Teoría de la Recursividad
Capítulo 6: Apuntes finales al análisis de los problemas anteriores
Consideraciones finales.

La búsqueda recursiva de las soluciones de un problema, es, esencialmente, una estrategia de "tanteo". Cuando carecemos de un algoritmo determinado que sea capaz de obtener directamente la o las soluciones, siempre queda el recurso de generar las combinaciones posibles dentro de las condiciones del problema, y verificar cuáles de esas combinaciones son realmente soluciones.

Esta manera de ver las cosas tiene una contra-partida extremadamente importante: el tiempo necesario para obtener la o las soluciones.

En este comentario la idea de "tiempo" está referida exclusivamente al tiempo de EJECUCION del programa. Ese "tiempo" - siempre mucho mayor que el de ejecución de un programa que pudiera ir directamente a la obtención de la o las soluciones - puede variar entre límites muy grandes: desde unos cuantos segundos o algunos minutos, hasta muchos días, o aún más, de ejecución del programa.

El lector poco familiarizado con estas ideas seguramente se sorprenderá de la magnitud de la cantidad de combinaciones posibles en problemas tan sencillos como los que hemos analizado hasta el momento. En ese "universo", cantidades como millones, decenas de millones o miles de millones son muy frecuentes. Por mucho que pensemos en modernas y velocísimas computadoras, cantidades como esas llevan inevitablemente a tiempos de ejecución totalmente inaceptables desde el punto de vista práctico.

Es esencial, por tanto, programar con criterios muy claros para disminuir el tiempo de ejecución todo lo posible.

En principio, si el programa genera TODAS las combinaciones posibles sin dudas generará entre ellas las que sean solución. Pero en este caso, el tiempo de ejecución será el máximo... o incluso infinito. Cualquier solución que se base solamente en esa consideración, puede demorar horas, días o meses ejecutándose en la máquina, a veces, inútilmente. Es por ello que hay que seguir varios criterios para disminuir ese tiempo:

a) Tomar en cuenta la mayor cantidad de restricciones posibles al generar cada combinación, de modo de disminuir al máximo la cantidad de combinaciones que deberá generar y analizar la computadora.
b) Evitar o disminuir al máximo posible la repetición de combinaciones obtenidas por diferentes vías. Si las repeticiones no se pueden evitar, al menos hay que detectarlas y dar "marcha atrás".
c) Controlar tan pronto como sea posible, si en la combinación que se está generando hay características que contradigan algunas de las condiciones del problema, para descartar el paso dado y probar con otro.

Estos criterios quedarán mucho más claros si se ilustran con algunos ejemplos. Nos apoyaremos en los programas vistos en el capítulo 4 y 5, para aportar elementos sobre el tema.

Criterio a):

El “problema de las 8 damas” ilustra con claridad este criterio.

En principio, se trata de colocar 8 damas en las 64 casillas de un tablero de ajedrez. Si no tuviéramos en cuenta otras condicionantes del problema - excepto la muy evidente de que en una misma casilla no se puede colocar más de una dama - la cantidad de “combinaciones” posibles surge del siguiente razonamiento: Para la colocación de la primera dama hay 64 casillas disponibles; colocada ésta en cualquiera de ellas, quedan 63 casillas libres para ubicar la segunda; y así sucesivamente. Llegaríamos a un total de “combinaciones” posibles de 64*63*62*61*60*59*58*57, un número de alrededor de 200 mil millones de combinaciones. Una computadora muy rápida, capaz de obtener 10 MIL de esas combinaciones POR SEGUNDO, y trabajando día y noche ininterrumpidamente, llevaría más de MEDIO AÑO tan sólo para obtenerlas todas... si además, tuviera que analizarlas, ese tiempo se puede multiplicar fácilmente por 3 o más...

Pero en nuestro programa, hemos tenido en cuenta el hecho elemental de que siempre que haya dos damas en la misma fila, ambas se amenazan mutuamente, y por tanto, desde el principio hemos eliminado esta posibilidad, exigiendo que en cada fila sólo pueda situarse UNA dama. Con ello, hemos reducido la cantidad de combinaciones a un máximo de 16,7 millones (una cantidad más de 100 mil veces menor), lo que significa directamente que estamos reduciendo el tiempo de ejecución en 100 mil veces.

Para que esta formidable reducción de tiempo resulte más clara, baste decir que, con la variante de colocar una sola dama en cada fila, hemos reducido a UN SEGUNDO lo que en la variante anterior nos llevaría 28 DIAS de ejecución interrumpida.

No obstante, la cantidad de casi 17 millones de combinaciones que aún mantenemos, sigue siendo demasiado grande para la velocidad de las computadoras. Ya volveremos sobre este ejemplo y esta cifra, al analizar el criterio c).

Sobre el criterio b):

Para ilustrar este caso, pensemos ahora en el problema de las “cajas chinas”. En el mismo, no es posible prever ninguna variante de optimización para generar nuevas transformaciones. A partir de la configuración inicial de letras que constituyen el “dato”, cualquiera de las 6 transformaciones posibles es buena para avanzar en la solución, y no hay criterio “a priori” para descartar ninguna.

