1. DIVIDE Y VENCERAS: Divide & Conquer

1.1. PRINCIPIOS

El esquema de Divide y Vencerás es una aplicación directa de las técnicas de diseño recursivo de algoritmos. Hay dos rasgos fundamentales que caracterizan los problemas que son resolubles aplicando el esquema de Divide y Vencerás. El primero de ellos es que es necesario que el problema admita una formulación recursiva. Hay que poder resolver el problema inicial a base de combinar los resultados obtenidos en la resolución de un número reducido de subproblemas. Estos subproblemas son del mismo tipo que el problema inicial pero han de trabajar con datos de tamaño estrictamente menor. Y el segundo rasgo es que el tamaño de los datos que manipulan los subproblemas ha de ser lo más parecido posible y debe decrecer en progresión geométrica. Si *n* denota el tamaño del problema inicial entonces ***n/c***, siendo c>0 una constante natural, denota el tamaño de los datos que recibe cada uno de los subproblemas en que se descompone.
Estas dos condiciones están caracterizando un algoritmo, generalmente recursivo múltiple, en el que se realizan operaciones para *fragmentar* el tamaño de los datos y para *combinar* los resultados de los diferentes subproblemas resueltos.

Además, es conveniente que el problema satisfaga una serie de condiciones adicionales para que sea rentable, en términos de eficiencia, aplicar esta estrategia de resolución. Algunas de ellas se enumeran a continuación:

1. En la formulación recursiva nunca se resuelve el mismo subproblema más de una vez.
2. Las operaciones de fragmentación del problema inicial en subproblemas y las de combinación de los resultados de esos subproblemas han de ser eficientes, es decir, han de costar poco.
3. El tamaño de los subproblemas ha de ser lo más parecido posible.
4. Hay que evitar generar nuevas llamadas cuando el tamaño de los datos que recibe el subproblema es suficientemente pequeño. En [BB 90] se discute cómo determinar el tamaño umbral a partir del cual no conviene seguir utilizando el algoritmo recursivo y es mejor utilizar el algoritmo del caso directo.

Existen casos particulares de aplicación del esquema que se convierten en degeneraciones del mismo. Por ejemplo cuando alguna de las llamadas recursivas sólo se produce dependiendo de los resultados de llamadas recursivas previas, o cuando el problema se fragmenta en más de un fragmento pero sólo se produce una llamada recursiva. Esto último sucede en el algoritmo de *Búsqueda dicotómica* sobre un vector ordenado en el que se obtienen dos fragmentos pero sólo se efectúa la búsqueda recursiva en uno de ellos.

El coste del algoritmo genérico DIVIDE_VENCERAS se puede calcular utilizando el segundo teorema de reducción siempre que se haya conseguido que los subproblemas reciban datos de tamaño similar. Entonces, la expresión del caso recursivo queda de la siguiente forma:

$$T_2(n) = k \cdot T_2(n/c) + coste\ (\ MAX(\ FRAGMENTAR,\ COMBINAR\ )\ )$$

Conviene destacar que la utilización de este esquema NO GARANTIZA NADA acerca de la eficiencia del algoritmo obtenido. Puede suceder que el coste empeore, se mantenga o mejore respecto al de un algoritmo,

probablemente iterativo, que ya se conocía de antemano. En los apartados siguientes veremos las tres situaciones posibles.

El algoritmo de Búsqueda dicotómica y el de ordenación rápida, *Quicksort* (Hoare, 1962), son dos aplicaciones clásicas del esquema de Divide y Vencerás (se supone que ambos algoritmos son bien conocidos por el lector). La recurrencia de la Búsqueda dicotómica es $T_{bd}(n) = T_{bd}(n/2) + 1$, y su coste es $\theta(\log n)$. Quicksort presenta un comportamiento menos homogéneo. En él el tamaño de los fragmentos a ordenar depende de la calidad del pivote. Un buen pivote, por ejemplo la mediana, construye dos fragmentos de tamaño *n/2*, pero un mal pivote puede producir un fragmento de tamaño *1* y otro de tamaño *n-1*. En el caso de utilizar un buen pivote la recurrencia es $T_b(n) = 2 \cdot T_b(n/2) + \theta(n)$ lo que da un coste de $\theta(n \cdot \log n)$. En el otro caso la recurrencia es absolutamente diferente, $T_m(n) = T_m(n-1) + \theta(n)$, que tiene un coste de $\theta(n^2)$. Estos son los costes en el caso mejor y el caso peor, respectivamente, del algoritmo del Quicksort. A pesar del coste en el caso peor, su coste en el caso medio es también $\theta(n \cdot \log n)$ lo que le convierte en un algoritmo de ordenación eficiente.

1.2. UN ALGORITMO DE CLASIFICACION: Mergesort

Otro algoritmo interesante y sumamente sencillo, que utiliza la técnica que aquí se presenta, es el algoritmo de clasificación por fusión, *Mergesort* (Von Neumann, 1945). Supongamos que se ha de ordenar una secuencia de *n* elementos. La solución Divide y Vencerás aplicada consiste en fragmentar la secuencia de entrada en dos subsecuencias de tamaño similar, ordenar recursivamente cada una de ellas y, finalmente, fusionarlas del modo adecuado para generar una sola secuencia en la que aparezcan todos los elementos ordenados.

