TÉCNICAS DE DISEÑO DE ALGORTIMOS

1) DIVIDE Y VENCERÁS

El término **Divide y Vencerás**, en su acepción más amplia, es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos.
En nuestro contexto, Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Esto naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en *k* subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es *n*, hemos de conseguir dividir el problema en *k* subproblemas (donde $1 <= k <= n$), cada uno con una entrada de tamaño n_k y donde $0 <= n_k < n$. A esta tarea se le conoce como *división*.

2. En segundo lugar han de resolverse independientemente todos los subproblemas, ya sea directamente si son elementales, o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos *base*.

3. Por último, *combinar* las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión, esto es:
a) Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
b) Sin embargo, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Desde el punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Como ejemplo pensemos en el cálculo de la sucesión de Fibonacci, el cual, a pesar de ajustarse al esquema general y de tener sólo dos llamadas recursivas, tan sólo se puede considerar un algoritmo recursivo pero no clasificarlo como diseño Divide y Vencerás. Esta técnica está concebida para resolver problemas de manera eficiente y evidentemente este algoritmo, con tiempo de ejecución exponencial, no lo es.

Para aquellos problemas en los que la solución haya de construirse a partir de las soluciones de subproblemas entre los que se produzca necesariamente solapamiento existe otra técnica de diseño más

apropiada, y que permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos. Estamos hablando de la Programación Dinámica.

Otra consideración importante a la hora de diseñar algoritmos Divide y Vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible. En caso contrario nos podemos encontrar con "anomalías de funcionamiento" como le ocurre al algoritmo de ordenación Quicksort. Éste es un representante claro de los algoritmos Divide y Vencerás, y su caso peor aparece cuando existe un desequilibrio total en los subproblemas al descomponer el vector original en dos subvectores de tamaño 0 y n–1.
Como vimos en el capítulo anterior, en este caso su orden es $O(n^2)$, frente a la buena complejidad, $O(n\log n)$, que consigue cuando descompone el vector en dos subvectores de igual tamaño.

También es interesante tener presente la dificultad y es esfuerzo requerido en cada una de estas fases va a depender del planteamiento del algoritmo concreto.
Por ejemplo, los métodos de ordenación por Mezcla y Quicksort son dos representantes claros de esta técnica pues ambos están diseñados siguiendo el esquema presentado: dividir y combinar.

La división de Quicksort es costosa, pero una vez ordenados los dos subvectores la combinación es inmediata. Sin embargo, la división que realiza el método de ordenación por Mezcla consiste simplemente en considerar la mitad de los elementos, mientras que su proceso de combinación es el que lleva asociado todo el esfuerzo.

El algoritmo de búsqueda binaria es un ejemplo claro de la técnica Divide y Vencerás.
En este ejemplo la división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de "simplificación". Por supuesto, aquí no es necesario un proceso de combinación de resultados.
Su caso base se produce cuando el vector tiene sólo un elemento. En esta situación la solución del problema se basa en comparar dicho elemento con x.
Como el tamaño de la entrada (en este caso el número de elementos del vector a tratar) se va dividiendo en cada paso por dos, tenemos asegurada la convergencia al caso base.

2) ALGORITMOS ÁVIDOS

El método que produce algoritmos ávidos es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización.
Dado un problema con n entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema. Cada uno de los subconjuntos que cumplan las restricciones diremos que son soluciones *prometedoras*. Una solución prometedora que maximice o minimice una función objetivo la denominaremos solución óptima.

Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo ávido vamos a definir una serie de elementos que han de estar presentes en el problema:

- Un conjunto de *candidatos*, que corresponden a las n entradas del problema.
- Una *función de selección* que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.

- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor*. Entendemos por prometedor que sea posible seguir añadiendo candidatos y encontrar una solución.
- Una *función objetivo* que determine el valor de la solución hallada. Es la función que queremos maximizar o minimizar.
- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

  Con estos elementos, podemos resumir el funcionamiento de los algoritmos ávidos en los siguientes puntos:

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.
2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.
3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadilo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa.

