Universidad Rey Juan Carlos

Prueba de Diseño de Algoritmos – Curso 2017-2018 – 18 de mayo de 2018

Diseño y Análisis de Algoritmos

Valor: 65 % de la nota final. Duración: **2 horas y 45 minutos**

---

1. Análisis de algoritmos

Ejercicio 1 [1 punto]

Demuestra **mediante la definición** de complejidad asintótica Ω (sin usar límites), si se verifica:

$$(n+1)! \in \Omega(n!)$$

Posible solución:

Para verificar la expresión debemos encontrar una constante $c > 0$, y otra $n_0 > 0$ tal que

$$(n+1)! \geq c \cdot n!$$

se cumpla para todo $n \geq n_0$. En este caso, podemos escoger simplemente $c = 1$. En concreto, tendríamos:

$$(n+1)! \geq n!,$$

y faltaría analizar para qué valores de n se verifica dicha expresión. Dividiendo ambos lados por $n!$ obtenemos:

$$\frac{(n+1)!}{n!} \geq \frac{n!}{n!}.$$

Como $(n+1)! = (n+1) \cdot n!$, simplificando obtenemos:

$$n + 1 \geq 1,$$

lo cual se cumple para todo $n \geq 0$. Por tanto, podemos coger como n_0 cualquier valor positivo, por ejemplo, $n_0 = 1$.

Ejercicio 2 [2.5 puntos]

Calcula el número de operaciones ($T(n)$) que realiza el siguiente código:

```
1  int i=0;
2  while (i<=n){
3      int j=i;
4      while (j<=n){
5          procesa(i,j);  // dos operaciones
6          j++;
7      }
8      i++;
9  }
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Posible solución:

El algoritmo se compone de dos bucles while anidados, donde las expresiones de inicialización (líneas 1 y 3), y de incrementos (líneas 6 y 8) aparecen de manera explícita. Las comparaciones también aparecen de manera explícita en las líneas 2 y 4. Para analizar el algoritmo usaremos la fórmula para calcular el número de operaciones que realiza un bucle:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n}(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}) + 1_{\text{última comparación}}.$$

En este caso, la función T que mide el número de operaciones (en función de n) será igual a la función asociada al bucle externo:

$$T(n) = 1 + \sum_{i=0}^{n}(1 + T_{\text{bucle\_interno}} + 1) + 1,$$

mientras que

$$T_{\text{bucle\_interno}} = 1 + \sum_{j=i}^{n}(1 + T_{\text{cuerpo}} + 1) + 1.$$

Como $T_{\text{cuerpo}} = 2$, por la instrucción `procesa(i,j)` (ya hemos tenido en cuenta el incremento `j++` como parte integral del bucle) tenemos:

$$T_{\text{bucle\_interno}} = 1 + \sum_{j=i}^{n}(1 + 2 + 1) + 1 = 2 + \sum_{j=i}^{n} 4 = 2 + 4(n - i + 1) = 4n - 4i + 6.$$

Sustituyendo en $T(n)$ obtenemos:

$$T(n) = 1 + \sum_{i=0}^{n}(1 + 4n - 4i + 6 + 1) + 1 = 2 + \sum_{i=1}^{n}(4n - 4i + 8).$$

Descomponemos el sumatorio en tres:

$$T(n) = 2 + \sum_{i=0}^{n} 4n - \sum_{i=0}^{n} 4i + \sum_{i=0}^{n} 8 = 2 + 4n(n+1) - 4\sum_{i=1}^{n} i + 8(n+1).$$

Usando la fórmula de la suma de los primeros números naturales obtenemos:

$$T(n) = 2 + 4n^2 + 4n - 4\frac{n(n+1)}{2} + 8n + 8 = 2 + 4n^2 + 4n - 2n^2 - 2n + 8n + 8$$

$$= 2n^2 + 10n + 10.$$

Se puede comprobar que la fórmula es correcta con este código (que no se pide en el ejercicio), que acumula el número de operaciones en la variable `cont`:

```
def suma(n):
    cont = 0;

    i=0
    cont = cont + 1  # una inicialización

    while i<=n:
        cont = cont + 1  # una comparación verdadera

        j=i
        cont = cont + 1  # una inicialización

        while j<=n:
            cont = cont + 1  # una comparación verdadera

            cont = cont + 2  # dos operaciones básicas

            j=j+1
            cont = cont + 1  # un incremento

        cont = cont + 1   # una comparación falsa (ya que j>n)

        i=i+1
        cont = cont + 1  # un incremento

    cont = cont + 1   # una comparación falsa (ya que i>n)

    return cont


for n in range(0,10):
    print(suma(n),' = ',2*n*n + 10*n + 10)
```

Ejercicio 3 [2.5 puntos]

Resolved la siguiente relación de recurrencia por el **método general de resolución de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \log_2 n & \text{si } n > 1 \end{cases}$$

Posible solución:

Como la recurrencia no es una ecuación en diferencias (en el lado derecho tenemos $T(n/2)$, en lugar de términos del tipo $T(n-k)$; es decir, se reduce el argumento aplicando una división en vez de restas) es necesario aplicar un cambio de variable. Como se divide el argumento por dos, el cambio de variable debe ser $n = 2^k$. De esta manera tenemos:

$$T(n) = T(2^k) = 2T(2^{k-1}) + \log_2 2^k = 2T(2^{k-1}) + k.$$

Dado que $T(2^k)$ es una función que depende de k, la podemos llamar $t(k)$. Así, tendríamos:

$$T(n) = T(2^k) = t(k) = 2t(k-1) + k.$$

Podemos reorganizar la expresión de la siguiente manera:

$$t(k) - 2t(k-1) = k^1 \cdot 1^k.$$

A continuación calculamos el polinomio característico. Del lado izquierdo obtenemos el término $(x-2)$, y del derecho $(x-1)^2$. Por tanto, el polinomio característico es:

$$(x-2)(x-1)^2.$$

Esto implica que:

$$t(k) = C_1 2^k + C_2 1^k + C_3 k 1^k = C_1 2^k + C_2 + C_3 k$$

Para hallar las constantes necesitamos tres casos base. El enunciado indica que $T(1) = 1 = t(0)$. Usando la recurrencia tenemos $T(2) = 3 = t(1)$, y $T(4) = 8 = t(2)$. Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcrcrcl} t(0) & = & C_1 & + & C_2 & & & = & 1 \\ t(1) & = & 2C_1 & + & C_2 & + & C_3 & = & 3 \\ t(2) & = & 4C_1 & + & C_2 & + & 2C_3 & = & 8 \end{array}\right\}$$

Las soluciones son $C_1 = 3$, $C_2 = -2$, y $C_3 = -1$. Por tanto, $t(k) = 3 \cdot 2^k - k - 2$. Deshaciendo el cambio de variable, donde $k = \log_2 n$, obtenemos:

$$t(k) = T(2^k) = T(n) = 3n - \log_2 n - 2 \quad \in \quad \Theta(n)$$

Ejercicio 4 [4 puntos]

Resolved la siguiente relación de recurrencia por el **método de expansión de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \log_2 n & \text{si } n > 1 \end{cases}$$

Posible solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= 2T(n/2) + \log_2 n \\ &= 2\left[2T(n/4) + \log_2(n/2)\right] + \log_2 n \\ &= 4T(n/4) + 2\log_2 n - 2\log_2 2 + \log_2 n \\ &= 4\left[2T(n/8) + \log_2(n/4)\right] + 2\log_2 n - 2\log_2 2 + \log_2 n \\ &= 8T(n/8) + 4\log_2 n - 4\log_2 4 + 2\log_2 n - 2\log_2 2 + \log_2 n \\ &\vdots \\ &= 2^i T(n/2^i) + \log_2 n \cdot \sum_{j=0}^{i-1} 2^j - \sum_{j=1}^{i-1} 2^j \log_2 2^j \end{aligned}$$

El primer sumatorio es una simple serie geométrica (para potencias de dos el resultado es el término siguiente al último, menos el primero). Simplificando, tenemos:

$$T(n) = 2^i T(n/2^i) + (2^i - 1)\log_2 n - \sum_{j=1}^{i-1} j 2^j.$$

El último sumatorio se puede calcular a través de la fórmula que involucra derivar una serie geométrica (que encontraréis en las transparencias de la asignatura). Además, se puede usar otra estrategia (también vista en clase) que consiste en descomponer el sumatorio de la siguiente manera:

$$\begin{array}{ccccccccccc} 1\cdot 2^1 & + & 2\cdot 2^2 & + & 3\cdot 2^3 & + & \cdots & + & (i-1)\cdot 2^{(i-1)} & & \\ \hline 2^1 & + & 2^2 & + & 2^3 & + & \cdots & + & 2^{i-1} & = & 2^i - 2^1 \\ & & 2^2 & + & 2^3 & + & \cdots & + & 2^{i-1} & = & 2^i - 2^2 \\ & & & & 2^3 & + & \cdots & + & 2^{i-1} & = & 2^i - 2^3 \\ & & & & & & & & \vdots & & \\ & & & & & & & & 2^{i-1} & = & 2^i - 2^{i-1} \end{array}$$

Cada fila por debajo de la raya es una serie geométrica, y tendremos que sumar todos los términos de la columna de la derecha. Por un lado hay que sumar 2^i un total de $i-1$ veces, mientras que aparece otra serie geométrica sencilla de potencias de dos. En total tenemos:

$$(i-1)2^i - (2^1 + 2^2 + 2^3 + \cdots + 2^{i-1}) = (i-1)2^i - (2^i - 2) = i2^i - 2\cdot 2^i + 2.$$

Por tanto,

$$T(n) = 2^i T(n/2^i) + (2^i - 1)\log_2 n - i2^i + 2\cdot 2^i - 2.$$

El caso base $T(1)$ se alcanza cuando $n/2^i = 1$. Es decir, cuando $n = 2^i$, e $i = \log_2 n$. Sustituyendo obtenemos la fórmula final:

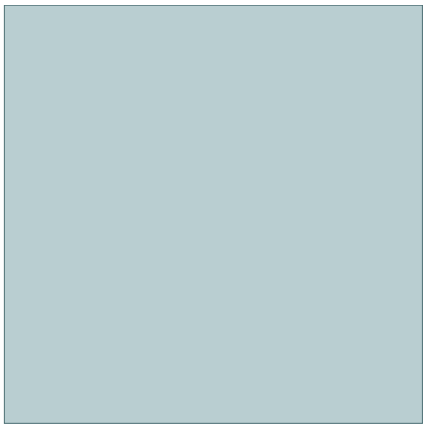