{mergesort.sml}

```
fun    msort [] = []
|      msort [a] = [a]
|      msort lista =

let

       fun    split [] = ([],[])
              |      split [a] =([a],[])
              |      split (x1::x2::resto) =
                     let val (M,N)= split(resto)
                     in
                            (x1::M,x2::N)
                     end;
       fun    merge (a,  nil) = a
              |      merge (nil, a) = a
              |      merge (L1 as x::xs,L2 as y::ys) =
                            if x < y then x::merge(xs,L2) else y::merge(L1,ys);

       val (M,N) = split lista;
              val M = msort M;
              val N = msort N
in
              merge(M,N)
end;
```

La implementación tradicional de la función MERGE requiere un coste $\theta(n)$ y consiste en procesar secuencialmente las dos subsecuencias ordenadas de la entrada. A cada paso se decide de qué

subsecuencia de la entrada se elige el primer elemento para que vaya a la secuencia de salida, al lugar que ocupará definitivamente. Usando una fusión de este tipo el Mergesort tiene un coste de $\theta(n \cdot \log n)$ que se obtiene al resolver la recurrencia $T(n)= 2 \cdot T(n/2) + \theta(n)$.

2. ALGORITMOS VORACES o AVIDOS: *Greedy Algotithms*

Los algoritmos para problemas de optimización típicamente evolucionan por una secuencia de pasos, con un conjunto de opciones o elecciones en cada paso. Los algoritmos voraces, siempre hacen la elección que parece ser la mejor en ese momento. Esto significa, hacen una ***elección óptima local*** con la esperanza de que esta elección conduzca a la solución óptima global.
Los algoritmos ávidos no siempre conducen a soluciones óptimas, pero para muchos problemas si lo hacen.

2.1 Caracterización del Método

a) Se utilizan generalmente en la resolución de problemas de optimización (obtener el máximo o el mínimo de una función).
b) Toman decisiones en función de la información que está disponible en cada momento
c) Una vez tomada una decisión, ésta no vuelve a replantearse en el futuro.
d) Suelen ser rápidos y fáciles de implementar.
e) No siempre garantizan alcanzar la solución óptima. , por lo tanto siempre se debe estudiar la ***Corrección del Algoritmo***, para demostrar si las soluciones obtenidas son óptimas o no.

Para poder resolver un problema utilizando el enfoque ávido, tendremos que considerar 6 elementos:

1. **Conjunto de candidatos** (elementos seleccionables)
2. **Solución parcial** (candidatos seleccionados)
3. **Función de selección** (determina el mejor candidato del conjunto de candidatos seleccionables).
4. **Función de factibilidad** (determina si es posible completar la solución parcial para alcanzar una solución del problema).
5. **Criterio que define lo que es una solución** (indica si la solución parcial obtenida resuelve el problema).
6. **Función Objetivo** (valor de la solución alcanzada)

Habiendo definido los elementos propios de un enfoque greedy, ahora trataremos de describir el esquema general:

- Se parte de un conjunto solución vacío: S = Ø.
- De la lista de candidatos, se elige el mejor (de acuerdo con la **Función de Selección**)
- Comprobamos si se puede llegar a una solución con el candidato seleccionado (**Función de Factibilidad**). Si no es así, lo eliminamos de la lista de candidatos y nunca más lo consideraremos.
- Si aún no hemos llegado a una solución, seleccionamos otro candidato y repetimos el proceso hasta llegar a una solución.

Para entender mejor el enfoque voraz, vamos a tratar de identificar los elementos del esquema en algoritmos bien conocidos por nosotros, y desarrollados en cursos anteriores.
Consideremos el problema de encontrar el Árbol Abarcador de Costo Mínimo de un grafo plano conexo. Para su resolución conocemos dos algoritmos, Prim y Kruskal. Ambos son enfoques greedy basados en la propiedad MST, *"… Sea un grafo conexo G = (V,E) con una función costo definida sobre sus aristas. Sea U algún*

subconjunto adecuado de los vértices V. Si (u, v) *es una arista de costo mínimo, tal que* $u \in U$ *y* $v \in V - U$*, luego existe un Arbol Abarcador de Costo Mínimo que incluye a la arista* (u, v)*.*[1]"

Es necesario tener en cuenta que algunos elementos y/o pasos pueden ser triviales en algoritmos que emplean un enfoque voraz. Por ejemplo en algoritmo de Prim, no necesita de una función de factibilidad ya que la función de selección directamente elige aristas factibles. Sin embargo el algoritmo de Kruskal en ese sentido es más gráfico, ya que selecciona una arista de costo mínimo(u, v) y luego se debe validar que el conjunto al que pertenece el vértice u, sea distinto al que pertenece el vértice v, esto conforma una función de factibilidad.

2.2 EL PROBLEMA DEL CAMBIO

Dado un conjunto *C* de *N* tipos de monedas con un número inagotable de ejemplares de cada tipo, hay que conseguir, si se puede, formar la cantidad *M* empleando el MÍNIMO número de ellas.

Este es un problema que satisface el principio de optimalidad, ("… *un problema presenta estructura óptima, si una solución óptima a él, incluye o contiene las soluciones óptimas a subproblemas..."* [2])

Se puede demostrar que si se tiene una solución óptima ***A={e1,e2, ..., ek}*** para el problema de formar la cantidad ***M***, siendo los ***ei*** , $1 \leq i \leq k$, ejemplares de las monedas de ***C***, entonces sucede que ***A-{e1}*** es también solución óptima para el problema de formar la cantidad ***M - e1***. Como se establecerá más adelante, en estas condiciones puede pensarse en un Voraz como esquema aplicable.