En resumen, se desprende que los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre? En primer lugar, porque no todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global, como mostraremos en varios ejemplos de este capítulo. La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.
Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno para hoy siempre es bueno para el futuro. Y aquí radica la dificultad de estos algoritmos.
Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo.

No basta con diseñar un procedimiento ávido, que seguro que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema. Debido a su eficiencia, este tipo de algoritmos es muchas veces utilizado aún en los casos donde se sabe que no necesariamente encuentran la solución óptima. En algunas ocasiones la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada (piénsese en el localizador de un avión de combate, o en los procesos de toma de decisiones de una central nuclear).

También hay otras circunstancias, como veremos al analizar los algoritmos que siguen la técnica de Ramificación y Poda, en donde lo que interesa es conseguir cuanto antes una solución del problema y, a partir de la información suministrada por ella, conseguir la óptima más rápidamente. Es decir, la eficiencia

de este tipo de algoritmos hace que se utilicen aunque no consigan resolver el problema de optimización planteado, sino que sólo den una solución "aproximada".

El nombre de algoritmos ávidos, también conocidos como voraces (su nombre original proviene del término inglés *greedy*) se debe a su comportamiento: en cada etapa "toman lo que pueden" sin analizar consecuencias, es decir, son glotones por naturaleza. En este tipo de algoritmos el proceso no acaba cuando disponemos de la implementación del procedimiento que lo lleva a cabo. Lo importante es la demostración de que el algoritmo encuentra la solución óptima en todos los casos, o bien la presentación de un contraejemplo que muestra los casos en donde falla.

Ejemplos de aplicación de la Técnica Algoritmos Ávidos:
LOS ALGORITMOS DE PRIM Y KRUSKAL

Partimos de un grafo conexo, ponderado y no dirigido $g = (V,A)$ de arcos no negativos, y deseamos encontrar el árbol de recubrimiento de g de coste mínimo.
Por árbol de recubrimiento de un grafo g entendemos un subgrafo sin ciclos que contenga a todos sus vértices. En caso de haber varios árboles de coste mínimo, nos quedaremos de entre ellos con el que posea menos arcos.

Existen al menos dos algoritmos muy conocidos que resuelven este problema, como son el de Prim y el de Kruskal. En ambos se va construyendo el árbol por etapas, y en cada una se añade un arco. La forma en la que se realiza esa elección es la que distingue a ambos algoritmos.

El algoritmo de Prim comienza por un vértice y escoge en cada etapa el arco de menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya seleccionados y el otro no. Al incluir un nuevo arco a la solución, se añaden sus dos vértices al conjunto de vértices seleccionados.

En el de Kruskal se ordenan primero los arcos por orden creciente de peso, y en cada etapa se decide qué hacer con cada uno de ellos. Si el arco no forma un ciclo con los ya seleccionados (para poder formar parte de la solución), se incluye en ella; si no, se descarta.

Nuestro objetivo en esta sección no es la de describir en detalle estos algoritmos desde el punto de vista de matemática discreta o la teoría de grafos, sino la de considerarlos desde la perspectiva de los algoritmos ávidos.
a) En primer lugar, nos planteamos la implementación de ambos algoritmos siguiendo el esquema descrito y el análisis de su complejidad (espacio y tiempo).
b) Estos algoritmos trabajan sobre grafos conexos. Nos preguntamos lo que ocurriría si por error se les suministrara un grafo no conexo.

Supongamos que suministramos un grafo no conexo como entrada al algoritmo de Kruskal. En primer lugar el algoritmo terminaría puesto que el bucle va recorriendo todos los arcos de tal grafo. Y en segundo lugar su salida sería un árbol de expansión no conexo, pero que si observamos con detenimiento descubriremos que corresponde a la unión de los árboles de expansión mínimos de cada una de las componentes conexas del grafo. En este sentido, el algoritmo es bastante robusto.

No ocurre así con el de Prim, que no funciona en este caso puesto que hace uso de que sea conexo para buscar en cada paso el vértice k sobre el cual construir la solución. El que no sea conexo hace que, o bien k valga cero y por tanto se indexe erróneamente la matriz solución, o bien no se modifique su valor en cada

paso, lo que hace que el algoritmo no termine nunca. Podemos concluir por tanto que el suministrar un grafo conexo como entrada es una precondición fuerte del algoritmo de Prim implementado.