$$T(n) = n + (n-1)\log_2 n - n\log_2 n + 2n - 2 = 3n - \log_2 n - 2.$$

2. Diseño de algoritmos

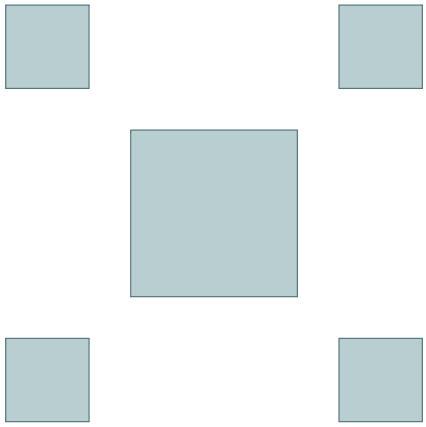
Ejercicio 5 [5 puntos]

Se pide implementar un algoritmo recursivo para generar el fractal de la Figura 1 para un orden n determinado. Para $n = 1$ simplemente se debe dibujar un cuadrado centrado en un punto $\mathbf{p}$ en el plano, cuyo lados horizontal y vertical miden $2s$. Tanto $\mathbf{p}$, como s, como n serán parámetros del algoritmo. A medida que aumenta el orden n se dibujarán más cuadrados. En concreto, a partir de las coordenadas de los nuevos cuadrados añadidos en el fractal de orden $n - 1$, se generarán nuevos cuadrados, como se ilustra en la Figura 2.

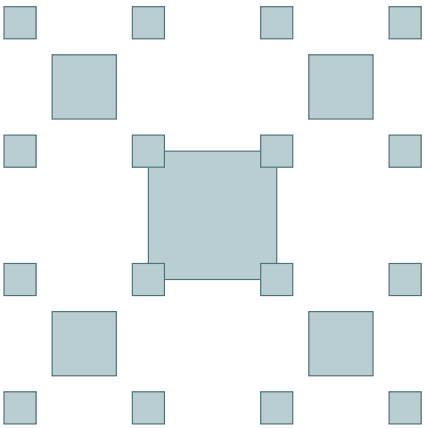
El algoritmo se escribirá en Python. Por simplicidad, podéis considerar que $\mathbf{p}$ es una lista de dos componentes. Además, para dibujar un cuadrado el algoritmo llamará a una función `dibuja_cuadrado` a la que se le pasará como parámetros el punto central $\mathbf{p}$, y la longitud del lado.



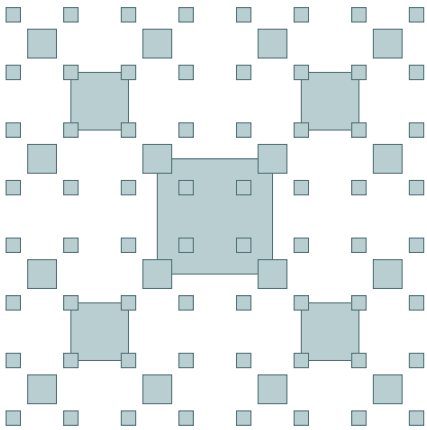
Orden 1



Orden 2



Orden 3



Orden 4

Figura 1: Fractales de orden 1–4.

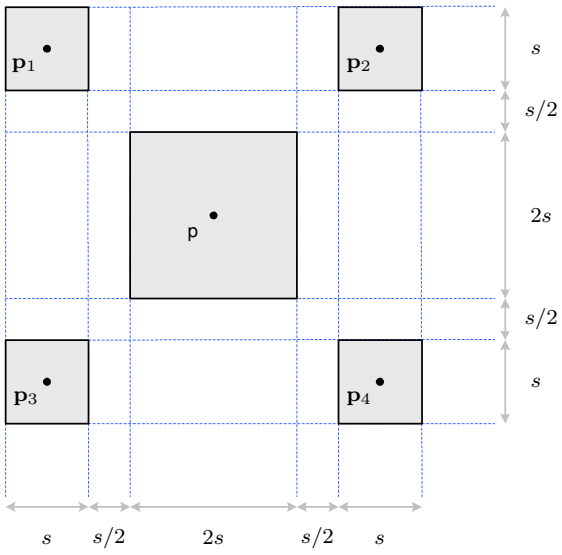


Figura 2: Considérese un cuadrado con centro $\mathbf{p}$ y cuyos lados miden $2s$. El fractal genera cuatro nuevos cuadrados cuyos lados miden s, con centros en $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ y $\mathbf{p}_4$.

Posible solución:

La clave para construir el algoritmo basado en la técnica de divide y vencerás es darse cuenta de que el fractal de orden n se compone de un cuadrado central, y cuatro fractales más sencillos de orden $n-1$ (esto se aprecia claramente en el fractal de orden 4 de la Figura 1). Dado un punto inicial $\mathbf{p}$, los fractales más sencillos están centrados en los puntos $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ y $\mathbf{p}_4$, ilustrados en la Figura 2.

Por tanto, el algoritmo (siempre) debe dibujar el cuadrado central, y cuando $n > 1$ simplemente debe calcular los nuevos puntos centrales ($\mathbf{p}_i$) de los "subfractales", reducir la longitud del lado del cuadrado (usando $s/2$), y el orden en una unidad, y llevar a cabo las cuatro llamadas recursivas que generan los subfractales. El siguiente código ilustra una posible solución:

```
def fractal_squares(ax,p,s,n):

    dibuja_cuadrado(ax,p,2*s)

    if n > 1:
        p1 = [p[0] - 2*s, p[1] + 2*s]
        fractal_squares(ax, p1, s/2, n - 1)

        p2 = [p[0] + 2*s, p[1] + 2*s]
        fractal_squares(ax, p2, s/2, n - 1)

        p3 = [p[0] - 2*s, p[1] - 2*s]
        fractal_squares(ax, p3, s/2, n - 1)

        p4 = [p[0] + 2*s, p[1] - 2*s]
        fractal_squares(ax, p4, s/2, n - 1)
```

El método recibe un parámetro adicional en este ejemplo, para que pueda funcionar con el siguiente código que podéis probar:

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

cedge = (80/255,113/255,118/255)
cface = (185/255,206/255,209/255)

def dibuja_cuadrado(ax,p,l):
    ax.add_patch(Rectangle((p[0]-l/2, p[1]-l/2),
        l, l, facecolor=cface, edgecolor=cedge, linewidth=0.5))


def fractal_squares(ax,p,s,n):

    dibuja_cuadrado(ax,p,2*s)

    if n > 1:
        p1 = [p[0] - 2*s, p[1] + 2*s]
        fractal_squares(ax, p1, s/2, n - 1)

        p2 = [p[0] + 2*s, p[1] + 2*s]
        fractal_squares(ax, p2, s/2, n - 1)

        p3 = [p[0] - 2*s, p[1] - 2*s]
        fractal_squares(ax, p3, s/2, n - 1)

        p4 = [p[0] + 2*s, p[1] - 2*s]
        fractal_squares(ax, p4, s/2, n - 1)


fig = plt.figure()
fig.patch.set_facecolor('white')
ax = plt.gca()
p = [0,0]
s = 1;
n = 5
fractal_squares(ax, p, s, n)
plt.axis('equal')
plt.axis('off')
plt.show()
```

Ejercicio 6 [5 puntos]

Un cuadrado mágico de orden n es una matriz con todos los números enteros de 1 a n^2 tales que la suma de los elementos de todas sus filas, columnas, y diagonales es idéntica. En concreto, a dicha suma se la denomina "constante mágica", y vale:

$$n \cdot \frac{n^2+1}{2}.$$

Por ejemplo, el siguiente cuadrado, de orden 4, es mágico:

1	15	14	4
12	6	7	9
8	10	11	5
13	3	2	16

No se repite ningún número, y la suma de los elementos de todas las filas, columnas, y diagonales es igual a $34 = 4 \cdot (16+1)/2$.

Se pide implementar un algoritmo basado en la técnica de backtracking para calcular el número total de cuadrados mágicos de orden n. La solución parcial se almacenará en una matriz de tamaño $n \times n$. Por otro lado, para acelerar el algoritmo, el método usará dos vectores con la suma parcial de los elementos de las filas y las columnas. También llevará dos parámetros con las sumas parciales de las diagonales. El método **no** llamará a otro para comprobar la validez de una solución parcial. Por último, el método deberá incluir los parámetros necesarios para llevar a cabo la tarea, y no usará variables globales.

Posible solución:

Los algoritmos basados en la técnica de *backtracking* son métodos recursivos de "fuerza bruta" que realizan búsquedas exhaustivas. Para poder terminar en tiempos razonables deben podar los árboles de recursión, introduciendo condiciones que eviten realizar llamadas recursivas que no conduzcan a posibles soluciones. Estas condiciones son fundamentales, pero dependiendo del problema también pueden ser complejas. Para este problema, las condiciones tendrían que ser excesivamente complejas como para obtener un algoritmo eficiente, incluso para $n = 4$. Por este motivo, la solución que se presenta a continuación introduce unas condiciones de poda relativamente simples, que emplean la información en las estructuras de datos que se mencionan en el enunciado. De esta manera, aunque sería posible usar condiciones más complejas, se considera que las que se describen en esta solución son suficientes como para obtener la máxima nota en el ejercicio.

El algoritmo se basa en rellenar la matriz `S` de la solución parcial por filas. En cada llamada, el método recibe la fila `i`, y la columna `j`, donde intentará colocar los candidatos (que son los números de 1 a n^2). El método comienza comprobando si la fila es igual a n. En ese caso el tablero estará lleno, ya que las filas están indexadas desde 0 hasta $n-1$. A continuación se determina si el cuadrado es mágico. Para ello emplea las listas `suma_filas` y `suma_columnas`, que almacenan la suma de los elementos de cada fila y columna de `S`, respectivamente; y `suma_pdiag` y `suma_sdiag`, que almacenan la suma de los elementos en la diagonal principal y secundaria de `S`, respectivamente. En concreto, como los elementos de `suma_filas` y `suma_columnas` nunca serán mayores que `M` (por las condiciones que se usarán en el caso recursivo del algoritmo), es suficiente comprobar `sum(suma_filas)==M*n` y `sum(suma_columnas)==M*n`. Además, naturalmente necesitamos `suma_pdiag==M` y `suma_sdiag==M`. Si se cumplen todas estas condiciones el cuadrado será mágico y podremos devolver un 1, al haber encontrado una solución. En caso contrario se devuelve 0.

