Ficha 32

Estrategias de Resolución de Problemas: Algoritmos Ávidos, Programación Dinámica y Backtracking

1.] Introducción.

La tarea de encontrar y plantear un algoritmo que resuelva un problema dado (cosa que debe hacer casi todo el tiempo un programador) requiere conocimientos en el campo de ese problema, pero también paciencia, creatividad, ingenio, disciplina para realizar pruebas, capacidad de autocrítica… y aún así en muchos casos encontrar una solución no es un proceso simple ni obvio. A veces un programador encuentra una solución que parece correcta pero luego descubre fallas o demuestra que la misma no es correcta para todos los casos. Otras veces se encuentra una solución, pero luego se comprueba que la misma es poco eficiente (ya sea porque su tiempo de ejecución es alto o porque ocupa demasiada memoria, por ejemplo). Y muchas veces la solución simplemente no aparece [1].

En este sentido, a lo largo de los años los investigadores y los especialistas en el área de los algoritmos han estudiado y propuesto diversas técnicas básicas que ayudan a plantear algoritmos para situaciones específicas. Estas técnicas *no implican una garantía de éxito*, sino simplemente un punto de partida para intentar encontrar una solución frente a un problema particular. Si el programador conoce lo suficiente acerca del problema, y domina los principios de aplicación de cada técnica, *posiblemente* podrá entonces plantear alguna de ellas para intentar dar con una solución al problema. Nada garantiza que lo logre, y si lo logra, nada garantiza que la solución encontrada sea eficiente, pero incluso así el programador habrá tenido elementos para comenzar a trabajar y al menos poder descartar algunos caminos.

Tradicionalmente, las técnicas o estrategias de planteo de algoritmos que se sugieren como básicas son las siguientes:

- ***Fuerza Bruta***: Consiste en explorar y aplicar sistemáticamente, una por una, todas y cada una de las posibles combinaciones de solución para el problema dado. A modo de ejemplo, el algoritmo de *ordenamiento de Selección Directa* es esencialmente un algoritmos de *fuerza bruta*. Por lo general, los algoritmos obtenidos por *fuerza bruta* resultan intuitivamente simples de comprender e implementar (de hecho, un algoritmo de fuerza bruta suele ser lo primero que se le ocurre a un programador) pero por otra parte suelen ser muy ineficientes (o bien son muy lentos si el tamaño de la entrada es grande, o bien ocupan demasiados recursos de memoria) ya que la esencia del planteo consiste justamente en no ahorrar pasos.

  En la medida de lo posible, si se tiene un algoritmo de *fuerza bruta* para un problema, el programador debería esforzarse en refinarlo, eliminar elementos de redundancia lógica o agregar elementos que permitan ahorrar trabajo y tratar de obtener una solución mejor. Sin

embargo, es notable que para muchos problemas muy estudiados y comunes, sólo se conocen soluciones generales de *fuerza bruta*, aunque para casos especiales de esos problemas se aplican estrategias de optimización que permiten mejorar el rendimiento de la solución. Un conocido ejemplo de esta situación, es el famoso *Problema del Viajante* (o del *Vendedor Ambulante*): dadas *n* ciudades y las distancias entre ellas, establecer un recorrido que permita pasar una sola vez por todas ellas pero recorriendo la menor distancia total. La solución trivial de fuerza bruta enumera todos los posibles recorridos y se queda con el más corto, pero eso lleva a un tiempo de ejecución *O(n!)*, que lo vuelve inaplicable si el número de ciudades se hace mayor a 20. Hasta hoy, una de las mejores soluciones se basa en técnicas de *Programación Dinámica* con tiempos de ejecución $O(n^2 2^n)$. No se conoce una solución general más eficiente para todos los casos de este problema, pero se pueden aplicar estrategias que mejoren la situación si se sabe que el número de ciudades es pequeño, o *estrategias subóptimas* que obtienen soluciones aproximadas aunque no necesariamente óptimas.

- ***Recursión***: Como sabemos, la *recursión* o *recursividad* es la propiedad que permite que un proceso se invoque a si mismo una o más veces como parte de la solución. Si se establece que un problema puede ser entendido en base a versiones más pequeñas y manejables de si mismo, entonces un *planteo recursivo* puede ayudar a lograr un algoritmo que normalmente será muy claro para entender e implementar, al costo de utilizar cierta cantidad de memoria adicional en el segmento de stack del computador (y algo de tiempo extra para gestionar el apilamiento en ese stack). En muchos casos, este costo no es aceptable y es preferible plantear un *algoritmo no recursivo* que resuelva el mismo problema. Muchos ejemplos conocidos que se usan para explicar el funcionamiento de la *recursividad* (cálculo del factorial de un número, cálculo de un término de la secuencia de Fibonacci, etc.) son justamente casos en los que la *recursión* ***no*** es aconsejable.

  A pesar de los costos extra en uso de memoria y tiempo de ejecución, para algunos problemas de lógica compleja la recursión constituye una herramienta que posibilita el planteo lógico en forma clara y concisa, frente a planteos no recursivos extensos, intrincados y sumamente difíciles de mantener frente a cambios en los requerimientos. Casos así, por ejemplo, se dan en algoritmos (que hemos visto) para generar *gráficas fractales* o en situaciones más conocidas como la implementación de *algoritmos de inserción y borrado en árboles de búsqueda equilibrados*.

- ***Vuelta Atrás (Backtracking)***: Se trata de una técnica que permite explorar en forma incremental un conjunto de potenciales soluciones (*soluciones parciales*) a un problema, de manera que si se detecta que una *solución parcial* ***no puede*** ser una solución válida, se la descarta junto a todas las candidatas que podrían haberse propuesto a partir de ella. Típicamente, se implementa mediante *recursión* generando un árbol de invocaciones recursivas en el que cada nodo constituye una solución parcial. Como cada nueva invocación recursiva agrega un nodo en el nivel siguiente del árbol por la inclusión de un nuevo paso simple en el proceso, entonces es fácil ver que si se llega a un punto en el cual se obtiene una solución no válida, puede anularse toda la rama que llevó a esa solución y continuar el análisis en ramas vecinas. Cuando es aplicable, el *backtracking* suele ser más eficiente que la enumeración por *fuerza bruta* de todas las soluciones, ya que con *backtracking* pueden eliminarse muchas soluciones sin tener que analizarlas.

  La estrategia de *backtracking* se usa en problemas que admiten la idea de solución parcial, siempre y cuando se pueda comprobar en forma aceptablemente rápida si una solución parcial es válida o no. Ejemplos muy conocidos de aplicación se dan con el problema de las *Ocho Reinas* (tratar de colocar ocho reinas en un tablero de ajedrez, sin que se ataquen entre ellas), o en el planteo de soluciones para juegos de *Palabras Cruzadas*, o en el planteo de tableros de *Sudoku* o en general, en problemas de *optimización combinatoria*.