Una vez analizados ambos algoritmos, el uso de uno u otro va a estar condicionado por el tipo de grafo que tratemos. La complejidad del algoritmo de Prim es siempre de orden $O(n^2)$ mientras que el orden de complejidad del algoritmo de Kruskal $O(a \log n)$ no sólo depende del número de vértices, sino también del número de arcos.
Así, para grafos densos el número de arcos a es cercano a $n(n-1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2 \log n)$, peor que la complejidad $O(n^2)$ de Prim.
Sin embargo, para grafos dispersos en los que a es próximo a n, el algoritmo de Kruskal es de orden $O(n \log n)$, comportándose probablemente de forma más eficiente que el de Prim.

3) PROGRAMACIÓN DINÁMICA

En el diseño de la técnica Divide y Vencerás veíamos cómo para resolver un problema lo dividíamos en subproblemas independientes, los cuales se resolvían de manera recursiva para combinar finalmente las soluciones y así resolver el problema original. El inconveniente se presenta cuando los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos; entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que conlleva. En estos casos es cuando la Programación Dinámica nos puede ofrecer una solución aceptable.
La eficiencia de esta técnica consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para su futura utilización.
La Programación Dinámica no sólo tiene sentido aplicarla por razones de eficiencia, sino porque además presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado.

Donde tiene mayor aplicación la Programación Dinámica es en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).
La solución de problemas mediante esta técnica se basa en el llamado **principio de óptimo** enunciado por Bellman en 1957 y que dice:

"*En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima*".

Hemos de observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión. Un ejemplo claro para el que no se verifica este principio aparece al tratar de encontrar el camino de coste máximo entre dos vértices de un grafo ponderado.

Para que un problema pueda ser abordado por esta técnica ha de cumplir dos condiciones:
- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de óptimo.

En grandes líneas, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:
1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.
2. Definición recursiva de la solución.

3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

Ejemplos de la Técnica Programación Dinámica:

a) **CÁLCULO DE LOS NÚMEROS DE FIBONACCI**

Dicha sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:
$Fib(n) = 1 \text{ si } n = 0,1$
$Fib(n) = Fib(n-1) + Fib(n-2) \text{ si } n > 1$

El inconveniente es que el algoritmo resultante es poco eficiente ya que su tiempo de ejecución es de orden exponencial, $O(2^n)$.
Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y por tanto es necesario volver a calcular cada vez .
Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para poder reutilizarlos:

Fib(0)	*Fib*(1)	*Fib*(2)	... . ..	*Fib*(*n*)

El algoritmo iterativo que calcula la sucesión de Fibonacci utilizando tal tabla es:

```
Si  n<=1 Entonces Retorna 1
          Sino
            T[0]:=1;
            T[1]:=1:
            Para  i:=2 hasta n
                  T[i]:=T[i-1]+T[i-2]
            Retorna T[n]
```

Existe aún otra mejora a este algoritmo, que aparece al fijarnos que únicamente son necesarios los dos últimos valores calculados para determinar cada término, lo que permite eliminar la tabla entera y quedarnos solamente con dos variables para almacenar los dos últimos términos:

```
Si  n<=1  Entonces Retorna 1
           Sino
             x:=1; y:=1;
             Para i:=2 Hasta n
                suma:=x+y;
                y:=x;
                x:=suma;
             Retorna suma
```

Aunque esta función sea de la misma complejidad temporal que la anterior (lineal), consigue una complejidad espacial menor, pues de ser de orden O(*n*) pasa a ser O(1) ya que hemos eliminado la tabla.

El uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos, pieza clave de los algoritmos de Programación Dinámica, hace que nos fijemos no sólo en la complejidad temporal de los algoritmos estudiados, sino también en su complejidad espacial.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada.

b) **CÁLCULO DE LOS COEFICIENTES BINOMIALES**

En la resolución de un problema, una vez encontrada la expresión recursiva que define su solución, muchas veces la dificultad estriba en la creación del vector o la tabla que ha de conservar los resultados parciales. Así en este segundo ejemplo, aunque también sencillo, observamos que vamos a necesitar una tabla bidimensional algo más compleja. Se trata del cálculo de los coeficientes binomiales, definidos como:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n$$

$$\binom{n}{k} = 1 \quad \text{si } k=0 \text{ ó } k=n$$

El algoritmo recursivo que los calcula resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden O(*nk*) basado en la idea del Triángulo de Pascal.
Para ello es necesario la creación de una tabla bidimensional en la que iremos almacenando los valores intermedios que se utilizan posteriormente:

	0	1	2	3	……	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
….							
….							
n-1						C(*n*-1,*k*-1)	C(*n*-1,*k*)
n							C(*n*,*k*)

Se generará esta tabla por filas de arriba hacia abajo y de izquierda a derecha, mediante un algoritmo de complejidad polinómica.

c) EL PROBLEMA DEL CAMBIO DE MONEDA

Usando la técnica de algoritmos ávidos se puede obtener un algoritmo para minimizar, dado un sistema monetario, el número de monedas necesarias para reunir una cantidad. Dicho algoritmo funcionará cuando los tipos de monedas sean, por ejemplo, de 1, 5, 10 y 25 unidades; pero no obtiene necesariamente la descomposición óptima si añadiésemos una moneda de 12 unidades al sistema.
Dado que el algoritmo ávido para este problema falla en algunas ocasiones, debemos resolver dicho problema utilizando Programación Dinámica de forma que la solución sea satisfactoria en todos los casos.

d) EL ALGORITMO DE DIJKSTRA

Sea un grafo ponderado $g = (V,A)$, donde V es su conjunto de vértices, A el conjunto de arcos y sea $L[i,j]$ su matriz de adyacencia. Queremos calcular el camino más corto entre un vértice v_i tomado como origen y cada vértice restante v_j del grafo.
El clásico algoritmo de Dijkstra trabaja en etapas, en donde en cada una de ellas va añadiendo un vértice al conjunto D que representa aquellos vértices para los que se conoce su distancia al vértice origen.
Inicialmente el conjunto D contiene sólo al vértice origen.
Aún siendo el algoritmo de Dijkstra un claro ejemplo de algoritmo ávido, nos preguntamos si puede ser planteado como un algoritmo de Programación Dinámica, y si de ello se deriva alguna ventaja.
La técnica de la Programación Dinámica tiene grandes ventajas, y una de ellas es la de ofrecer un diseño adecuado y eficiente a todos los problemas que puedan plantearse de forma recursiva y cumplan el principio del óptimo.
Así, es posible plantear el algoritmo de Dijkstra en términos de la Programación Dinámica, y de esta forma aprovechar el método de diseño y las ventajas que esta técnica ofrece.
En primer lugar, observemos que es posible aplicar el principio de óptimo en este caso:

si en el camino mínimo de v_i a v_j está un vértice v_k como intermedio, los caminos parciales de v_i a v_k y de v_k a v_j han de ser a su vez mínimos.

La complejidad temporal del algoritmo es de orden $O(n^2)$, siendo de orden $O(n)$ su complejidad espacial.
No ganamos sustancialmente en eficiencia mediante el uso de esta técnica frente al planteamiento ávido del algoritmo, pero sin embargo sí ganamos en sencillez del diseño e implementación de la solución a partir del planteamiento del problema.

e) EL ALGORITMO DE FLOYD

Sea g un grafo dirigido y ponderado. Para calcular el menor de los caminos mínimos entre dos vértices cualesquiera del grafo, podemos aplicar el algoritmo de Dijkstra a todos los pares posibles y calcular su mínimo, o bien aplicamos el algoritmo de Floyd que, dada la matriz L de adyacencia del grafo g, calcula una matriz D con la longitud del camino mínimo que une cada par de vértices.
Nos planteamos si tal algoritmo puede ser considerado o no de Programación Dinámica, es decir, si reúne las características esenciales de ese tipo de algoritmos.
Este algoritmo puede ser considerado de Programación Dinámica ya que es aplicable el principio de óptimo, que puede enunciarse para este problema de la siguiente forma:

si en el camino mínimo de v_i a v_j, v_k es un vértice intermedio, los caminos de v_i a v_k y de v_k a v_j han de ser a su vez caminos mínimos.