Se trata de uno de esos problemas en los que hay que ir generando TODAS las combinaciones posibles. Sin embargo - y esto se analiza al estudiar el programa de solución - un simple ejemplo nos muestra que es posible obtener, en muy pocos pasos, una combinación que ya habíamos encontrado anteriormente. La repetición de una combinación es una dificultad de consecuencias decisivas: Si de una combinación determinada se puede llegar a repetirla en cierta cantidad de pasos, no sólo estamos desperdiciando los pasos intermedios, sino que inevitablemente, al cabo de esa cantidad de pasos volveremos a obtenerla, y así sucesivamente, transformando nuestro programa en una sucesión infinita de pasos inútiles.

Para mayor claridad, repetimos aquí el ejemplo analizado en aquel momento (en el que siempre se desplaza hacia “el hueco” la pareja de le tras situada más a la izquierda):

```
Dato:       abba--abab
            **
   Paso 1   --baababab
              **
   Paso 2   ba--ababab
           **
   Paso 3   --baababab   (y se repite el resultado obtenido en el paso 1).
```

Nada impide, por otra parte, que partiendo de diferentes combinaciones se llegue finalmente, por diferentes transformaciones intermedias, a una misma combinación, con las mismas consecuencias de “ejecución infinita”.

La solución a este inconveniente decisivo es, como ya vimos, ir guardando en una “pila” las combinaciones que vamos logrando, y comparar cada nueva combinación con todas las anteriores (incluyendo por supuesto el “dato”), de modo de evitar en todo momento una repetición.

El lector podrá pensar que, en el ejemplo de que hablamos, ese “chequeo” de repetición de combinaciones también requiere un tiempo de ejecución, y que lo que se gana por un lado, se pierde por otro. Pero ésta es una reflexión que, aunque pueda parecer muy lógica, no resiste el menor análisis: evitando las repeticiones, se puede llegar siempre a la solución del problema; sin evitarlas, necesariamente en alguna variante se caerá en una repetición, y por tanto, en una ejecución “infinita”.

Veamos, sobre el punto c):

Volvemos al “problema de las 8 damas”. Lo habíamos dejado en el punto en que en cada fila sólo nos permitiremos colocar una dama. Una dama en una fila puede ocupar, en principio, 8 posiciones.

Tomando en cuenta tan solo esa restricción, se podría generar un total de 8 elevado a 8 combinaciones diferentes, o sea más de 16,7 millones de combinaciones, como vimos anteriormente. Si esperáramos a completar cada una de ellas para analizar si la combinación es admisible o no, el tiempo de ejecución del programa sería de todas maneras enorme.

Intentemos lograr una idea de cuánto demoraría:

Supongamos que tenemos un programa tan simple y una computadora tan buena como para lograr generar y analizar 1000 combinaciones por segundo. Entonces, tendríamos una ejecución que duraría 16700 segundos... o sea, más de 4 horas y media. En cambio, si el lector ha comprobado la solución a ese problema que aparece en las páginas anteriores, habrá constatado que aún en la computadora más lenta y más limitada, las 92 soluciones se obtienen en un tiempo brevísimo, de tan solo unos pocos segundos.

Esta ganancia en velocidad de ejecución se obtiene simplemente por que no se espera a completar cada combinación para analizarla, sino que se rechaza, desde la segunda fila en adelante, toda posibilidad de combinación que no pueda llegar a ser solución al completarse. (Esta es, en esencia, la finalidad de la función “Posible”, que impide continuar con una combinación que, al empezar a formarse, ya incumple con las condiciones exigidas para ser solución)

Otro breve cálculo ayuda a formarse una idea de cuánto tiempo inútil se ahorra por esta vía:

Supongamos que comenzamos con la dama de la fila 1 en la columna 1 y nuestro programa “permite” que en la fila 2 también se coloque la dama en la columna 1. (Obviamente, ambas damas se amenazan mutuamente por estar en la misma columna, y cualesquiera sean las posiciones de las restantes 6 damas, la combinación que se obtenga no puede ser solución). De generar todas las combinaciones posibles que contengan una sola dama en cada fila, ¿cuántas de ellas tendrían origen en esa inaceptable ubicación inicial? Pues nada menos que 8 elevado a la 6, o sea más de 262 mil combinaciones, totalmente inútiles porque se originan en una posición de las dos primeras damas que contradice las condiciones del problema. Es más, hay casi 6 millones de combinaciones de posiciones que, por la ubicación tan solo de las dos primeras damas, ya no pueden ser una solución... Para ese trabajo inútil, y tan fácil de evitar, estaríamos dedicando casi 2 horas de trabajo de nuestra veloz computadora.

Aún hay otra cuestión que es necesario analizar sobre este tema:

Con frecuencia, los problemas propuestos son casos particulares de problemas mucho más generales: El problema “de las 8 damas” es un caso particular de lo que sería el “problema de las N damas” (por supuesto, en un tablero de N x N casillas); el de las “cajas chinas” podría plantearse no ya con una cadena de longitud 10, sino de longitud N. Otras veces, los problemas se proponen a partir de N datos.