Para el problema que nos ocupa se puede comenzar por definir *factible* y *solución*. En este caso, el problema no tiene restricciones y se tiene una solución cuando los ejemplares de monedas elegidas suman exactamente *M*, si es que esto es posible.

La forma en que habitualmente se devuelve el cambio nos ofrece un candidato a función de selección: *elegir a cada paso la moneda disponible de mayor valor*.

La función factible tendrá que comprobar que el valor de la moneda seleccionada junto con el valor de las monedas de la solución en curso no supera *M*.

Inmediatamente se observa que para facilitar el trabajo de la función de selección se puede ordenar el conjunto *C*, de tipos de monedas de la entrada, por orden decreciente de valor. De este modo, la función de selección sólo tiene que tomar la primera moneda de la secuencia ordenada.

Cubierta la primera parte del proceso, ahora hay que demostrar la optimalidad de este criterio y basta con un contraejemplo para comprobar que no conduce siempre al óptimo.

En el siguiente ejemplo, demostramos que tal como fue enunciado el problema y sin condiciones adicionales, no podemos asegurar que siempre se consiga un resultado óptimo.

{Cambio.sml}

```
exception No_Hay_Cambio;
fun cambio 0  Cr = []
|          cambio M  [] = raise No_Hay_Cambio
|          cambio M  (L as e::t)  =
           if  e <= M then e::(cambio(M - e) L)
           else cambio M t;
```

Sea

`val C = [50, 11, 5, 1]` la lista de valores de monedas disponibles.

[1] Estructuras de Datos y Algoritmos: Aho, Hoppcroft, Hullman

[2] Introduction to Algorithms : T.Cormen, C.Leiserson, R.Rivest

Entonces `cambio 65 C`, da como resultado `[1, 1, 1, 1, 11, 50]`, lo cual difiere del resultado óptimo que sería `[5, 5, 5, 50].`

No está todo perdido porque se puede intentar caracterizar *C* de modo que esta estrategia funcione siempre correctamente. En primer lugar se necesita que *C* contenga la monedad unidad. Caso de que esta moneda no exista, no se puede garantizar que el problema tenga solución y, tampoco, que el algoritmo la encuentre. En segundo lugar *C* ha de estar compuesto por tipos de monedas que sean potencia de un tipo básico *t*, con $t > 1$. Con estas dos condiciones sobre *C*, el problema se reduce a encontrar la descomposición de *M* en base *t*. Se sabe que esa descomposición es única y, por tanto, mínima con lo que queda demostrado que en estas condiciones este criterio conduce siempre al óptimo.

2.3 MOCHILA

Se dispone de una colección de *n* objetos y cada uno de ellos tiene asociado un peso y un valor. Más concretamente, se tiene que dado el objeto *i* con $1 \leq i \leq n$, su valor es v_i y su peso es p_i. También se tiene una mochila capaz de soportar, como máximo, el peso PMAX. Determinar qué objetos hay que colocar en la mochila de modo que el valor total que se transporte sea máximo pero sin que se sobrepase el peso máximo, PMAX, que puede soportar.

Este problema es conocido por *The Knapsack problem*. Existen dos posibles formulaciones de la solución:

- *Mochila fraccionaria*: se admite fragmentación, es decir, se puede colocar en la solución un trozo de alguno de los objetos.
- *Mochila entera*: NO se admite fragmentación, es decir, un objeto o está completo en la solución o no está en ella.

Para ambas formulaciones el problema satisface el principio de optimalidad. Ahora bien, mochila fraccionaria se puede resolver aplicando un voraz mientras que, de momento, mochila entera no (se necesita un Bactracking y, si se introduce alguna otra restricción en el problema, es posible incluso aplicar Programación Dinámica).

Intentaremos, por tanto, encontrar una buena función de selección que garantice el óptimo global para el problema de mochila fraccional. Formalmente se pretende:

$$[\boldsymbol{MAX}] \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{x_i} * \boldsymbol{v_i}$$

Sujeto a

$$\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{x_i} * \boldsymbol{p_i} \leq \boldsymbol{Pmax} \ , \boldsymbol{0 \leq x_i \leq 1}$$

La solución al problema viene dada por el conjunto de valores x_i, con $1 \leq i \leq n$, siendo x_i un valor real entre 0 y 1 que se asocia al objeto *i*. Así, si el objeto *3* tiene asociado un 0 significará que este objeto no forma parte de la solución, pero si tiene asociado un 0.6 significará que un 60% de él está en la solución.