- ***Algoritmos Ávidos o Devoradores (Greedy Algorithms)***: Un *algoritmo ávido* es aquel que aplica una regla intuitivamente válida (una *heurística*) en cada paso local del proceso, con la *esperanza* de obtener finalmente una solución global óptima que resuelva el problema original. Para muchos problemas esta estrategia no produce una solución óptima, pero suele servir de todos modos para intentar plantear soluciones óptimas locales, que luego puedan aproximar una solución óptima final en un tiempo aceptable.

  La ventaja de un *algoritmo ávido* (cuando es correcto) es que por lo general lleva a una solución simple de entender, muy directa de implementar y razonablemente eficiente. Pero la desventaja es que como no siempre lleva a una solución correcta, se debe realizar una *demostración de la validez* del algoritmo que podría no ser sencilla de hacer. En general, si se puede demostrar que una *estrategia ávida* es válida para un problema dado, entonces la misma suele producir resultados más eficientes que otras estrategias como la *programación dinámica*.

  Algunos ejemplos de aplicación conocidos en los cuales un *algoritmo greedy* se aplica con éxito son los algoritmos de *Prim* y de *Kruskal* para obtener el *árbol de expansión mínimo de un grafo*, el algoritmo de *Dijkstra* para obtener *el camino más corto entre nodos de un grafo*, y el algoritmo de *Huffman* para construir árboles que permitan obtener la *codificación binaria óptima para una tabla de símbolos*, de forma de reducir el espacio ocupado por un mensaje armado con esos símbolos.

- ***Divide y Vencerás***: Consiste en tratar de dividir el lote de datos en dos o más subconjuntos de tamaños aproximadamente iguales, procesar cada subconjunto por separado y finalmente unir los resultados para obtener la solución final. Normalmente, se aplica *recursión* para procesar a cada subconjunto y el proceso total podrá ser más o menos eficiente en cuanto a tiempo de ejecución dependiendo de tres factores: la *cantidad de invocaciones recursivas* que se hagan, el *factor de achicamiento del lote de datos* (por cuanto se divide al lote en cada pasada) y el *tiempo que lleve procesar en forma separada a un subconjunto*. Ejemplos muy conocidos de aplicación exitosa de la técnica de *divide y vencerás*, son los algoritmos *Quicksort* y *Mergesort* para ordenamiento de arreglos.

- ***Programación Dinámica***: Esta técnica sugiere almacenar en una tabla las soluciones obtenidas previamente para los subproblemas que pudiera tener un problema mayor, de forma que cada subproblema se resuelva sólo una vez y luego simplemente se obtengan sus soluciones consultando la tabla si esos subproblemas volvieran a presentarse. Esto tiene mucho sentido: en muchos planteos originalmente basados (por ejemplo) en *divide y vencerás*, suele ocurrir que al dividir un problema en subproblemas se observe que varios de estos últimos se repiten más de una vez, con la consecuente pérdida de tiempo que implicaría el tener que volver a resolverlos.

  La idea de la *programación dinámica* es entonces resolver primero los subproblemas más simples, y luego ir usando esas soluciones hacia arriba para combinarlas y resolver problemas con entradas mayores. Esto es especialmente útil en problemas en los que el número de subproblemas que se repiten crece en forma exponencial. Se aplica en *problemas de optimización*: encontrar la mejor solución posible entre varias alternativas presentes. Algunos ejemplos en los que la programación dinámica se aplica con éxito son el algoritmo de *Bellman-Ford* y el algoritmo de *Floyd-Warshall* para encontrar el *camino más corto entre dos vértices de un grafo*, o los diversos algoritmos conocidos para *alineación de secuencias*, cuyo objetivo es encontrar la *mínima cantidad de cambios que deben hacerse en una secuencia de entrada para convertirla en otra*.

- **Randomización (Algoritmos de Base Aleatoria)**: Las estrategias para planteo de algoritmos que hemos enumerado hasta aquí no son las únicas posibles, pero son quizás las más conocidas. Todas ellas se basan en el intento de arribar a un algoritmo (más o menos

eficiente) que resuelva un problema, y en todas la idea es que ese algoritmo sea *determinista*. La característica de un algoritmo *determinista* es que ante una misma entrada, produce siempre la misma salida. Cada paso que el algoritmo aplica es una consecuencia lógica del estado en que se encontraba en el paso anterior y por lo tanto, un algoritmo determinista correctamente planteado siempre produce la solución correcta al problema analizado.

Está claro que siempre queremos que un algoritmo sea correcto, pero también esperaríamos que un algoritmo sea *eficiente* al menos en su tiempo de ejecución. Sin embargo, muchos problemas no son tan simples. Existen problemas que recurrentemente aparecen y requieren soluciones prácticas, pero no siempre esas soluciones prácticas son eficientes. Si la cantidad de posibles soluciones fuese exponencial (por ejemplo), un *algoritmo determinista de Fuerza Bruta* tendría un tiempo de ejecución exponencial ($O(2^n)$) lo cual es inaplicable incluso para valores de n pequeños.

Frente a estos casos existe la alternativa de plantear una *estrategia de Randomización* (o *de Aleatorización*). La idea es intentar plantear algoritmos que ya *no sean deterministas*, sino *de base aleatoria*: ahora, ante la misma entrada, el algoritmo podría producir salidas diferentes ya que el siguiente paso a aplicar surge de algún tipo de *selección aleatoria*. Y está claro que si interviene el azar, entonces es posible que el algoritmo no llegue eventualmente a una solución correcta.

Lo anterior implica que si se piensa seriamente en aplicar un *algoritmo randomizado* para un problema dado, ese algoritmo debe ser cuidadosamente analizado desde varios aspectos:

- Por lo pronto, debe quedar claro que el algoritmo valga la pena en cuanto a tiempo de ejecución esperado (por ejemplo, si obtiene un tiempo de ejecución polinómico o que combine una expresión polinómica con otra logarítmica) Si se aplica un *algoritmo randomizado* que de todos modos tendrá un tiempo de ejecución exponencial, no habrá ningún beneficio y todavía se tendrá la desventaja de eventualmente no obtener la solución correcta.
- Si bien se sabe que un *algoritmo randomizado* podría no obtener la solución correcta, se espera poder conocer con precisión cuál es la *probabilidad de que el algoritmo falle*, y diseñarlo de tal modo que esa *probabilidad sea realmente baja o muy baja*.
- Y finalmente, en la medida de lo posible, se esperaría que el algoritmo planteado sea relativamente *simple de implementar* (cosa que suele ser el caso de los *algoritmos randomizados*).

2.] Conceptos básicos sobre *Algoritmos Ávidos* y *Programación Dinámica*.

Como dijimos, los investigadores y programadores han caracterizado algunas técnicas generales que a menudo llevan a algoritmos eficientes para la resolución de ciertos problemas. En fichas anteriores hemos aplicado con mayor o menor profundidad técnicas de *fuerza bruta* y de *recursividad* para resolver algunos problemas y situaciones prácticas (en general, son algoritmos de *fuerza bruta* los que hemos empleado para buscar el menor o el mayor en un arreglo y los algoritmos de ordenamiento simples, mientras que son algoritmos *recursivos* los que hemos visto para generar gráficas fractales) En esta ficha en particular, haremos una introducción a los principios básicos de los *algoritmos ávidos* y de la *programación dinámica*.