En estos casos, es frecuente que el programador se conforme con probar su programa con unos pocos datos, o sea, con valores de N relativamente pequeños y deduzca de ello, erróneamente, que la velocidad de ejecución lograda es satisfactoria. Pero este es un criterio de prueba muy peligroso. Con relación al número de combinaciones posibles, el mismo no es en general proporcional al “tamaño” de N, sino de una potencia de N que depende de las características de cada problema. Este crecimiento “exponencial” de la demora en la ejecución de un programa, resulta generalmente “enmascarado” o pasa inadvertido cuando se prueba el programa sólo con valores pequeños de N. Y así una demora que podría ser aceptable para N = 5 ó N = 6 se vuelve absolutamente inadmisible para valores más grandes de N.

Veamos un ejemplo sumamente sencillo que ilustra este aspecto con mucha claridad:

En un tablero de N x N se sitúa un rey en una casilla de la 1ra. Fila. El rey debe ir pasando por cada fila, hasta llegar a la última, y para pasar de una fila a la siguiente tiene 3 posibilidades: o bien "baja" por la misma columna, o bien "baja" en diagonal, ya sea hacia la izquierda o hacia la derecha. Por supuesto, en las dos casillas extremas de cada fila no tiene 3 posibilidades, sino solamente 2. ¿Cuántos caminos diferentes puede recorrer para "bajar" desde la primera fila hasta la última?

Para N = 2, hay sólo 4 caminos;
Con N = 3, hay 17 caminos;
Para N = 4, hay 68 caminos;
...
Si N = 7, los caminos son ya 3387.
...
Para N = 12, en fin, hay exactamente 1 515 926 caminos.

Este ejemplo, entre miles, debería ser suficiente para alertar a los programadores sobre el peligro de probar los programas para valores "pequeños" de N... y darse por satisfechos con el resultado...

Los cálculos presentados, que suelen ser más complicados y requerir buenos conocimientos de teoría combinatoria, son suficientemente expresivos en cuánto al cuidado extremo que hay que tener para evitar que el programa sea en principio correcto, pero demasiado lento para obtener las soluciones.

Así pues, nuestra estrategia no podrá ser nunca la de generar todas las combinaciones posibles, según el enunciado. Más bien, lo que debemos lograr desde el principio, (y para ello habrá que extremar nuestro análisis), es la MENOR CANTIDAD de combinaciones que nos faciliten encontrar las soluciones del problema.

Este aspecto es el que marca la diferencia entre los que han logrado dominar la herramienta recursiva y aquellos que simplemente empiezan a conocerla.

CONSIDERACIONES FINALES.

En el diseño:

a) Tomar en cuenta la mayor cantidad de restricciones posibles al generar cada combinación, de modo de disminuir al máximo la cantidad de combinaciones que deberá generar y analizar la computadora.
b) Evitar o disminuir al máximo posible la repetición de combinaciones obtenidas por diferentes vías. Si las repeticiones no se pueden evitar, al menos hay que detectarlas y dar "marcha atrás".
c) Controlar tan pronto como sea posible, si en la combinación que se está generando hay características que contradigan algunas de las condiciones del problema, para descartar el paso dado y probar con otro.

En la programación:

d) No "llene" el programa con el detalle de cada acción necesaria. Utilice subprogramas, para dejar más "a la vista" los aspectos esenciales del algoritmo... Pero no se vaya al otro extremo: Un programa excesivamente fraccionado, también pierde claridad y no deja percibir el conjunto. Un criterio práctico, es el de tratar que cada subprograma no sobre pase mucho el tamaño que puede apreciarse en una pantalla. Para que su análisis y depuración sea más fácil.
e) En los métodos auxiliares, declare variables locales para todo el trabajo auxiliar propio del método. De no proceder así, se corre el riesgo de alterar valores que luego utilizará otro método o el programa principal. Los errores que se originan por esta causa son EXTREMADAMENTE DIFICILES de encontrar.

En la puesta a punto:

f) Comience trabajando con un juego de datos relativamente "pequeño", según las condiciones del problema. Generalmente, los enunciados de problemas traen algún juego de datos como ejemplo. De ser así, tome éste para comenzar. Cerciórese de que los datos no contengan errores. Ese juego de datos debe declararse como constante para que no lo esté tecleando continuamente. Esto ahorrará considerable tiempo y trabajo en la puesta a punto.
g) Después que el programa funcione satisfactoriamente para el juego de datos inicial, pruebe con nuevos juegos de datos más complejos. ES FRECUENTE, que los juegos de datos que vienen como ejemplo NO PRESENTEN las situaciones más problemáticas de la tarea propuesta. Además, el propio trabajo previo de programación que se ha hecho hasta ese momento, ayuda a crear datos "más exigentes" para el programa que se esté probando.
h) SOLO ENTONCES, dedique tiempo a la lectura y validación de los datos. Detecte la mayor cantidad de errores posible.