Es posible formular unas cuantas funciones de selección. De las que se presentan a continuación las dos primeras son bastante evidentes mientras que la tercera surge después del fracaso de las otras dos:

i. Seleccionar los objetos por orden creciente de peso. Se colocan los objetos menos pesados primero para que la restricción de peso se viole lo más tarde posible. Con un contraejemplo se ve que no funciona.

ii. Seleccionar los objetos por orden decreciente de valor. De este modo se asegura que los más valiosos serán elegidos primero. Tampoco funciona.

iii. Seleccionar los objetos por orden decreciente de relación valor/peso. Esto surgiría de considerar una variación del segundo caso, tratando de tener en cuenta las restricciones de peso. Como el valor total de la mochila cargada con un solo tipo de objeto i, está dado por $\frac{P_{max}}{p_i} * v_i$, vemos claramente que la evaluación de un elemento i no depende ni de su peso (p_i) ni de su valor (v_i) sino de una combinación de ambos en la relación valor/peso (v_i / p_i) Así se consigue colocar primero aquellos objetos cuya unidad de peso tenga mayor valor. Parece que si funciona.

*La demostración de la optimalidad, de la solución en **Apendice (D1)***

Ej:

Mochila con **Pmax = 100Kg**
Objetos :

i		*1*	*2*	*3*	*4*	*5*
v_i	Beneficio [$]	20	30	65	40	60
p_i	Peso [Kg]	10	20	30	40	50
v_i/p_i	Benef/Peso	2	1.5	2.17	1	1.2

Veamos como resultan las distintas funciones de selección

i. Los elegimos por peso creciente (primero los más livianos)

Peso = (1*10) + (1*20) + (1*30) + (1*40) = 100 Kg
Beneficio = (1*20) + (1*30)+ (1*65) + (1*40) = **$155**
S = {1, 1, 1, 1, 0}

ii. Los elegimos por valor decreciente (primero los más valiosos)

Peso = (1*30) + (1*50) + (0.5*40) = 100 Kg
Beneficio = (1*65) + (1*60) + (0.5*40) = **$145**
S = {0, 0, 1, 0.5, 1}

iii. En función decreciente de la relación Beneficio/Peso (mayor v_i/p_i primero)

Peso = (1*30) + (1*10) + (1*20) + (0.8*50) = 100Kg
Beneficio = (1*65) + (1*20) + (1*30) + (0.8*60) = **$ 163**
S = {1, 1, 1, 0, 0.8}

{knapsackfracc.sml}

```
(* C es la lista de objetos (id_obj, valor, peso, v/p)
 *)
```

```
val C = [(3, 65.0, 30.0, 65.0/30.0),(1, 20.0, 10.0, 2.0),(2, 30.0, 20.0, 1.5),(5, 60.0,
50.0, 1.2),(4, 40.0, 40.0, 1.0)];

fun ksf(Pmax, (i,vi,pi,r)::t, V, S, p) =
      if p = Pmax then (S,V)
      else
            if ( pi <= (Pmax - p)) then ksf(Pmax, t, V+vi,S @ [(i,1.0)], p + pi)
            else
                  let val prop = (Pmax - p)/pi
                  in ksf(Pmax, t, V + prop * vi ,S @ [(i,prop)],Pmax)
                  end;

fun mochila (Pmax,g)=
let
      val (L,valor) = ksf(Pmax, g, 0.0,[],0.0)
in
      (print("\nValor Maximo= "^makestring(valor)^"\nSolucion:");
       L)
end;
```

3. Búsqueda con Retroceso: Backtracking
Ramificación y Poda: Branch & Bound

El backtracking es una estrategia usada para encontrar soluciones a problemas que tienen una solución completa, en los que el orden de los elementos no importa, y en los que existen una serie de variables a cada una de las cuales debemos asignarle un valor teniendo en cuenta unas restricciones dadas. El término fue acuñado por el matemático D. H. Lehmer en la década de los cincuenta y formalizado por Walker, Golom y Baumert en el siguiente decenio. El NIST (U.S. National Institute of Standards and Technology) lo define en su *Diccionario de Algoritmos y Estructuras de Datos* [1] de la siguiente manera:

> *Find a solution by trying one of several choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice.*
>
> Traducción:
>
> *Encontrar una solución intentándolo con una de varias opciones. Si la elección es incorrecta, el cómputo retrocede o vuelve a comenzar desde el punto de decisión anterior y lo intenta con otra opción.*

Este esquema algorítmico es utilizado para resolver problemas en los que la solución consta de una serie de decisiones adecuadas hacia nuestro objetivo. El backtracking está muy relacionado con la búsqueda combinatoria. De manera más exacta podemos indicar que los problemas a considerar son los siguientes:

1. *Problemas de Decisión:* Búsqueda de las soluciones que satisfacen ciertas restricciones.
2. *Problemas de Optimización*: Búsqueda de la mejor solución en base a una función objetivo.

De forma general, el método del backtracking, concebido como tal, genera todas las secuencias de forma sistemática y organizada, de manera que prueba todas las posibles combinaciones de un problema hasta que encuentra la correcta. En general, la forma de actuar consiste en elegir una alternativa del conjunto de opciones en cada etapa del proceso de resolución, y si esta elección no funciona (no nos lleva a ninguna solución), la búsqueda vuelve al punto donde se realizó esa elección, e intenta con otro valor. Cuando se han agotado todos los posibles valores en ese punto, la búsqueda vuelve a la anterior fase en la que se hizo otra elección entre valores. Si no hay más puntos de elección, la búsqueda finaliza.

Para implementar algoritmos con Backtracking, se usan llamadas a funciones recursivas, en la que en cada llamada la función asigna valores a un determinado punto de la posible solución, probando con todos los valores posibles, y manteniendo aquella que ha tenido éxito en las posteriores llamadas recursivas a las siguientes partes de la solución.

