4 PROGRAMACIÓN DINÁMICA

La programación dinámica es una técnica para solucionar problemas en la que, partiendo de un algoritmo cuya implantación directa resultaría ineficiente en tiempo, se definen estructuras de datos adicionales que sirven para almacenar cálculos con el objeto de que no sean repetidos o, incluso, de evitar que otros cálculos innecesarios sean realizados.

Es decir, se cambia espacio por tiempo. Si el algoritmo que se mejora es iterativo, la forma en que la operación del programa se acelera corresponde a "poner a trabajar algún invariante", en contraposición a que sea el programa el que trabaje.

Para usar la técnica, en la solución de un problema específico:

- Se define un *lenguaje* para establecer formalmente el problema. Se incluye notación para una función `f`, cuyo cálculo en un valor específico x_0, corresponda a la solución del problema.
- Se establece una *recurrencia que defina la función* `f` en un dominio que incluya a x_0.
- Se estudia la recurrencia, determinando un *diagrama de necesidades*, i.e., para cada elemento `x` del dominio de `f`, establecer para qué elementos del dominio deberá conocerse el valor de `f` para poder calcular `f(x)`.
- El diagrama de necesidades sugiere un *invariante para un ciclo* que calclule todos los valores previos necesarios para evaluar $f(x_0)$. Además, un *orden de evaluación* para los elementos del dominio de `f`.

Es común encontrar usos de esta técnica en el cálculo de funciones recurrentes, aun antes de imaginar algún programa iterativo que lo haga. En contraste con la simple implantación, en un algoritmo recurrente, que pueda sugerir el "dividir y conquistar", la complejidad temporal puede rebajar considerablemente. Por otra parte, puesto que en las estructuras de datos auxiliares se almacenan cálculos parciales ya realizados, el orden de evaluación es, en general, "hacia adelante". En otras palabras, se busca calcular lo necesario (y ojalá sólo esto!) para lograr el objetivo, sin dejar operaciones pospuestas.

El uso de la programación dinámica se ilustrará con ejemplos clásicos como el problema del morral (inglés: *knapsack*) y el del análisis sintáctico.

4.1 EL PROBLEMA DEL MORRAL

El *problema del morral* (inglés: *knapsack*) es un problema clásico de la investigación de operaciones. Una solución recurrente ingenua puede resultar demasiado onerosa. La solución que se presenta, utilizando programación dinámica, es eficiente y práctica[1].

Problema:
Dados

- `n` objetos, $o_1,\ldots,o_n$.
- Para $i = 1,\ldots,n$, p_i es el peso de o_i. Naturalmente, $p_i>0$.
- Un morral, que soporta un peso máximo `P`.
- Para $i = 1,\ldots,n$, si se carga en el morral el objeto o_i, se obtiene un beneficio o utilidad u_i.

Se quiere maximizar la utilidad de cargar en el morral algunos de los objetos.

El lenguaje:

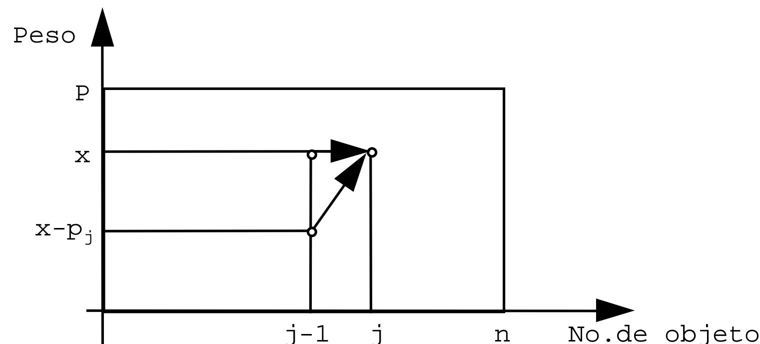
`ut(j,x)` : Utilidad máxima que se consigue, si los objetos que se pueden cargar se eligen entre $o_1,\ldots,o_j$, si el peso máximo que se puede soportar es `x`.

`ut(n,P)` : la respuesta deseada.

