, es decir, los primeros *threads*, tienen más carga que los últimos. Esto es así porque en el bucle que se paraleliza, lo que se intenta es generar todas las combinaciones sin repetir y sin orden que hay desde esa posición inicial hasta el final. Obviamente, cuanto más cerca estemos del final, menos combinaciones habrá que comprobar.

Para contar el tiempo con OpenMP tenemos que emplear sus propias instrucciones, no se pueden emplear tácticas que empleen el número de ticks de reloj que se han producido en el intervalo que queremos medir y luego multiplicar por el tiempo que tarda cada tick de reloj, ya que se producen ticks en paralelo y las mediciones no son realistas. Se emplean las siguientes instrucciones:

start = omp_get_wtime();

[...]

end = omp_get_wtime();

*printf("**Tiempo total: %.8lf\n**", (**double**) (end - start));*

Esto permite medir el tiempo al inicio y al final del algoritmo y poder establecer el intervalo real entre los dos puntos.

Para poder comprobar realmente cuánto es la ganancia que se obtiene con la paralelización, hemos comparado en la siguiente gráfica los tiempos en comprobar todas las posibilidades de la corona 7x7. Dependiendo del grado de paralelismo que se emplee, los tiempos se reducen:

Combinaciones total corona	70756407616	70756407616	70756407616	70756407616	70756407616
Coronas OK	67244878	67244878	67244878	67244878	67244878
Tiempo	563.6021713	331.4194066	227.3548404	191.6385282	185.902264
Número de threads	1	2	4	8	16
% Ganancia		170.06%	247.90%	294.10%	303.17%

Ilustración 55 Progresión del algoritmo según el grado de paralelización

Por supuesto, los resultados obtenidos son independientes del número de *threads* que se utilicen para resolver el problema.

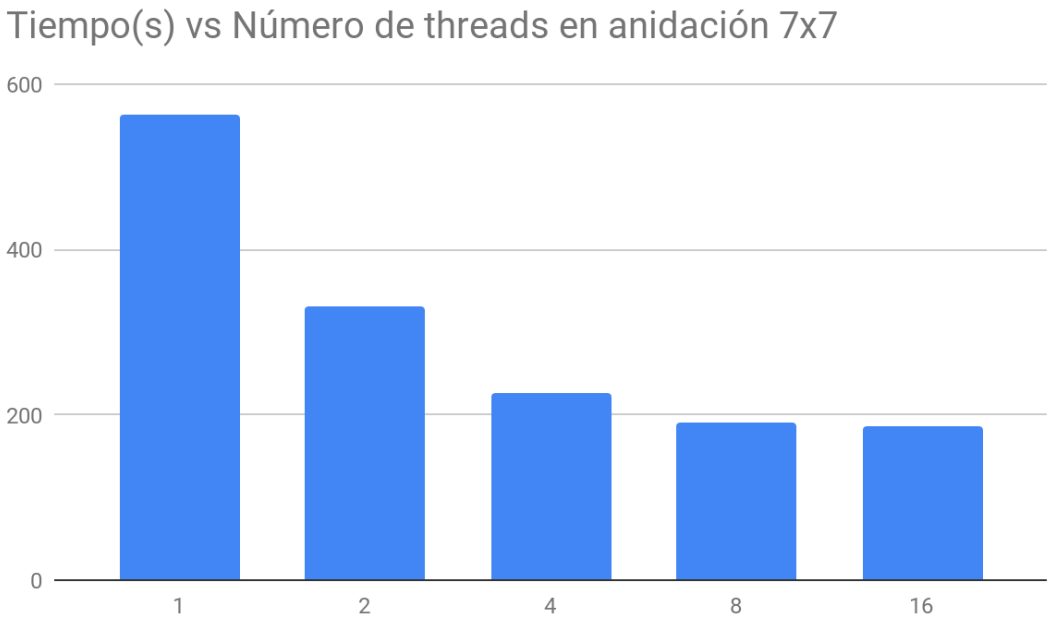

Ilustración 56 Relación entre número de threads y rendimiento

Queda en evidencia que el margen de mejora a partir de 8 threads es marginal, esto se debe en parte a dos cuestiones:

1. El equipo sobre el que se han realizado las pruebas tiene una CPU Ryzen 2600 de 8 cores.
2. El espacio de búsqueda no se reparte equitativamente entre las ejecuciones del bucle que se paraleliza. Cuanto más cerca estemos del final, menos carga tienen esas iteraciones.

12.4 Conclusiones

Esta versión del algoritmo surge para hacer frente a las carencias que tenía la primera aproximación. En este caso, hemos acotado aún más las restricciones, limitando el cuadrado al orden 7x7 por las siguientes razones:

- El código no tiene por qué ser adaptativo.
- El código puede ser completamente escrito sin necesidad de llamadas recursivas que hacen complicada la optimización y el paralelismo.
- Tener un tamaño reducido permite solucionar o probar ciertas partes a mano para comprobar y verificar tanto el algoritmo como sus optimizaciones.