3.1 Planteamiento del problema

Se trata de hallar todas las soluciones que satisfagan un predicado P.

La solución debe poder expresarse como una tupla $(x_1,\ldots, x_n)$ donde cada x_i pertenece a un C_i .

Si $|C_i| = t_i$, entonces hay

$$t = \prod_{i=1}^{n} ti$$

n-tuplas que satisfacen P.

Método de ***fuerza bruta***: examinar las t n-tuplas y seleccionar las que satisfacen P.

Búsqueda con retroceso (backtracking): formar cada tupla de manera progresiva, elemento a elemento, comprobando para cada elemento $\boldsymbol{x_i}$, añadido a la tupla$(x_1,\ldots,x_i)$, que puede conducir a una tupla completa satisfactoria.

Deben existir funciones objetivos parciales o predicados acotadores $P_i(x_1,\ldots,x_i)$. Los cuales dicen si la tupla $(x_1,\ldots,x_i)$ puede conducir a una solución.

Diferencia entre Fuerza Bruta y Backtracking:

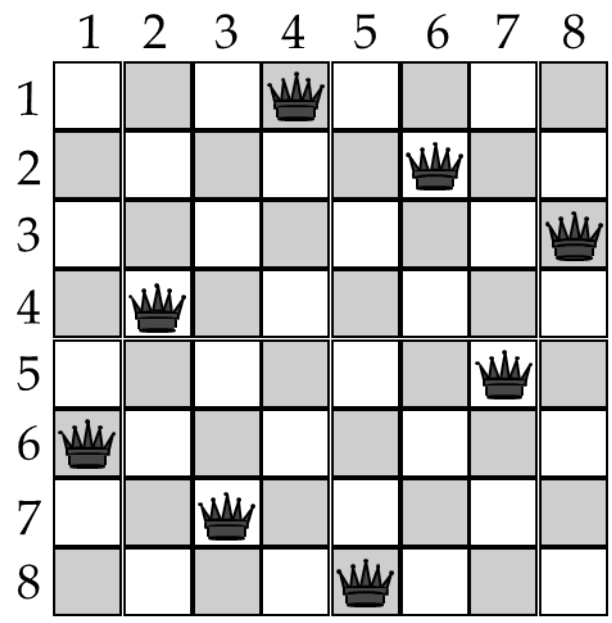
Si se comprueba que la tupla $(x_1,\ldots,x_i)$ no puede conducir a ninguna solución se evitan formar las $t_{i+1} \times \ldots \times t_n$ *tuplas que comienzan con* $(x_1,\ldots,x_i)$.

Para saber si una n-tupla es solución, suele haber dos tipos de restricciones

- Explícitas: describen el conjunto C_i de valores que puede tomar x_i (todas las tuplas que satisfacen estas restricciones definen un **espacio de soluciones posibles**)
- Implícitas: describen las relaciones que deben cumplirse entre los x_i (qué soluciones posibles satisfacen el predicado objetivo *Sol*).

3.2 El Problema de las Ocho Reinas

El problema consiste en colocar ocho reinas en un tablero de ajedrez sin que se den jaque (dos reinas se dan jaque si comparten fila, columna o diagonal).



Formulación 1: $\binom{64}{8} = 4.426.165.368$ soluciones posibles probando por fuerza bruta todas las celdas del tablero (64 celdas tomadas de a 8).

Formulación 2: Puesto que no puede haber más de una reina por fila, podemos replantear el problema como: *"colocar una reina en cada fila del tablero de forma que no se den jaque"*.

En este caso, para ver si dos reinas se dan jaque basta con ver si comparten columna o diagonal.

Por lo tanto, toda solución del problema puede representarse con una 8-tupla $(x_1,\ldots,x_8)$ en la que x_i es la columna en la que se coloca la reina que está en la fila *i* del tablero. ***El espacio de soluciones consta de 8^8 8-tuplas - (16.777.216 8-tuplas)***

Formulación 3: Puesto que no puede haber más de una reina por columna, sólo hace falta que consideremos las 8-tuplas $(x_1,\ldots,x_8)$ que sean permutaciones de (1,2,...,8)

El espacio de soluciones consta de 8! 8-tuplas - (40.320 8-tuplas)[3]

3.3 El Problema de la Suma de Subconjuntos

[3] Una solución del problema implementada en ML, podemos encontrarla en *"ML for the Working Programmer"* 2nd Edition – Lawrence Paulson – Cap. 5.19

Dado un conjunto $W = \{w_1, \ldots, w_n\}$, de ***n*** números enteros positivos y otro entero positivo ***M***, se trata de encontrar todos los subconjuntos de ***W*** que suman ***M***.