Si la solución parcial `S` no está completa pasamos a ejecutar el algoritmo a partir de la línea 11. Primero se inicializa un contador a 0, que aumentará si las llamadas recursivas encuentran soluciones válidas. A continuación tenemos el bucle principal para seleccionar los candidatos a incluir en la solución parcial. Como no puede haber elementos repetidos en la solución, empleamos la lista Booleana `nums_libres` para indicar los elementos que están libres, tal y como hacíamos en el problema de las n-reinas, o la generación de permutaciones. Si el candidato no se ha usado, se introduce en la matriz, y se marca como usado. A continuación se actualizan los parámetros que indican las sumas de los elementos de las filas, columnas, y diagonales. Después el `if` de la línea 25 realiza más podas. Como se ha comentado anteriormente, por sencillez este algoritmo simplemente verifica que la suma de los elementos de la fila `i`, y la de los de la columna `j`, sea menor o igual que `M` (esta condición fuerza a que cada componente de `suma_filas` y `suma_columnas` sea menor o igual a `M`, lo cual se aprovecha en el caso base del algoritmo). Además, la suma de elementos de las diagonales de la matriz tampoco puede ser mayor que `M`.

Posteriormente, si la solución parcial puede ser válida se calcula la nueva fila y columna tras avanzar de izquierda a derecha (o pasar a la siguiente fila si `j` hace referencia a la última columna). Luego se realiza una llamada recursiva, y se suma el resultado obtenido al contador `cont`. Al salir del `if` se deshacen los cambios en las estructuras de datos que llevan las sumas parciales, y se vuelve a marcar el candidato `k` como libre. Finalmente, se devuelve el resultado del contador.

En cuanto al método `cuenta_cuadrados_magicos_wrapper`, inicializa las listas con las sumas parciales a 0, declara una matriz `S` de tamaño $n \times n$ (la inicializa con ceros aunque este valor es irrelevante), y crea una lista Boolean de longitud n^2 inicializada con valores `True` (para indicar los candidatos que están libres para ser incluidos en la solución parcial).

Por último, para $n = 3$ hay 8 cuadrados mágicos, mientras que para $n = 4$ hay 7040. Este algoritmo halla la solución para $n = 3$ en tiempo real, pero puede tardar más de una hora en un PC actual para hallar la solución para $n = 4$.

```
def cuenta_cuadrados_magicos(i,j,nums_libres,suma_filas,suma_columnas,
                             suma_pdiag,suma_sdiag,S,M):
    n = len(S)
    if i==n:
        if sum(suma_filas)==M*n and sum(suma_columnas)==M*n and \
            suma_pdiag==M and suma_sdiag==M:
            return 1
        else:
            return 0
    else:
        cont = 0
        for k in range(1,n*n+1):
            if nums_libres[k-1]:
                S[i][j] = k

                nums_libres[k-1] = False

                suma_filas[i] = suma_filas[i] + k
                suma_columnas[j] = suma_columnas[j] + k
                if i==j:
                    suma_pdiag = suma_pdiag + k
                if i+j==n-1:
                    suma_sdiag = suma_sdiag + k

                if suma_filas[i]<=M and suma_columnas[j]<=M and \
                    suma_pdiag<=M and suma_sdiag<=M:

                    if j==n-1:
                        i_nueva = i+1
                        j_nueva = 0
                    else:
                        i_nueva = i
                        j_nueva = j+1

                    cont = cont + cuenta_cuadrados_magicos(
                        i_nueva, j_nueva, nums_libres, suma_filas,
                        suma_columnas, suma_pdiag, suma_sdiag, S, M)

                suma_filas[i] = suma_filas[i] - k
                suma_columnas[j] = suma_columnas[j] - k
                if i==j:
                    suma_pdiag = suma_pdiag - k
                if i+j==n-1:
                    suma_sdiag = suma_sdiag - k

                nums_libres[k-1] = True

        return cont


def cuenta_cuadrados_magicos_wrapper(n):
    suma_filas = [0]*n
    suma_columnas = [0]*n

    # Matriz que almacena las soluciones parciales
    S = [None]*n
    for i in range(0,n):
        S[i] = [0]*n

    nums_libres = [True]*(n*n)

    return cuenta_cuadrados_magicos(0,0,nums_libres,suma_filas,suma_columnas,
                                    0,0,S,n*(n*n+1)/2)


print(cuenta_cuadrados_magicos_wrapper(3))
```