- Cualquier tamaño mayor de 7x7, incluso con la versión final optimizada, tarda en ejecutarse demasiado tiempo como para realizar pruebas.

La principal diferencia es la estructuración del código. Ahora, a la vez que se seleccionan los pares, se seleccionan los índices de los primos dentro de los pares, permitiendo reducir en gran medida el tiempo de ejecución gracias a la optimización por descarte ante combinaciones incompletas que nunca van a poder ofrecer una solución.

La medida principal de optimización proviene del reconocimiento de la selección de primos en pares, que reduce el tiempo de ejecución en más de la mitad. Consideramos que el uso y empleo de la paralelización tiene sentido siempre que antes hayamos realizado un ejercicio de optimización minucioso para que realmente trabajemos sobre una buena base sobre la que paralelizar.

La modificación para implementar OpenMP, tal y como está planteado el algoritmo, con bucles secuenciales, es prácticamente trivial. La gran complejidad del código de *fors* anidados hace que la paralelización sea tan sencilla como aplicar un modelo de distribución del espacio de búsqueda mediante el primer bucle *for*.

La mejora de tiempos por paralelización, de esta manera, se incrementa casi en un orden de magnitud por cada thread utilizado, pero a medida que se aumenta el número de hilos, cada vez la ganancia es mucho más marginal. El algoritmo se comporta bien ante una paralelización con 4-8 núcleos, pero a partir de ahí sería conveniente tomar otras alternativas para mejorar el rendimiento.

La versión final de este algoritmo se ha llevado a un clúster de computación del grupo ARCO de la Universidad de Extremadura, con el cual se han podido resolver completamente cuadrados mágicos de 7x7 en un tiempo aproximado de 8 horas.

13 ELEMENTOS DESTACABLES

¿Cómo paralelizar?

Antes de tomar la decisión de emplear OpenMP se estudiaron distintas alternativas que nos podrían ayudar a paralelizar el algoritmo. A continuación, mostramos algunas de las ideas en las que se trabajaron y el motivo por el que fueron descartadas.

- **Computación en GRID**. Cuando se inició este proyecto de fin de máster todavía era una tecnología con impacto en la supercomputación. Por características del problema, es muy fácil adaptarlo a un GRID: tenemos el mismo problema (encontrar un cuadrado mágico de orden n) y queremos solucionarlo probando con diferentes entradas de datos (distintos pares iniciales). Se podría aprovechar la arquitectura con memoria no compartida de un Grid para que cada nodo intentara solucionar el problema con distintos pares y enviar mensajes con los resultados.

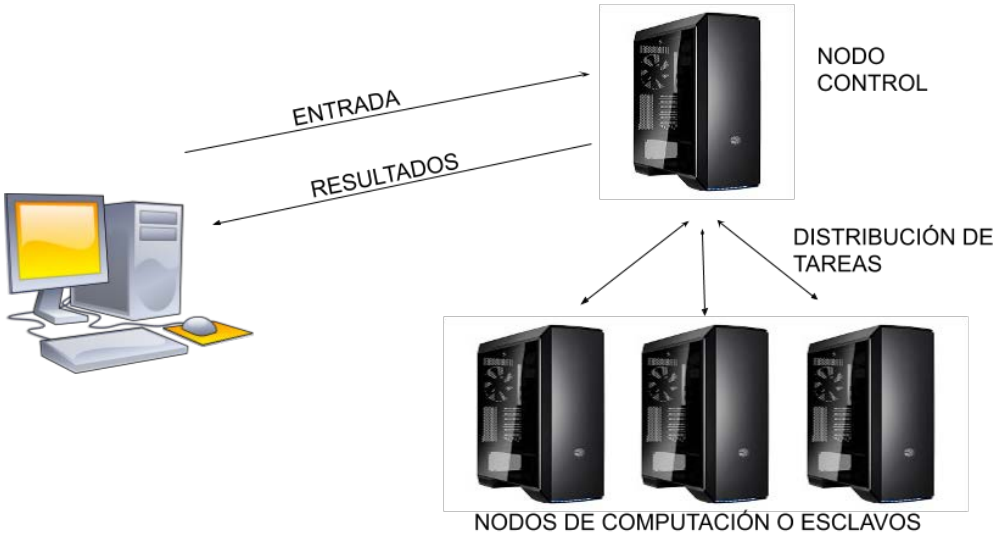

Ilustración 57 Representación de computación GRID

Esta estructura no se ha empleado porque en el curso 2018-2019 ya no es una tecnología tan utilizada y no hemos podido obtener acceso a una GRID para realizar pruebas.