Sin embargo, es bueno recordar que existen algunos problemas para los cuales ni éstas ni otras técnicas conocidas han producido hasta ahora soluciones eficientes. Cuando se encuentra algún problema de este tipo suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta y más difícil de calcular.

Un conocido problema que suele usarse para mostrar la aplicación de los *algoritmos ávidos* y de la *programación dinámica*, es el *problema del cambio en monedas*, que enunciamos de la siguiente forma:

Problema 63.) *Problema del Cambio de Monedas: Se trata de plantear un programa que pueda calcular la **mínima cantidad de monedas** en la que puede cambiarse una cierta cantidad **x** de centavos, conociendo los valores nominales de las monedas disponibles. Supongamos que disponemos de un conjunto de monedas **coins** de 1, 5, 10 y 25 centavos. Nos dan una cantidad **x** de centavos, y nos piden calcular la mínima cantidad de monedas de **M** en la que puede expresarse (o cambiarse) el valor **x.***

*Así, si **x = 25** entonces la mínima cantidad de monedas de **coins = {1, 5, 10, 25}** que podemos usar es 1, ya que disponemos de una moneda de exactamente 25 centavos. Si **x = 31**, entonces la mínima cantidad de monedas es 3: una de 25, una de 5 y una más de 1 centavo.*

Se pide explorar distintas posibilidades algorítmicas para este problema, y plantear los programas correspondientes.

Discusión y solución: Podemos dar diversas soluciones a este problema, basadas en distintas estrategias algorítmicas. Una forma muy intuitiva (*que no siempre funciona*), consiste en elegir sistemáticamente la moneda de mayor valor, devolver tantas de ella como se pueda, y luego continuar así con las monedas de mayor valor que siguen hasta cubrir el valor pedido. Así, si el conjunto de valores de monedas disponible es *coins = [1, 5, 10, 25]* y el valor a cambiar es *x = 63*, elegimos dos veces la moneda de 25, una vez la de 10 y tres la de 1, lo cual da un total de 6 monedas (que efectivamente, en el contexto planteado es la mínima cantidad de monedas para llegar a cubrir *x = 63*).

Note que es exigible que la moneda de valor 1 exista para que el problema tenga solución, pues de otro modo habría valores de *x* que serían imposibles de cambiar. Si las monedas disponibles son las que hemos supuesto para el conjunto *coins*, se puede probar que esta forma de trabajar *resolverá siempre el problema en forma correcta*, y constituye un ejemplo de la estrategia de resolución de problemas conocida como *algoritmos ávidos* o *devoradores* (o *greedy algorithms*) [2].

En términos generales, un conjunto *coins* de valores monetarios que siempre produce una *solución óptima* para el problema del cambio de monedas (es decir, siempre logra encontrar la menor cantidad posible de monedas para cambiar cualquier valor $x > 0$) aplicando la regla ávida que hemos indicado, se designa como un *sistema de monedas canónico*. Los sistemas monetarios como el de Argentina, Estados Unidos y Europa (por ejemplo) son *canónicos*: con todos ellos, el *algoritmo ávido* descripto en esta sección funcionará correctamente.

Como hemos indicado, un *algoritmo ávido* es aquel en el que en todo momento se toma la decisión que localmente parece la más apropiada u óptima para el caso, y se aplica esa regla una y otra vez sin importar ni analizar si esa decisión traerá malas consecuencias en el futuro, y sin posibilidad de volver atrás y cambiar la decisión en ningún momento.

También hemos visto que la ventaja de las estrategias ávidas es que suelen ser simples de plantear y de entender. Normalmente se ajustan en forma directa al modelo de solución mental que se tiene del problema, por lo que intuitivamente suelen ser las estrategias preferidas si pueden aplicarse. Lo malo, es que no siempre funcionan: si en el problema del cambio de monedas dispusiéramos también de una moneda de 21 centavos en el conjunto *coins*, podemos ver que la estrategia de *algoritmo ávido* que hemos empleado para el cambio de *x* = 63 seguiría informando que el mínimo es de 6 monedas, cuando es obvio que ahora la solución óptima es 3. En este caso, el conjunto coins = [1, 5, 10, 21, 25] no *es canónico*, ya que nuestro *algoritmo ávido* no funcionará [2].

En todo caso, el problema con las *reglas ávidas* es que debemos demostrar que funcionan para toda combinación posible de entradas, y esa demostración no siempre es simple de hacer. Suponiendo que el conjunto *coins* de valores de monedas *es canónico*, entonces el modelo *avido.py* incluido en el proyecto *[F32] Avidos-Dinamica-Backtraking* que acompaña a esta ficha resuelve el problema en forma óptima aplicando la regla ávida que hemos analizado. Se muestra a continuación:

```
__author__ = 'Cátedra de AED'


def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tSe pidio mayor a', lim, '... cargue de nuevo...')
    return n


def insertion_sort(v):
    n = len(v)
    for j in range(1, n):
        y = v[j]
        k = j - 1
        while k >= 0 and y > v[k]:
            v[k+1] = v[k]
            k -= 1
        v[k+1] = y


def greedy_change(x, coins):
    # chequear si existe una moneda única que sea la solución...
    for value in coins:
        if value == x:
            return 1

    # si no existe, ordenar coins de MAYOR a MENOR...
    insertion_sort(coins)

    # ... y aplicar la regla ávida...
    count, amount = 0, 0
    for value in coins:
        while amount + value <= x:
            amount += value
            count += 1
        if amount == x:
            break
    return count


def change():
    print('Cambio de Monedas: Solución con Algoritmo Ávido')
```

```
    coins = [1, 5, 10, 25]
    print('Sistema monetario: - coins =', coins)
    print('(Nota: el programa funcionará SÓLO si coins es canónico...)')
    print()

    print('Monto a cambiar...')
    x = validar_mayor(0)

    cant = greedy_change(x, coins)
    print()
    print('Cantidad mínima de monedas a devolver:', cant)


if __name__ == '__main__':
    change()
```

El programa anterior inicializa el arreglo *coins* con un conjunto que sabemos que es canónico. Luego carga por teclado el valor *x* a cambiar, validando que sea mayor a cero, e invoca a la función *greedy_change()* que es la que aplica el algoritmo ávido.

La función *greedy_change()* comienza chequeando con un *for iterador* [3] [4] el conjunto *coins*, para determinar si existe una moneda cuyo valor ya sea igual a *x*, en cuyo caso termina retornando un 1 (la solución óptima es devolver *x* con una sola moneda).