f) EL ALGORITMO DE WARSHALL

Al igual que ocurre con el algoritmo de Floyd citado con anterioridad, estamos interesados en encontrar caminos entre cada dos vértices de un grafo. Sin embargo, aquí no nos importa su longitud, sino sólo su existencia. Por tanto, lo que deseamos es diseñar un algoritmo que permita conocer si dos vértices de un grafo están conectados o no, lo que nos llevaría al cierre transitivo del grafo.

4) VUELTA ATRÁS

Dentro de las técnicas de diseño de algoritmos, el método de Vuelta Atrás (del inglés *Backtracking*) es uno de los de más amplia utilización, en el sentido de que puede aplicarse en la resolución de un gran número de problemas, muy especialmente en aquellos de optimización.
Los métodos estudiados en los capítulos anteriores construyen la solución basándose en ciertas propiedades de la misma; así en los algoritmos Ávidos se va construyendo la solución por etapas, siempre avanzando sobre la solución parcial previamente calculada; o bien podremos utilizar la Programación Dinámica para dar una expresión recursiva de la solución si se verifica el principio de óptimo, y luego calcularla eficientemente.
Sin embargo ciertos problemas no son susceptibles de solucionarse con ninguna de estas técnicas, de manera que la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posibles soluciones, en las que tratamos de encontrar una o todas las soluciones y por lo tanto también la óptima.
Para llevar a cabo este estudio exhaustivo, el diseño Vuelta Atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.
En su forma básica, esta técnica de Vuelta Atrás se asemeja a un recorrido en profundidad dentro de un árbol cuya existencia sólo es implícita, y que denominaremos *árbol de expansión*. Este árbol es conceptual y sólo haremos uso de su organización como tal, en donde cada nodo de nivel *k* representa una parte de la solución y está formado por *k* etapas que se suponen ya realizadas. Sus hijos son las prolongaciones posibles al añadir una nueva etapa. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.
En este recorrido pueden suceder dos cosas:

La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución (una hoja del árbol). Si lo único que buscábamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscábamos eran todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que llamamos *nodos fracaso*. En tal caso, el algoritmo vuelve atrás (y de ahí su nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. En este retroceso, si existe uno o más caminos aún no explorados que puedan conducir a solución, el recorrido del árbol continúa por ellos.

La filosofía de estos algoritmos no sigue unas reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

Gran parte de la eficiencia (siempre relativa) de un algoritmo de Vuelta Atrás proviene de considerar el menor conjunto de nodos que puedan llegar a ser soluciones, aunque siempre asegurándonos de que el árbol "podado" siga conteniendo todas las soluciones. Por otra parte debemos tener cuidado a la hora de decidir el tipo de condiciones (*restricciones*) que comprobamos en cada nodo a fin de detectar nodos fracaso. Evidentemente el análisis de estas restricciones permite ahorrar tiempo, al delimitar el tamaño del árbol a explorar. Sin embargo esta evaluación requiere a su vez tiempo extra, de manera que aquellas restricciones que vayan a detectar pocos nodos fracaso no serán normalmente interesantes. No obstante, y como norma de actuación general, podríamos decir que las restricciones sencillas son siempre apropiadas, mientras que las más sofisticadas que requieren más tiempo en su cálculo deberían reservarse para situaciones en las que el árbol que se genera sea muy grande.

Vamos a ver como se lleva a cabo la búsqueda de soluciones trabajando sobre este árbol y su recorrido. En líneas generales, un problema puede resolverse con un algoritmo Vuelta Atrás cuando la solución puede expresarse como una *n*-tupla [x_1, x_2, ..., x_n] donde cada una de las componentes x_i de este vector es elegida en cada etapa de entre un conjunto finito de valores. Cada etapa representará un nivel en el árbol de expansión.
En primer lugar debemos fijar la descomposición en etapas que vamos a realizar y definir, dependiendo del problema, la *n*-tupla que representa la solución del problema y el significado de sus componentes x_i. Una vez que veamos las posibles opciones de cada etapa quedará definida la estructura del árbol a recorrer. Vamos a ver a través de un ejemplo cómo es posible definir la estructura del árbol de expansión.