Cada solución puede representarse con una *n*-tupla $(x_1,x_2,\ldots,x_n)$, tal que $x_i \in \{0,1\}$, $1 \le i \le n$,

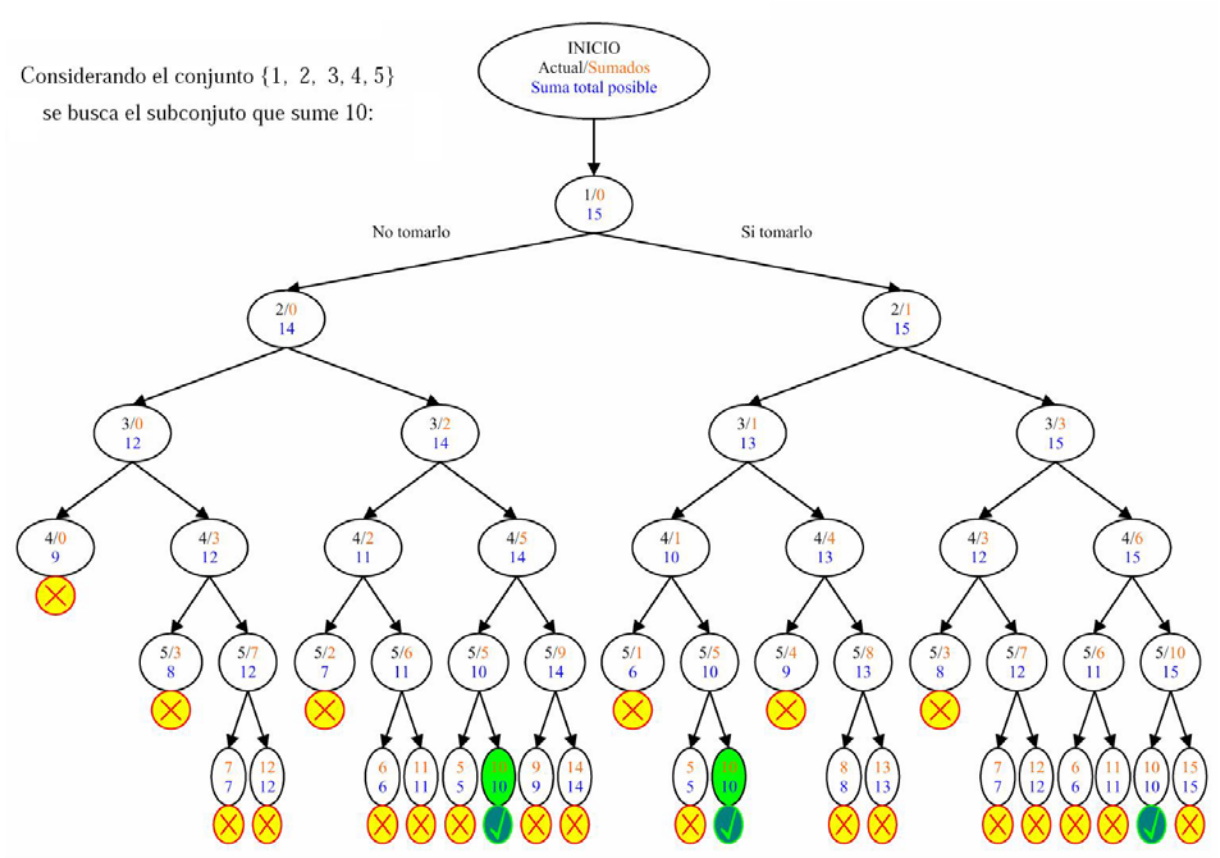
de forma que:

- $x_i = 0$ si w_i no se elige y
- $x_i = 1$ si w_i se elige.

Lo importante es que cualquiera sea la formulación, siempre conduce a un espacio de soluciones de tamaño 2^n tuplas.

Veamos cómo sería el árbol de soluciones para un conjunto de tamaño 5, donde

W= { 1, 2, 3, 4, 5}, y ***M*** = 10.



En este grafico podemos observar, un árbol de valores que se construye a partir de la consideración, o no, de cada uno de los w_i en la suma. Paralelamente a la evaluación sistemática de cada uno de los elementos del conjunto W, vamos llevando dos resultados parciales que utilizaremos como argumentos de la función acotadora.

En cada nodo podemos ver, el w_i que se está evaluando, cuanto es la sumatoria de los w_j con $1 \le j < i$, y cuanto es lo que se puede sumar contando los elementos ya considerados y los que quedan por evaluar.

Es evidente que cuando la suma posible se encuentra por debajo del valor consigna M, no será posible encontrar una solución, por tanto se recorta la rama. Lo mismo ocurre cuando el valor suma actual es igual al valor de consigna M.

```
{subconj.sml}
(* En este ejemplo se trata el problema de encontrar todos los subconjuntos de C para los
cuales la suma de sus integrantes es 10 *)
```

```
val C = [1, 2, 3, 4, 5];
val objetivo = 10;

(* Para registrar el estado de solucion tenemos ( i , sa, sp) donde
i: valor actual , sa: suma actual , sp: suma posible

Es importante considerar que la suma de los n miembros de C es mayor que el valor buscado,
en este caso 10. Ver arbol de soluciones
*)

val sum_max = foldr (fn (x,y) => x + y)  0 C ;

(* Primer caso, utilizando Fuerza Bruta:
se prueban cada una de las posibles soluciones *)

fun fzabruta( [], sa, sp, lista_incluidos) = if sa = objetivo then [lista_incluidos] else
[[]]
|   fzabruta ( x::t, sa, sp, L) =
              fzabruta( t , sa , sp - x, L) @ fzabruta( t, sa + x , sp , x::L);


(* Backtracking:
Acotando la expansion del arbol en esa rama cuando no es posible llegar a un resultado.
En este caso si la suma posible de alcanzar por esa rama es inferior al objetivo, se detiene *)

fun acotadora ( sp , M) = if sp >= M then true else false;


(* Una suma es solucion cuando es exactamente igual al objetivo, cualquier estado posterior
no conducira a una solucion, por lo tanto se cancela. *)

fun Es_sol  ( sa , M) = if sa = M then true else false;


(* Finalmente el algoritmo de resolucion*)
fun sub2([], M, sa, sp, S, soluciones) = if Es_sol(sa,M) then S::soluciones
                                         else soluciones
|   sub2(wi::W, M, sa, sp, S, soluciones) =
        if acotadora (sp, M) = false then soluciones else
           if Es_sol (sa, M) then S:: soluciones else
              sub2(W,M,sa,sp-wi,S,soluciones)@ sub2(W,M,sa+wi,sp,wi::S,soluciones);

(* sub2(C, objetivo, 0, sum_max, [], []); *)

fun subconjunto2(W, obj) = sub2(W, obj, 0, foldr op+ 0 C,[], []);


(* ------------ Ejecucion -------- *)
print ("\n\n\nFUERZA BRUTA\n");

fzabruta( C, 0, sum_max, []);

print ("\n\n\nBACKTRACKING\n");

subconjunto2( C, objetivo);
```