Pero si no existe una moneda con valor *x*, la función *greedy_change()* procede a ordenar el arreglo *coins* de mayor a menor usando el algoritmo de *ordenamiento por inserción simple*. Este ordenamiento es necesario para el *algoritmo ávido* si ***no*** se supone que el arreglo *coins* vendrá ya ordenado. Y en este caso puntual, hemos usado un ordenamiento simple debido a que suponemos que *coins* tendrá muy pocos elementos y un algoritmo simple será entonces aceptable en cuanto a tiempo de ejecución. Si se espera que *coins* tenga una *gran cantidad de elementos*, sabemos que entonces debería aplicarse un método compuesto (*quicksort*, *heapsort*, *shellsort*, etc.) [5]

Luego de ordenar el arreglo, se lanza otro ciclo *for iterador* que va tomando uno por uno los valores del arreglo *coins* (en la variable *value*). Como *coins* está ordenado de mayor a menos, los valores de *value* serán tomados uno a uno de forma que el primer valor será el mayor. Se usa un acumulador *amount* para ir sumando los valores seleccionados en *value*, y un contador *count* para llevar la cuenta de la cantidad de valores/monedas que se acumularon.

Dentro del *for iterador*, otro ciclo (un *while* en este caso) prueba a sumar el valor actual *value* de la moneda seleccionada tantas veces como sea posible, mientras no se supere el valor de *x*. Si se supera el valor de *x*, el ciclo interno *while* corta, y el *for iterador* toma el siguiente elemento de *coins*, asignándolo nuevamente en *value*, repitiendo este esquema hasta que finalmente *amount* llegue a valer *x*. En ese momento el *while* se detendrá, y también lo hará el *for iterador* con el *break* ubicado en la condición debajo del *while* [3] [4]. La función retorna el valor final de *count*, que la cantidad de monedas a usar para cambiar *x*.

Ahora bien: si el número y designación de las monedas disponibles en *coins* es *arbitrario,* entonces podría *no ser canónico* y la solución anterior basada en un *algoritmo ávido no funcionará.* Nos preguntamos cómo plantear entonces un algoritmo que *siempre* funcione.

Ya hemos indicado que lo primero será fijar alguna precondición que *garantice* que la solución existe y sabemos que eso depende de que siempre exista la moneda de 1 centavo: cualquiera sea la cantidad a expresar, siempre se podrá plantear como suma de monedas de

1 centavo. Pero si el conjunto *coins* no es canónico (incluso garantizando que contenga el valor 1) el algoritmo ávido no necesariamente dará la solución óptima.

Una solución mejor consiste en usar una *tabla* (un arreglo) en la cual se almacenen los resultados para valores menores a *x*, recalculados, y luego se use esa tabla para calcular nuevos casos. Al fin y al cabo, si queremos saber cuántas monedas hacen falta para cambiar *x* = 63 centavos, es posible que sirva saber cuántas monedas hacen falta para cambiar 30 o 60, y si esos cálculos ya fueron hechos con anterioridad sería útil poder reusar sus resultados sin tener que volver a calcular [2].

La idea: ir progresando en el cálculo del cambio óptimo para cada uno de los valores menores a *x*, (sin calcular dos o más veces para el mismo valor), y cada vez que se tenga un nuevo caso resuelto, se guarde su resultado en la tabla. Esta técnica o estrategia de solución basada en tablas de resultados previos, se conoce como *programación dinámica*.

En nuestro caso, dado el valor *x* a cambiar en monedas, queremos reutilizar los cálculos que hayamos realizado para cualquier valor menor que *x* durante el proceso, y para ello usaremos un arreglo *prev* que tendrá exactamente *x + 1* casilleros. De esta forma, el arreglo *prev* tendrá efectivamente su última casilla numerada con el valor *x*, y la idea es depositar en esa casilla el valor final: la *cantidad mínima de monedas* para cubrir el valor *x*.

Luego debemos ir llenando esa tabla *prev* con los resultados parciales de los subproblemas o valores *anteriores* al cálculo para *x*, para lo cual tendremos que plantear el mecanismo de llenado para cada casilla: este mecanismo se conoce como el planteo de la *recurrencia* a emplear para el llenado de la tabla. En todo proceso de *programación dinámica* se parte de uno o más *casos obvios*, llamados *casos base* de la recurrencia: en nuestro caso, el caso base es directo: si tenemos que cubrir un valor de *x* = 0, entonces necesitamos *0* monedas y sabemos entonces que la casilla *prev[0]* deberá valer 0.

A partir de allí, en cada casilla *prev[v]* (con *1 <= v <= x*) deberemos almacenar la *mínima cantidad de monedas posible para llegar al valor v*, usando los valores de las monedas disponibles. El cálculo podría parecer complicado, pero un ejemplo mostrará en forma clara la ecuación de recurrencia a emplear:

Suponga (igual que en la solución *ávida*) que los *n* valores nominales de las monedas disponibles están almacenados en el arreglo *coins*. Sea *coins* = [1, 5, 10, 25, 50] con *n = 5* y sea *v = 13*: esto es, queremos llenar la casilla *prev[13]* de la tabla, suponiendo que ya hemos llenado ***todas*** las anteriores. Está claro entonces que los posibles valores para *prev[13]* saldrán del ***mínimo*** entre todas las siguientes combinaciones:

- ✓ 13 monedas de 1 centavo.
- ✓ 1 moneda de 1 centavo + el mínimo para cubrir 12 centavos = 1 + prev[12]
- ✓ 1 moneda de 5 centavos + el mínimo para cubrir 8 centavos = 1 + prev[8]
- ✓ 1 moneda de 10 centavos + el mínimo para cubrir 3 centavos = 1 + prev[3]
- ✓ Y no podemos usar monedas de 25 o 50 centavos, ya que ambas son mayores a *v* = 13.

De allí surge con sencillez la expresión completa de la recurrencia a usar:

prev[v] = mínimo(v, mínimo(1 + prev[ui]))
(donde ui = v – coins[j] con j = 0, 1, 2, … n – 1)

Finalmente, el planteo del algoritmo completo sería entonces:

- Entradas:

- Un arreglo *coins*, con los *n* valores nominales de las monedas disponibles (de forma que el valor nominal 1 exista).
- Un valor entero *x* > 0 que será el valor a cubrir con las monedas disponibles.

- Salidas:
  - Un número entero que indique la mínima cantidad de monedas a devolver para cubrir el valor amount.

- Proceso (algoritmo básico – recurrencia a emplear):

```
dinamic_change(x, coins):
      1. Sea n = tamaño del arreglo coins (cantidad de valores de monedas)
      2. Sea la tabla prev con x + 1 casilleros, inicializados en 0
      3. Para  todo v en [1..x] (incluyendo 1 y x):
            3.1. prev[v] = mínimo(v,  mínimo(1 + prev[ui]))
                  (donde ui = v – prev[j] (con j = 0, 1, 2, …, n-1))
      4. Retornar prev[x]
```