Ejemplos de la Técnica Vuelta atrás:

a) LAS *n* REINAS

Un problema clásico que puede ser resuelto con un diseño Vuelta Atrás es el denominado de las ocho reinas y en general, de las *n* reinas. Disponemos de un tablero de ajedrez de tamaño 8x8, y se trata de colocar en él ocho reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren dos reinas ni en la misma fila, ni en la misma columna, ni en la misma diagonal.

b) LAS PAREJAS ESTABLES

Supongamos que tenemos *n* hombres y *n* mujeres y dos matrices *M* y *H* que contienen las preferencias de los unos por los otros. Más concretamente, la fila *M*[*i*,·] es una ordenación (de mayor a menor) de las mujeres según las preferencias del *i*-ésimo hombre y, análogamente, la fila *H*[*i*,·] es una ordenación (de mayor a menor) de los hombres según las preferencias de la *i*-ésima mujer.
El problema consiste en diseñar un algoritmo que encuentre, si es que existe, un emparejamiento de hombres y mujeres tal que todas las parejas formadas sean *estables*.
Diremos que una pareja (*h*,*m*) es estable si no se da ninguna de estas dos circunstancias:
1) Existe una mujer *m'* (que forma la pareja (*h'*,*m'*)) tal que el hombre *h* la prefiere sobre la mujer *m* y además la mujer *m'* también prefiere a *h* sobre *h*'.
2) Existe un hombre *h*" (que forma la pareja (*h*",*m*")) tal que la mujer *m* lo prefiere sobre el hombre *h* y además el hombre *h*" también prefiere a *m* sobre la mujer *m*".

c) EL LABERINTO

Una matriz bidimensional *n*x*n* puede representar un laberinto cuadrado. Cada posición contiene un entero no negativo que indica si la casilla es transitable (0) o no lo es (∞). Las casillas [1,1] y [*n*,*n*] corresponden a la entrada y salida del laberinto y siempre serán transitables.
Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida.

d) EL COLOREADO DE MAPAS

Dado un grafo conexo y un número $m > 0$, llamamos *colorear* el grafo a asignar un número i ($1 <= i <= m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales.
El nombre de este problema proviene de un problema clásico, el del coloreado de mapas en el plano. Para resolverlo se utilizan grafos puesto que un mapa puede ser representado por un grafo conexo. Cada vértice corresponde a un país y cada arco entre dos vértices indica que los dos países son vecinos. Desde el siglo XVII ya se conoce que con cuatro colores basta para colorear cualquier mapa planar, pero sin embargo existen situaciones en donde no nos importa el número de colores que se utilicen.
Para implementar un algoritmo de Vuelta Atrás, la solución al problema puede expresarse como una n-tupla de valores $X = [x_1, x_2, ..., x_n]$ donde x_i representa el color del i-ésimo vértice. El algoritmo que resuelve el problema trabajará por etapas, asignando en cada etapa k un color (entre 1 y m) al vértice k-ésimo.

e) RECONOCIMIENTO DE GRAFOS

Dadas dos matrices de adyacencia, el problema consiste en determinar si ambas representan al mismo grafo, salvo nombres de los vértices.
Nuestro punto de partida son dos matrices cuadradas L_1 y L_2 que representan las matrices de adyacencia de dos grafos g_1 y g_2. Queremos ver si g_1 y g_2 son iguales, salvo por la numeración en sus vértices.
Podemos suponer sin pérdida de generalidad que la dimensión de ambas matrices coincide. Si no, al estar suponiendo que los vértices de los grafos están numerados de forma consecutiva a partir de 1, ya podríamos decidir que ambos grafos son distintos por tener diferente número de vértices. Sea entonces n la dimensión de ambas matrices.