3.4 El Problema de la Mochila Entera o Mochila 0-1

Recordando: Tenemos *n* objetos y una mochila.

El objeto *i* tiene un peso p_i y su inclusión en la mochila produce un beneficio v_i.

El objetivo es llenar la mochila de capacidad C, a los efectos de que se maximice el beneficio.

$$maximizar \sum_{i=1}^{n} v_i \cdot x_i$$

$$sujeto\ a \sum_{i=1}^{n} p_i \cdot x_i \leq C$$

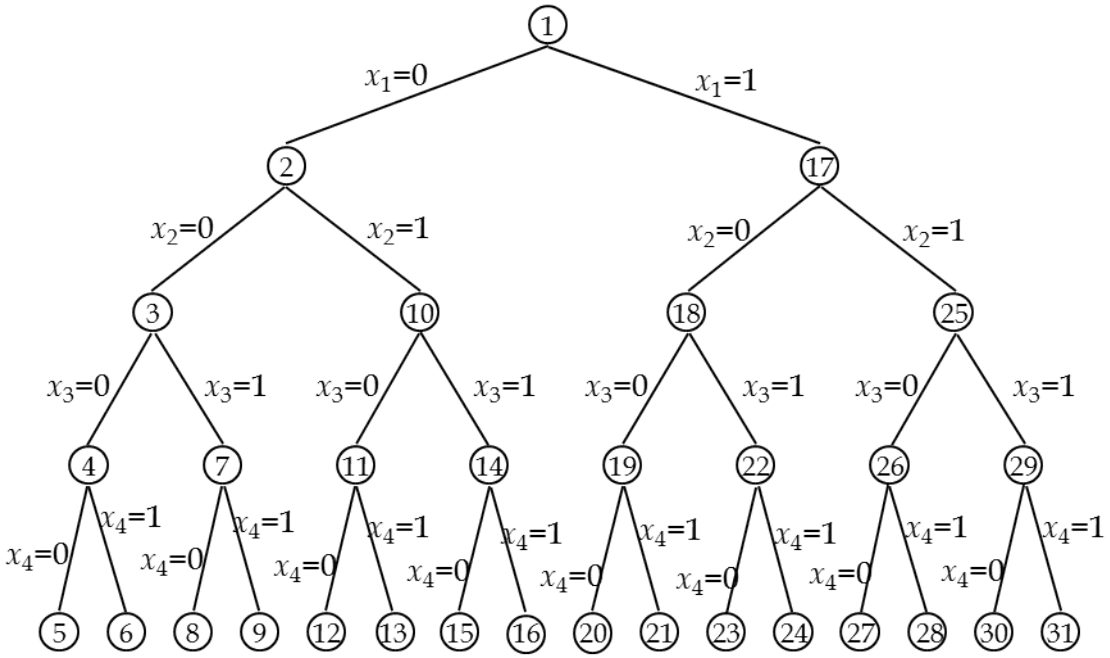
$$con\ x_i \in \{0,1\}, \quad v_i > 0, \quad p_i > 0, \quad 1 \leq i \leq n$$

Este será el primer ejemplo de problema de optimización que planteamos resolver mediante backtracking.

Consideraremos tuplas de tamaño fijo,

$x_i = 0$, el objeto i-ésimo no se incluye en la solución

$x_i = 1$, el objeto i-ésimo se incluye en la solución.



Consideraciones:

- Problema de optimización (maximización), sólo nos interesa la mejor solución.
- En todo momento guardamos el beneficio de la mejor solución encontrada hasta el momento **MS**, que resulta ser una cota inferior de la mejor solución.
- En la ramificación, sólo exploraremos donde podamos mejorar respecto a la mejor solución (poda basada en la mejor solución, PBMS)

Formalmente:

Sea *X [1 … k-1]*, la asignación actual en curso.

Sea ***B*** el beneficio de la mejor solución que hemos encontrado en lo que llevamos de búsqueda (si aún no hemos encontrado ninguna ***B = 0***).

Supongamos que ***C(X,k)*** es el beneficio de la mejor solución que podemos obtener extendiendo la elección actual *X [1 … k-1]*.