Un replanteo del problema del cambio de monedas, ahora usando *programación dinámica*, se muestra en el mismo proyecto que acompaña a esta ficha, pero ahora en el siguiente modelo llamado *dinamico.py*:

```
__author__ = 'Cátedra de AED'


def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tSe pidio mayor a', lim, '... cargue de nuevo...')
    return n


def dinamic_change(x, coins):
    # cantidad de valores de monedas disponibles...
    n = len(coins)

    # tabla para los resultados previos del cambio de los valores menores a x...
    # ... incluyendo una casilla prev[x] para el cambio del propio valor x...
    # ... y prev[0] = 0 por el caso base: hacen falta 0 monedas para x = 0...
    prev = [0] * (x + 1)

    # calcular el mínimo para cada posible valor de v con 1 <= v <= x
    for v in range(1, x + 1):
        # empezamos suponiendo que el mínimo es el propio v...
        minv = v

        # ... y verificamos si hay alguna combinación menor...
        for j in range(n):
            # si coins[j] se pasa, seguir el ciclo...
            if coins[j] > v:
                continue

            ui = v - coins[j]
            if prev[ui] + 1 < minv:
                minv = prev[ui] + 1

        # ...almacenar el mínimo encontrado para v en prev[v]...
        prev[v] = minv

    # retornar el mínimo para x, que está en la última casilla...
    return prev[x]
```

```
def change():
    print('Cambio de Monedas: Solución con Programación Dinámica')

    coins = [1, 5, 10, 21, 25]
    print('Sistema monetario: - coins =', coins)
    print('(Nota: vale cualquiera que tenga al 1, aún no canónico o desordenado)')
    print()

    print('Monto a cambiar...')
    x = validar_mayor(0)

    cant = dinamic_change(x, coins)
    print()
    print('Cantidad mínima de monedas a devolver:', cant)


if __name__ == '__main__':
    change()
```

La función *dinamic_change()* aplica en forma directa el algoritmo basado en la recurrencia explicada más arriba. Se deja para el alumno el estudio de los detalles.

3.] Conceptos básicos sobre *Backtracking*.

Otra de las estrategias clásicas de resolución de problemas (o planteo de algoritmos) se conoce como *backtracking* (o *vuelta atrás*). Se trata de una técnica de base recursiva mediante la cual se exploran todos los posibles caminos que pudieran conducir a la solución de un problema, pero de forma tal que cuando se descubre que un camino no conduce a una solución válida, se aprovecha el retorno (o *vuelta atrás*, o *backtracking*) de las instancias recursivas que se hayan activado para "deshacer" los cambios que hayan llevado por ese camino incorrecto, e intentar la exploración de otro camino con nuevos valores [6] [2].

La técnica de *backtracking* es muy útil en problemas en los que las soluciones surgen de una o más combinaciones de los mismos datos de entrada, ya que el proceso de *prueba y error* que esencialmente permite realizar garantiza que el algoritmo no dejará afuera ninguna posible solución sin explorar. Sin embargo, el uso de la recursión para explorar todos los posibles caminos puede llevar a un número considerable o demasiado elevado de invocaciones o instancias recursivas, aumentando notablemente el tiempo de ejecución y/o la cantidad de memoria de stack requerida.

Para evitar estos inconvenientes, los programadores normalmente estudian cada problema en particular, tratando de identificar posibles formas de evitar trabajo extra y ahorrando así algunas tandas de invocaciones recursivas. En algunos casos, las técnicas empleadas para evitar esa sobrecarga de trabajo tienen nombres y características propias (como el proceso de *poda alfa-beta*), y en otros casos se trata directamente de variantes y trucos sutiles que dependen de cada problema en particular. Los detalles de estas técnicas escapan al alcance introductorio de estas notas, pero aún así el esquema general y básico de aplicación de la estrategia de *backtracking* puede analizarse sin mayores problemas.

Como hemos indicado en la introducción de esta misma Ficha de estudios, la técnica de *backtracking* suele usarse con frecuencia (y con éxito) en problemas de optimización combinatoria (encontrar la mejor forma de combinar ciertos elementos), y muchos de esos problemas tienen que ver con juegos de ingenio. Uno muy conocido, y muy usado para exponer los principios de aplicación del *backtracking*, es el *Problema de las Ocho Reinas*, que enunciamos a continuación [6]:

Problema 64.) *Problema de las Ocho Reinas: Es un problema de ingenio derivado del juego del ajedrez, que como se sabe, consta de un tablero de 8 * 8 = 64 casilleros en el cual se ubican piezas de diversas características (rey, reina, alfil, caballo, torre y peón) y con diversas capacidades de movimiento. Entre esas piezas las más poderosa del juego es la **reina**, ya que puede moverse (y atacar) a lo largo de toda la fila, toda la columna y las dos diagonales que pasen por el casillero de su posición en el tablero. El Problema de las Ocho Reinas[1] se enuncia entonces de forma simple: en un tablero de ajedrez normal, tratar de ubicar ocho reinas pero de tal forma que **ninguna de ellas ataque a ninguna de las otras**.*

Discusión y solución: Este problema de ingenio fue planteado por primera vez en 1848 por el ajedrecista *Max Bezzel*, y fue estudiado por diversos matemáticos (entre ellos: *Gauss*, *Cantor*, *Nauck* y *Glaisher*). El uso y aplicación de la computadora y las técnicas recursivas han aportado numerosos planteos algorítmicos que permiten arribar a una o a todas las soluciones posibles en forma relativamente sencilla[2].

Un rápido análisis inicial muestra con claridad que cualquier solución del problema necesariamente tendrá que ubicar a cada una de las ocho reinas en ocho columnas diferentes y también en ocho filas diferentes del tablero (ya que de otro modo, estarían en posiciones de ataque mutuo).

Así, entonces, cada una de las reinas puede ser representada por un número *col* entre 0 y 7 que indique en qué *columna* del tablero estará ubicada, y la posición *fil* (o número de fila) de esa reina en la columna *col* puede almacenarse en un arreglo unidimensional de ocho componentes de tipo *int* que llamaremos *rc* (por *row on column = fila en la columna*). Cada casillero *rc[col]* se usará para guardar el número de fila dentro de la columna *col* en que estará ubicada la reina cuyo número es justamente *col*. Por ejemplo, si *rc* fuese:

```
rc = [2, 4, 7, 0, 5, 1, 3, 6]
```

entonces se estaría indicando que queremos a la primera reina (la reina de la columna 0) en la fila 2 del tablero; luego queremos a la segunda reina (la reina de la columna 1) en la fila 4 del tablero, y así sucesivamente.

Esta forma de notación es compacta y permite controlar rápidamente que las reinas no se ataquen entre ellas en ninguna fila y ninguna columna: está claro que no debería haber dos o más casilleros del arreglo *rc* con el mismo número, ya que eso implicaría que dos o más reinas en dos o más columnas distintas estarían ubicadas en la misma fila (como en el ejemplo siguiente, en el cual la tercera reina (columna 2) y la sexta (columna 5) están ubicadas en la misma fila (la 7):

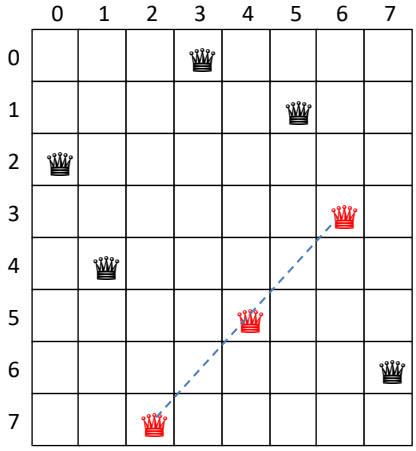
```
rc = [2, 4, 7, 0, 5, 7, 3, 6]
```

[1] El tema de las Ocho Reinas nos lleva inexorablemente a la excelente película argentina *Nueve Reinas* (del año 2000), dirigida por *Fabián Bielinsky* y protagonizada por *Ricardo Darín* y *Gastón Pauls*. En ella se muestra la historia de dos estafadores que en un momento dado se quedan con una plancha de nueve sellos postales, muy valiosa, conocida como las "Nueve Reinas". Claro… la plancha que ellos tenían era una falsificación y los dos bandidos intentan a toda costa venderla por un alto precio a un millonario español. Uno de los estafadores busca por todos los medios sacar al otro del negocio, para quedarse con toda la ganancia… pero sobre el final se produce un giro tan inesperado como deseado por todos los espectadores...