5) RAMIFICACIÓN Y PODA

Este método de diseño de algoritmos es en realidad una variante del diseño Vuelta Atrás.
Esta técnica de diseño, cuyo nombre en castellano proviene del término inglés *Branch and Bound*, se aplica normalmente para resolver problemas de optimización.
Ramificación y Poda, al igual que el diseño Vuelta Atrás, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.
Una característica que le hace diferente al diseño anterior es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos que el diseño Vuelta Atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones.
El diseño Ramificación y Poda en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo coste o LC).
Además de estas estrategias, la técnica de Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

La técnica de ramificación y poda se utiliza si existe una función "coste" que permita:

- Ordenar los nodos por probabilidad de conducir a la solución

- Detectar los nodos que no pueden conducir a la solución óptima

Los nodos del espacio de búsqueda se pueden etiquetar de tres formas distintas: nodo vivo, nodo muerto y en expansión

.

- Un *nodo vivo* es un nodo factible y prometedor del que no se han generado todos sus hijos.
- Un *nodo muerto* es un nodo del que no se van a generar más hijos por alguna de las tres razones siguientes:
  - Ya se han generado todos sus hijos o
  - No es factible o
  - No es prometedor (un nodo es prometedor si la información que tenemos de ese nodo significa que expandiéndolo se puede conseguir una mejor solución que la mejor solución en curso).

Definimos *nodo vivo* del árbol a un nodo con posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en alguna estructura que podamos recorrer.
Emplearemos una pila para almacenar los nodos que se han generado pero no han sido examinados en una búsqueda en profundidad (LIFO).
Las búsquedas en amplitud utilizan una cola (FIFO) para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden en que son creados.
La estrategia de mínimo coste (LC) utiliza una *función de coste* para decidir en cada momento qué nodo debe explorarse, con la esperanza de alcanzar lo más rápidamente posible una solución más económica que la mejor encontrada hasta el momento. Utilizaremos en este caso una estructura de montículo (o cola de prioridades) para almacenar los nodos ordenados por su coste.

Básicamente, en un algoritmo de Ramificación y Poda se realizan tres etapas:

La primera de ellas, denominada de *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo.
En la segunda etapa, la *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior.
Por último se realiza la tercera etapa, la *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección.
El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

Para cada nodo del árbol dispondremos de una función de coste que nos estime el valor óptimo de la solución si continuáramos por ese camino. De esta manera, si la cota que se obtiene para un nodo, que por su propia construcción deberá ser mejor que la solución real (o a lo sumo, igual que ella), es peor que una solución ya obtenida por otra rama, podemos podar esa rama pues no es interesante seguir por ella.
Evidentemente no podremos realizar ninguna poda hasta que hayamos encontrado alguna solución. Por

supuesto, las funciones de coste han de ser crecientes respecto a la profundidad del árbol, es decir, si *h* es una función de coste entonces $h(n) <= h(n')$ para todo *n*' nodo descendiente de *n*.

En consecuencia, y a la vista de todo esto, podemos afirmar que lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso. Si es demasiado simple probablemente pocas ramas puedan ser excluidas. Dependiendo de cómo ajustemos la función de coste mejor algoritmo se deriva.

Inicialmente, y antes de proceder a la poda de nodos, tendremos que disponer del coste de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un coste mayor. Como muchas veces no se desea esperar a encontrar la primera solución para empezar a podar, un buen recurso para los problemas de optimización es tomar como mejor solución inicial la obtenida con un algoritmo ávido, que como vimos no encuentra siempre la solución óptima, pero sí una cercana a la óptima.
Por último, sólo comentar una ventaja adicional que poseen estos algoritmos:
La posibilidad de ejecutarlos en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda. El disponer de algoritmos paralelizables (y estos algoritmos, así como los de Divide y Vencerás lo son) es muy importante en muchas aplicaciones en las que es necesario abordar los problemas de forma paralela para resolverlos en tiempos razonables, debido a su complejidad intrínseca.

Sin embargo, todo tiene un precio, sus requerimientos de memoria son mayores que los de los algoritmos Vuelta Atrás. Ya no se puede disponer de una estructura global en donde ir construyendo la solución, puesto que el proceso de construcción deja de ser tan "ordenado" como antes. Ahora se necesita que cada nodo sea autónomo, en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y para reconstruir la solución encontrada hasta ese momento.