Si ***Cota(X,k)*** es una cota superior de ***C(X,k)*** entonces se verifica que ***Cota(X,k)*** ≥ ***C(X,k)***, para todo ***X*** *[1 … k-1]*.

Entonces si ***Cota(X,k)*** ≤ ***B***, significa que el beneficio no se puede mejorar siguiendo por ese camino, por lo tanto hacemos una *"poda"* y volvemos hacia atrás (backtracking).

Ahora bien, la pregunta que nos estamos haciendo es ¿Cómo calculamos la ***Cota(X,k)***?

La respuesta es, relajando los requisitos de integridad en las decisiones que aún no hemos hecho:

$$x_i \in \{0,1\}\ se\ sustituye\ por\ 0 \leq\ x_i\ \leq 1, para\ los\ i\ tales\ que\ k \leq i \leq n$$

Y así aplicamos el algoritmo voraz visto en el caso fraccionario

```
(*
Cota ( lista_objetos_k_a_n, Capacidad_restante, beneficio_hasta_el_momento)
La lista de objetos esta ordenada en forma decreciente vi/pi
*)

fun    cota([], C, B) = B
|      cota(L, 0.0, B) = B
|      cota((id,vi,pi)::t, C, B) =
       if pi > C then B + vi*C/pi
       else cota(t, C - pi, B + vi);
```

Apéndice

(D1) Ahora se ha de demostrar la optimalidad de este criterio.

Demostración:

Sea X=(x_1, x_2, ...,x_n) la secuencia solución generada por el algoritmo Voraz aplicando el criterio de seleccionar los objetos por orden decreciente de relación valor/peso. Por construcción, la secuencia solución es de la forma X=(1, 1, 1, ..., 1, *0.x*, 0,..., 0, 0), es decir, unos cuantos objetos se colocan enteros, aparecen con valor 1 en la solución, un sólo objeto se coloca parcialmente, valor *0.x*, y todos los objetos restantes no se colocan en la mochila, por tanto valor 0.

Si la solución *X* es de la forma $\forall i : 1 \leq i \leq n : x_i=1$, seguro que es óptima ya que no es mejorable. Pero si *X* es de la forma $\exists i : 1 \leq i \leq n : x_i \neq 1$, entonces no se sabe si es óptima.

Sea *j* el índice más pequeño tal que ($\forall i : 1 \leq i < j : x_i=1$) y ($x_j \neq 1$) y ($\forall i : j+1 \leq i \leq n : x_i=0$). El índice *j* marca la posición que ocupa el objeto que se fragmenta en la secuencia solución.

- Supongamos que *X* NO es óptima. Esta suposición implica que existe otra solución, *Y*, que obtiene más beneficio que *X*, es decir,

$$\sum_{i=1}^{n} x_i * v_i < \sum_{i=1}^{n} y_i * v_i$$

Además, por tratarse de mochila fraccionada, ambas soluciones satisfacen :

$$\sum_{i=1}^{n} x_i * p_i = \sum_{i=1}^{n} y_i * p_i = P_{max}$$

Sea *k* el índice más pequeño de *Y* tal que $x_k \neq y_k$. Se pueden producir tres situaciones :

1/ $k < j$. En este caso $x_k=1$ lo que implica que $x_k > y_k$

2/ $k = j$. Forzosamente $x_k > y_k$ porque en caso contrario la suma de los pesos de *Y* sería mayor que PMAX y, por tanto, *Y* no sería solución.

3/ $k > j$. Imposible que suceda porque implicaría que la suma de los pesos de *Y* es mayor que PMAX y, por tanto, *Y* no sería solución.

Del análisis de los tres casos se deduce que si *Y* es una solución óptima distinta de *X*, entonces $x_k > y_k$. Como $x_k > y_k$, se puede incrementar y_k hasta que $x_k=y_k$, y decrementar todo lo que sea necesario desde y_{k+1} hasta y_n para que el peso total continúe siendo PMAX. De este modo se consigue una tercera solución Z= (z_1,z_2, ...,z_n) tal que ($\forall i : 1 \leq i \leq k : z_i=x_i$) y

$$\sum_{i=k+1}^{n} p_i(y_i - z_i) = p_k(z_k - y_k)$$

Para *Z* se tiene :

$$\sum_{i=1}^{n} z_i \cdot v_i = \sum_{i=1}^{n} y_i \cdot v_i + \frac{v_k \cdot p_k}{p_k} \cdot (y_k - z_k) - \sum_{i=k+1}^{n} \frac{v_i \cdot p_i}{p_i} \cdot (y_i - z_i) \geq \sum_{i=1}^{n} y_i \cdot v_i + \frac{v_k}{p_k} \cdot (p_k \cdot (z_k - y_k) - \sum_{i=k+1}^{n} p_i \cdot (y_i - z_i)) = \sum_{i=1}^{n} y_i \cdot v_i$$

Después de este proceso se llega a la siguiente desigualdad:

$$\sum_{i=1}^{n} z_i \cdot v_i \geq \sum_{i=1}^{n} y_i \cdot v_i$$

Pero, en realidad, sólo es posible la igualdad ya que el menor indicaría que *Y* no es óptima lo que contradice la hipótesis de partida. Se puede concluir que *Z* también es una solución óptima y que, a base de modificarla como se ha hecho en este proceso, se consigue una solución, *T*, idéntica a *X*, y por tanto, *X* también es óptima.