[2] De todos los posibles enfoques algorítmicos que pueden sugerirse, nos hemos basado especialmente en el esquema indicado por *Niklaus Wirth* en su libro *"Algoritmos + Estructuras de Datos = Programas"*, capítulo 3, sección 3.5.

Queda todavía el tema no menor del control de las diagonales. La notación *row on column* que expresamos en el arreglo *rc* permite con sencillez controlar las filas y las columnas, pero nada controla respecto de posibles ataques en diagonal. Si hacemos un gráfico más detallado de las posiciones indicadas por el primer arreglo *rc* = [2, 4, 7, 0, 5, 1, 3, 6], vemos que el tablero representado para las ocho reinas sería el siguiente:

Figura 1: Una ubicación incorrecta para las Ocho Reinas.



Podemos ver de inmediato que la disposición prevista en el arreglo *rc* no es correcta, ya que las reinas (dibujadas en *rojo*) en las columnas 2, 4 y 6 comparten una de las diagonales del tablero, y por lo tanto se están atacando. El punto es que esperaríamos poder hacer el control en las diagonales en forma simple, emulando lo que se hizo con el control de las filas y las columnas. Si el tablero se representa como una matriz cuadrada ese control no será sencillo, ya que tendremos que realizar una gran cantidad de cálculos y validaciones para saber si una reina comparte o no una diagonal con otra.

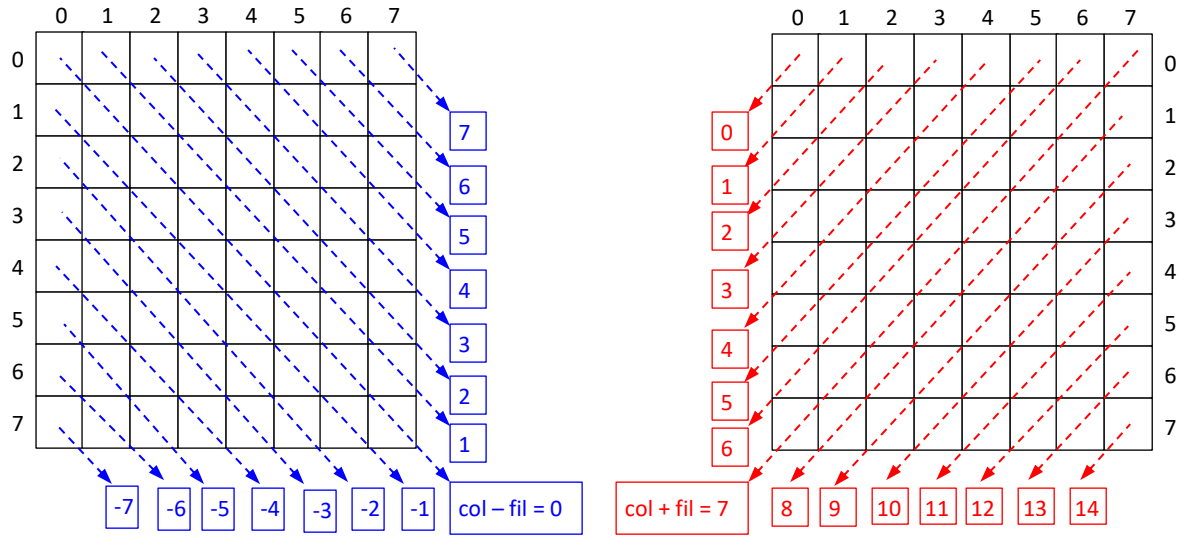
Para intentar simplificar, notemos que en general hay dos tipos de diagonales en una matriz (y esto es especialmente visible si la matriz es cuadrada): Por un lado, la diagonales que se orientan en la misma forma que la *diagonal principal* (y las llamaremos *diagonales normales*) y por el otro, las que se orientan en sentido inverso al de la diagonal principal (las designaremos como *diagonales inversas*). Y una propiedad interesante es que en cada una de las *diagonales normales*, la *diferencia* entre el número de columna y el número de fila de sus elementos es constante, mientras que en las *diagonales inversas* es constante la *suma* entre el número de fila y el de columna de cada elemento. La *Figura 2* (*página 629*) ilustra lo dicho (los números en azul o rojo al final de cada diagonal normal o inversa indican el valor de la resta o la suma de los índices de columna y fila de cada casillero en esas diagonales).

Como se puede ver en esa gráfica, una matriz cuadrada de 8 * 8 casilleros tiene 15 diagonales de cada tipo, y cada una de ellas puede asociarse a un *número único*: la resta de los índices de los casilleros en las diagonales normales (15 números diferentes entre -7 y 7) y la suma de esos índices en las diagonales inversas (15 números diferentes entre 0 y 14).

Con estos elementos a la vista, en el planteo de la solución para el *Problema de las Ocho Reinas* incorporaremos otros tres arreglos unidimensionales (además del arreglo *rc*) para controlar lo que ocurre en el tablero. Estos nuevos arreglos almacenarán valores booleanos, y serán los siguientes [6]:

- ***qr*** (de *queens on row*) con 8 casilleros, de forma que *qr[fil] = True* indicará que la fila *fil* está libre de reinas (o sea, no hay reinas en la fila indicada por *fil*).
- ***qid*** (de *queens on inverse diagonal*) con 15 casilleros, de forma que *qid[di] = True* indicará que la diagonal inversa *di* (con 0 <= *di* <= 14) está libre de reinas (o sea, no hay reinas en la diagonal inversa indicada por *di*).
- ***qnd*** (de *queens on normal diagonal*) con 15 casilleros, de forma que *qnd[dn] = True* indicará que la diagonal normal *dn* (con 0 <= *dn* <= 14) está libre de reinas (o sea, no hay reinas en la diagonal normal indicada por *dn*).

Figura 2: Diagonales normales e inversas en una matriz de 8 * 8.