- **Computación distribuida con tecnologías web**. Como hemos comentado anteriormente, considerábamos que una distribución del problema con distintas entradas era una buena solución para acelerar el procesamiento sin necesidad de modificar el algoritmo para adaptarlo a una paralelización de memoria compartida e hilos, como la que puede tener OpenMP.

  Como todo el código estaba escrito en C, se podría haber empleado WebAssembly para transformar automáticamente el código y que se pudiera ejecutar directamente en los navegadores. El esquema constaría de un servidor que distribuiría las tareas, de manera similar al nodo raíz del clúster, y los clientes se conectan, se les devuelve el algoritmo y la entrada, lo ejecutan y devuelven la solución. La comunicación estaría implementada como una API REST.

  Se descartó esta aproximación porque, en la práctica, era mucho más lenta que el código nativo. Además, ejecutar tareas pesadas en los navegadores no es posible porque quedan bloqueados procesando y no pueden utilizarse mientras hacen el procesamiento.

¿Qué gestor de build emplear?

Como el algoritmo se va ejecutar en distintas máquinas, se planteó desde el principio la necesidad de emplear un gestor de *builds* que facilitase la tarea de generar un ejecutable, sobre todo en el caso en el que necesitáramos enlazar dinámicamente librerías externas específicas que serán distintas para cada plataforma. Entre las opciones, CMake es la alternativa más empleada, por su relativa sencillez y por estar presente en casi todos los entornos.

Al final no han sido necesarias librerías dinámicas, excepto OpenMP, cuyo enlazado es muy simple. Por otra parte, en el clúster no está instalado CMake, por lo que la compilación en este entorno se ha hecho manualmente.

Scripts auxiliares

Se han generado los siguientes programas y scripts que se adjuntan con la entrega del proyecto:

- **GeneraPrimos**: Es un programa en C que realiza una criba de Eratóstenes. Los números cribados se almacenan en un fichero binario, por cada uno de ellos se emplea un BigInt.
- **GeneraPares**: Es un programa en C que emplea los números cribados para transformarlos en primos con pares centrados. Generar gran cantidad de pares centrados en cada uno de los primos tiene un coste computacional alto, de ahí que se calculen y queden almacenados en ficheros binarios que son la entrada de los dos algoritmos expuestos anteriormente.
- **JavaScript-Grid**: Una implementación simple de un grid computing empleando javascript para solucionar parcialmente un cuadrado mágico de primos. Queda de manifiesto cómo es mucho más sencillo de codificar, pero el rendimiento es mucho menor y el
- **ScriptsCluster**: Son los scripts en bash que se han creado para automatizar la subida del algoritmo al clúster, su compilación, ejecución y limpieza.

14 APUNTES FINALES Y PLANTEAMIENTOS FUTUROS

Este documento recoge un trabajo elaborado a lo largo de varios años sobre el estudio de la creación y composición de los cuadrados mágicos conformados con números primos.

Desde una visión teórica, se ha establecido que la cantidad de combinaciones posibles necesarias para resolver completamente un cuadrado mágico formado por primos es tan elevada que descarta procesados sencillos. Gran parte del trabajo llevado a cabo consiste en generar nuevas limitaciones o restricciones que acotan el espacio de búsqueda. La anidación de cuadrados concéntricos surge como principal medida para reducir el espacio de búsqueda hasta un margen computacionalmente manejable.

La construcción de los cuadrados consiste en crear iterativamente cuadrados de orden menor, donde el primer cuadrado generado es el de orden N y el siguiente es N-2 y así sucesivamente hasta llegar al 3x3 y generar una solución.

Para finalizar, destacamos los dos algoritmos propuestos. El primero encuentra una solución para un orden cualquiera dado. El segundo permite discriminar, para un orden fijo, si existe solución o no empleando paralelismo.

Como trabajo futuro a esta investigación se pueden plantear líneas enfocadas a aprovechar lo mejor de los dos algoritmos: que sean adaptativos y con una estructura de código en parte recursiva (por su facilidad para comprenderlo) y en parte optimizable y paralelizable (para aprovechar totalmente la capacidad de computación).

15 REFERENCIAS BIBLIOGRÁFICAS

1 Futility Closet, “Cuadrado mágico del prisionero”. http://www.futilitycloset.com/2010/08/14/time-well-spent/

2 MathWorld, “Cuadrado de Dürer”. http://mathworld.wolfram.com/DuerersMagicSquare.html

3 J. P. N. Phillips, “Journal of the Royal Statistical Society. Series C (Applied Statistics)” Vol. 13, No. 2 (1964), pp. 67-73

4 Stephano T., “My research of prime magic squares with the computer help”, https://digilander.libero.it/ice00/index.html

5 P. Pritchard, Linear prime-number sieves: a family tree, Sci. Comput. Program. 9 (1987) 17–35

6 Liskov M. (2011) Miller–Rabin Probabilistic Primality Test. In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security. Springer, Boston, MA