CONSIDERACIONES DE IMPLEMENTACIÓN
Uno de las dificultades que suele plantear la técnica de Ramificación y Poda es la implementación de los algoritmos que se obtienen. Para subsanar este problema, presentamos una estructura general de tales algoritmos, basada en tres módulos principales:
1. De un lado dispondremos del módulo que contiene el esquema de funcionamiento general de este tipo de algoritmos.
2. Por otro se encuentra el módulo que maneja la estructura de datos en donde se almacenan los nodos que se van generando, y que puede tratarse de una pila, una cola o un montículo (según se siga una estrategia LIFO, FIFO o LC).
3. Finalmente, necesitamos un módulo que describa e implemente las estructuras de datos que conforman los nodos.

Podemos ver que, además de encontrar una solución, el programa calcula tres datos:
a) el número de nodos generados,
b) el número de nodos analizados,
c) el número de nodos podados, los cuales permiten analizar el algoritmo y poder comparar distintas estrategias y funciones LC.

a) El primero de ellos (*numgenerados*) nos da información sobre el trabajo que ha tenido que realizar el algoritmo hasta encontrar la solución. Mientras más pequeño sea este valor, menos parte del árbol de expansión habrá tenido que construir, y por tanto más rápido será el proceso.
b) El segundo valor (*numanalizados*) nos indica el número de nodos que el algoritmo ha tenido que analizar, para lo cual es necesario extraerlos de la estructura y comprobar si han de ser podados y, si no, expandirlos. Éste es el valor más importante de los tres, pues indica el número de nodos del árbol de expansión que se recorren efectivamente. En consecuencia, es deseable que este valor sea pequeño.
c) Por último, el número de nodos podados nos da una indicación de la efectividad de la función de poda y las restricciones impuestas al problema. Mientras mayor sea este valor, más trabajo ahorramos al algoritmo.

Ejemplos de la Técnica: RAMIFICACIÓN Y PODA

a) EL VIAJANTE DE COMERCIO

El problema del viajante de comercio, puede ser resuelto bajo la técnica de Algortimos Ávidos cuyo enunciado es el que sigue:
Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible. Más formalmente, dado un grafo *g* conexo y ponderado, y dado uno de sus vértices v_0, queremos encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v_0.

b) LA ASIGNACIÓN DE TAREAS

El problema de la asignación de tareas puede resolverse también utilizando la técnica de Ramificación y Poda. Recordemos que este problema consiste en, dadas *n* personas y *n* tareas, asignar a cada persona una tarea minimizando el coste de la asignación total, haciendo uso de una matriz de tarifas que determina el coste de asignar a cada persona una tarea.

c) LAS *n* REINAS

Este problema se ha planteado bajo la técnica Vuelta Atrás, y consiste en encontrar una disposición de todas ellas en un tablero de ajedrez de tamaño nxn de forma que ninguna amenace a otra.
Necesitamos resolver este problema utilizando Ramificación y Poda mediante las estrategias FIFO y LIFO, y comparar ambas soluciones.
El estudio del árbol de expansión de este problema ya es conocido, y sólo recordaremos que se basa en construir un vector solución formado por *n* enteros positivos, donde el *k*-ésimo de ellos indica la columna en donde hay que colocar la reina de la fila *k* del tablero. En cada paso o etapa disponemos de *n* posibles opciones a priori (las *n* columnas), pero podemos eliminar aquellas columnas que den lugar a un vector que no sea *k*-prometedor, esto es, que la nueva reina incorporada amenace a las ya colocadas.
Más formalmente, diremos que el vector *s* de *n* elementos es *k*-prometedor (con $1 <= k <= n$) si y sólo si para todo par de enteros *i* y *j* entre 1 y *k* se verifica que:

$$s[i] <> s[j] \quad \text{y} \quad |s[i] - s[j]| <> |i - j|.$$

Esto da lugar a un árbol de expansión razonablemente manejable (del orden de 2000 nodos para $n = 8$) y por tanto convierte el problema en “tratable”.