Es posible que el estudiante haya notado un pequeño detalle en las descripciones anteriores: según la gráfica de la *Figura 2*, las *diagonales normales* están asociadas a números entre -7 y 7, pero ahora estamos indicando que los 15 componentes del arreglo *qnd* serán usados para marcar si cada diagonal normal está libre o no de reinas. Pero en Python un número negativo usado como índice de un arreglo estaría accediendo a un casillero tomando como referencia que el índice -1 identifica al último componente, el -2 al anteúltimo y así en forma consecutiva. Por lo tanto, si se quiere marcar un casillero del vector *qnd* como libre de reinas, pero ese casillero está asociado a una diagonal con identificador negativo, entonces el programa *podría* estar accediendo a un casillero incorrecto si el programador no hizo sus cálculos en forma detallada.

Para evitar el problema (y evitar también posibles confusiones al mezclar índices positivos con índices negativos), hemos procedido de la siguiente forma *en cuanto a las diagonales normales*: si se quiere saber en qué diagonal normal está el casillero del tablero con índices [*fil*][*col*], se calcula la resta *col – fil*, pero a ese resultado se le suma 7. Con esto, como las restas están en el rango [-7, 7], entonces los valores obtenidos al sumar 7 estarán en el rango [0, 14]. Así, en el arreglo *qnd* el casillero 0 se usará para representar a la diagonal normal con resta igual a -7, el casillero 1 para la diagonal con resta igual a -6, y así sucesivamente. Formalmente, el casillero *qnd[dn]* (con 0 <= *dn* <= 14) representará a la diagonal normal *dn = (col – fil) + 7*.

Finalmente, podemos ahora analizar el programa para calcular una posición válida para cada reina en el tablero (ver el modelo *backtracking01.py* en el proyecto que acompaña a esta ficha). La función de entrada o de arranque del programa es la siguiente:

```
def main():
    global rc, qr, qid, qnd

    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ...inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ...15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15

    # qnd[k] = true si la diagonal normal k está libre de reinas...
    # ...15 diagonales normales... inicialmente todas libres de reinas...
    qnd = [True] * 15

    print('Problema de las Ocho Reinas: Una solución con Backtraking')
    print()

    success = intend(0)
    if success:
        print('Solución posible encontrada:', rc)
    else:
        print('No hay solución posible para esa posición de partida...')
```

La función *main()* comienza definiendo los cuatro arreglos *rc*, *qr*, *qid* y *qnd* como variables globales, para hacer que sea más simple compartirlas con la función *intend()* que es la luego aplicará el proceso de *backtracking*. Los cuatro arreglos son creados de forma de contener tantos casilleros como serán requeridos (8 casilleros con valor inicial -1 para *rc*, otros 8 con valor inicial *True* para *qr*, y 15 con valor inicial *True* para *qid* y *qnd*).

El valor -1 en cada casillero de *rc* indica que por el momento no se sabe en qué fila del tablero será ubicada cada una de las reinas representadas por las columnas de ese arreglo. El valor *True* en cada casillero de los arreglos *qr*, *qid* y *qnd* indica que por ahora todas las filas y todas las diagonales están libres de reinas (ya que de hecho, ninguna columna está ocupada…)

Luego, la función *main()* simplemente invoca a la función *intend(col)* enviando como parámetro el valor 0. El parámetro enviado cuando se invoca a intend() la primera vez (o sea, *col = 0*) es el índice de la primera columna del tablero (de hecho, el número de la primera reina). La función *intend()* comienza sus cálculos justamente tratando de determinar en qué fila de esa columna *col = 0* deberá ubicar la primera reina, y luego aplicará *backtracking* para calcular el resto de las ubicaciones.

Cuando la función *intend(col)* finalice su ejecución, el arreglo *rc* estará completo y cada casillero contendrá el número de la fila del tablero en que debe colocarse cada reina, representada por cada índice del vector *rc*, por lo cual *main()* sólo debe mostrar ese arreglo. La función *intend(col)* en rigor, retorna un valor booleano que indica si se ha encontrado o no una solución, y *main()* formalmente chequea ese valor retornado antes de mostrar el arreglo. Sin embargo, si la función es invocada enviando como parámetro el valor 0, el

resultado será efectivamente *True* (simplemente, sabemos que al menos una solución existe si se comienza trabajando desde la columna 0…) por lo que el control realizado por *main()* es redundante en este caso.

La función *intend(col)* es la siguiente:

```
def intend(col):
    global rc, qr, qid, qnd

    fil, res = -1, False
    while not res and fil != 7:
        res = False
        fil += 1
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
            if col < 7:
                res = intend(col + 1)
                if not res:
                    qr[fil] = qid[di] = qnd[dn] = True
            else:
                res = True
    return res
```

La variable *fil* se usa para contener el índice de cada una de las filas del tablero, que serán exploradas una por una hasta dar con una solución. La variable *res* se usa como bandera para controlar si en algún momento se ha dado con una solución. Antes de iniciar el proceso, la variable *res* se pone en *False*, y luego se lanza un ciclo *while* para controlar cada posible fila. El ciclo continúa mientras queden filas sin explorar y la variable *res* se mantenga en *False*.

En cada vuelta del ciclo, se trabaja con la columna *col* (inicialmente, *col = 0*) y se incrementa *fil* en 1 para intentar ubicar la primera reina justamente en esa posición. En la primera vuelta del ciclo, la *posición propuesta* para la *reina 0* es entonces *col = 0* y *fil = 0*, que en nuestra representación sería *rc[col] = fil*.

El paso siguiente es comprobar si esa posición propuesta es *válida* (o *segura*). Está claro que la posición *rc[col] = fil* será *válida* si la fila *fil* está libre de reinas (o sea, si *qr[fil] == True*), y además si las dos diagonales *di* y *dn* que pasan por esa casilla están a su vez libres de reinas. Los índices de control de las dos diagonales son, respectivamente *di = col + fil = 0 + 0 = 0*, y *dn = col – fil + 7 = 0 + 0 + 7 = 7*. Por lo tanto, sólo debemos mirar los arreglos qid y qnd en esas posiciones para saber si esas diagonales están libres. La condición de control completa sería entonces:

```
if qr[fil] and qid[di] and qnd[dn]:
```

Si esa condición es *True*, entonces la posición *rc[col] = fil* es segura, y se puede intentar dejar la reina allí. Por lo tanto, si se entra en la rama verdadera, simplemente *se registra que ahora esa posición está ocupada (bloqueando la fila y las dos diagonales)*:

```
if qr[fil] and qid[di] and qnd[dn]:
    rc[col] = fil
    qr[fil] = qid[di] = qnd[dn] = False
```

Lo que sigue es el *paso recursivo*: habiendo registrado a la reina de la columna *col = 0* en la posición *rc[col] = fil*, debemos ahora *intentar* ubicar a la reina siguiente (la reina de la

columna *col + 1*) en el tablero. Esto sólo será posible (obviamente) si la reina que se acaba de ubicar en el tablero *no es todavía la de la columna 7* (si fuese la 7, sería ya la última y *col + 1 = 8* no existiría). Por ese motivo, se pregunta si *col < 7*, y en ese caso se activa recursivamente la propia función *intend()*, tomando como parámetro ahora el valor *col + 1*.

En la nueva instancia recursiva el proceso comienza nuevamente, pero ahora centrado en la columna col = 1 (reina de la columna 1). Los arreglos qr, qid y qnd han modificado sus valores, por lo cual alguna fila, diagonal normal y/o diagonal inversa podrían estar bloqueadas. Esto hace que cada nueva reina que se va incorporando (con cada nueva instancia recursiva) tenga cada vez menos libertad de acción para su posición propuesta, y podría ocurrir que para la reina actual la posición *rc[col] = fil* no sea válida. Si tampoco fuese válida ninguna otra fila en esa columna, la instancia actual de la función *intend()* terminaría retornando *False*. Pero ese retorno de *False* sería tomado por la instancia recursiva *anterior* de *intend()*. La condición inmediatamente debajo de la llamada recursiva a *intend()*

```
res = intend(col + 1)
if not res:
    qr[fil] = qid[di] = qnd[dn] = True
```

controla justamente si la invocación recursiva para ubicar a la *siguiente* reina tuvo éxito o no. Y ***si no lo tuvo***, entra en acción el mecanismo de corrección: la posición de la reina actual era segura, pero la reina siguiente no tuvo lugar para ella, por lo cual el camino seguido no conduce a una solución: se debe retroceder, eliminar el registro seguro de la reina actual (es decir, informar que la fila y las dos diagonales no están bloqueadas por esa reina), y probar una nueva ubicación en la siguiente vuelta del ciclo.

Este *mecanismo de corrección* es justamente el proceso de *backtracking* que hemos venido a buscar. La recursión permite ir hacia adelante explorando posibles posiciones de cada reina, y mientras todo vaya bien, no hay motivo para cambiar las posiciones ya registradas. Pero si en algún momento se llega a un callejón sin salida (la reina actual no puede ubicarse en ningún sitio), la instancia recursiva que no logró ubicar a esa reina termina, y vuelve atrás a la instancia anterior retornando *False*, lo cual es tomado como un aviso de *solución incorrecta*. La única forma de cambiar eso, consiste en quitar de su posición a la reina anterior y probar todo de nuevo. Y eso es justamente lo que el *backtracking* permite hacer: ensayar un camino, dejarlo registrado, y si algo falla, retroceder, borrar el registro anterior, y ensayar un nuevo camino.

Los detalles finos del mecanismo de trabajo de esta función se dejan para ser estudiados por el alumno. El programa completo se muestra a continuación:

```
__author__ = 'Cátedra de AED'


def intend(col):
    global rc, qr, qid, qnd

    fil, res = -1, False
    while not res and fil != 7:
        res = False
        fil += 1
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
```

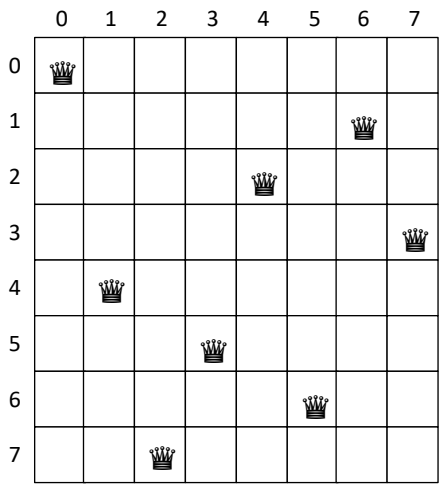
```
            if col < 7:
                res = intend(col + 1)
                if not res:
                    qr[fil] = qid[di] = qnd[dn] = True
            else:
                res = True
    return res


def main():
    global rc, qr, qid, qnd

    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ...inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ...15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15

    # qnd[k] = true si la diagonal normal k está libre de reinas...
    # ...15 diagonales normales... inicialmente todas libres de reinas...
    qnd = [True] * 15

    print('Problema de las Ocho Reinas: Una solución con Backtraking')
    print()

    success = intend(0)
    if success:
        print('Solución posible encontrada:', rc)
    else:
        print('No hay solución posible para esa posición de partida...')

if __name__ == '__main__':
    main()
```

Específicamente, la función *intend(0)* dejará el arreglo *rc* con estos valores:

```
Solución posible encontrada: [0, 4, 7, 5, 2, 6, 1, 3]
```

que corresponden a ***una*** solución (de muchas posibles), que gráficamente sería equivalente al tablero que sigue:

Figura 3: El tablero equivalente a la solución calculada por *intend(0)*.



4.] Cálculo de *todas* las soluciones al Problema de las Ocho Reinas.

En el problema 58 hemos estudiado un algoritmo basado en *backtracking* para calcular ***una*** solución al *Problema de las Ocho Reinas*. Sin embargo, es intuitivamente simple concluir que existen muchas otras soluciones posibles. Un análisis sistemático permite ver que el total de soluciones posibles (para un tablero de 8 * 8 y exactamente 8 reinas) es 92, aunque un estudio aún más detallado muestra que las soluciones realmente distintas son en realidad 12, y todas las otras son variaciones simétricas (rotadas de alguna forma) con relación a alguna de las 12 básicas.

Si se quiere obtener un listado de las 92 soluciones posibles (básicas o simétricas…), se puede hacer una modificación simple del programa que ya hemos mostrado para el problema 54: en lugar de un ciclo while que explore las filas una por una y se detenga con una bandera de corte cuando encuentre una solución, se puede usar un ciclo *for* que obligatoriamente recorra las 8 filas en cada una de las 8 columnas (comenzando igual que antes desde col = 0). Cada vez que se encuentre una solución, contarla y mostrarla en ese mismo momento (mostrar en ese momento el contador y el contenido actual del vector *rc*). Y como esa solución ya fue encontrada y mostrada, inmediatamente luego de mostrarla borrar su registro y proceder con otra vuelta de ciclo para buscar la siguiente.

El modelo *backtracking02.py* del proyecto que acompaña a esta ficha, hace justamente eso. Al ejecutarlo, se mostrarán 92 líneas con otras tantas variantes del arreglo *rc*, cada una representando un tablero diferente para las *Ocho Reinas*. El programa es el siguiente (y de nuevo, se dejan los detalles para ser analizados por el estudiante):

```
__author__ = 'Cátedra de AED'


def intend(col):
    global rc, qr, qid, qnd, cs
    for fil in range(8):
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
            if col < 7:
                intend(col + 1)
            else:
                cs += 1
                print(rc, ' (', cs, ')', sep='')
            qr[fil] = qid[di] = qnd[dn] = True


def main():
    global rc, qr, qid, qnd, cs
    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ...inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ...15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15
```

```
    # qnd[k] = true si la diagonal normal k está libre de reinas...
    # ...15 diagonales normales... inicialmente todas libres de reinas...
    qnd = [True] * 15

    # contador de soluciones...
    cs = 0
    print('Problema de las Ocho Reinas: Todas las soluciones...')
    print()
    print('Distintas soluciones encontradas...')
    intend(0)

if __name__ == '__main__':
    main()
```

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

[3] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: https://docs.python.org/3/.

[4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

[5] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[6] N. Wirth, Algoritmos + Estructuras de Datos = Programas, Madrid: Ediciones del Castillo, 1989.