La recurrencia:

$$\begin{aligned} ut(j,x) &= 0 &&, \text{ si } j=0 \\ &= ut(j-1,x) &&, \text{ si } 1\le j\le n,\ 0\le x<p_j \\ &= \max\{ut(j-1,x),\ ut(j-1,x-p_j) + u_j\} &&, \text{ si } 1\le j\le n,\ 0<p_j\le x \end{aligned}$$

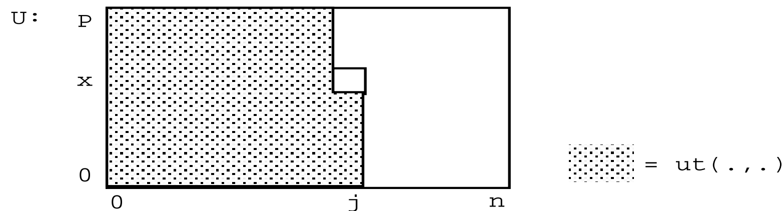
El diagrama de necesidades:



[1] A pesar de ser un problema NP-completo (cf. Intratabilidad), la solución que se presenta es eficiente en la praxis, si se concede que los números involucrados no sean arbitrariamente grandes.

El algoritmo

```
[Ctx: U[0..n,0..P]
   U[0,.]:= 0;
   j,x:= 1,0;
   {Inv:
```




```
   }
   do j≠n+1    →   if   x<pj →   U[j,x]:= U[j-1,x]
                   []   x≥pj →   U[j,x]:= max(U[j-1,x],U[j-1,x-pj]+uj)
                   fi;
                   if   x=P  →   j,x:= j+1,0
                   []   x≠P  →   x:= x+1
                   fi
   od
   {Pos:  U[n,P] = ut(n,P)}
]
```

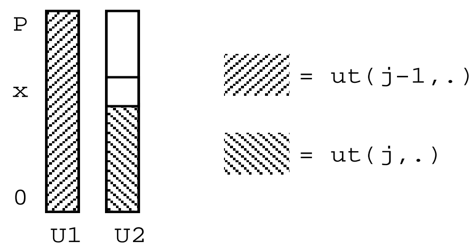
Así:

```
T(n,P) = O(nP)
S(n,P) = nP
```

Mejoras:

El algoritmo puede mejorarase, si se observa con más cuidado el diagrama de necesidades: basta guardar únicamente las dos últimas columnas calculadas:

```
[Ctx: U1[0..P], U2[0..P], pn+1=0
   U1,U2:= 0,0;
   j,x:= 1,p1;
   {Inv:  0≤j≤n+1  ∧
```



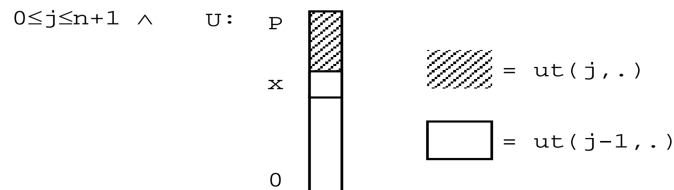

```
                                                        }
```

```
    do j≠n+1    →   U2[x]:= max(U1[x],U1[x-pj]+uj)
                    if  x=P  →   j,x:= j+1,pj+1;
                                 U1:= U2
                    []  x≠P  →   x:= x+1
                    fi
    od
    {Pos:  U1[P] = ut(n,P)}
]
```

Así:

```
T(n,P) = O(nP)
S(n,P) = 2P
```

El ahorro en espacio es significativo. La situación puede, incluso, mejorar hasta lograr que se deba almacenar sólo una columna, si se piensa que se puede mantener un invariante que afirme:



El nuevo algoritmo puede escribirse de la siguiente forma:

```
[Ctx: U[0..P]
    U:= 0;
    j,x:= 1,P;
    {Inv:  0≤j≤n+1 ∧ pj≤x≤P ∧ U[0..x]=ut(j-1,.) ∧ U[x+1..P]=ut(j,.)
    do j≤n+1    →   if U[x]<U[x-pj]+uj then U[x]:= U[x-pj]+uj fi;
                    if x>pj   then x:= x-1
                              else j,x:= j+1,P
                    fi
    od
    {Pos:  U[P] = ut(n,P)}
]
```

Hasta aquí, se ha solucionado el problema de conocer cuál sería la utilidad máxima posible. En la práctica se quiere averiguar, además, en qué forma se puede alcanzar este óptimo.

Ahora, *¿cómo saber qué objetos se llevan?*

La técnica se extiende para considerar una función adicional que permita recordar las decisiones que se tomaron en el cálculo del óptimo. Los valores de esta función deben almacenarse para construir la respuesta a la nueva pregunta.

Para este caso:

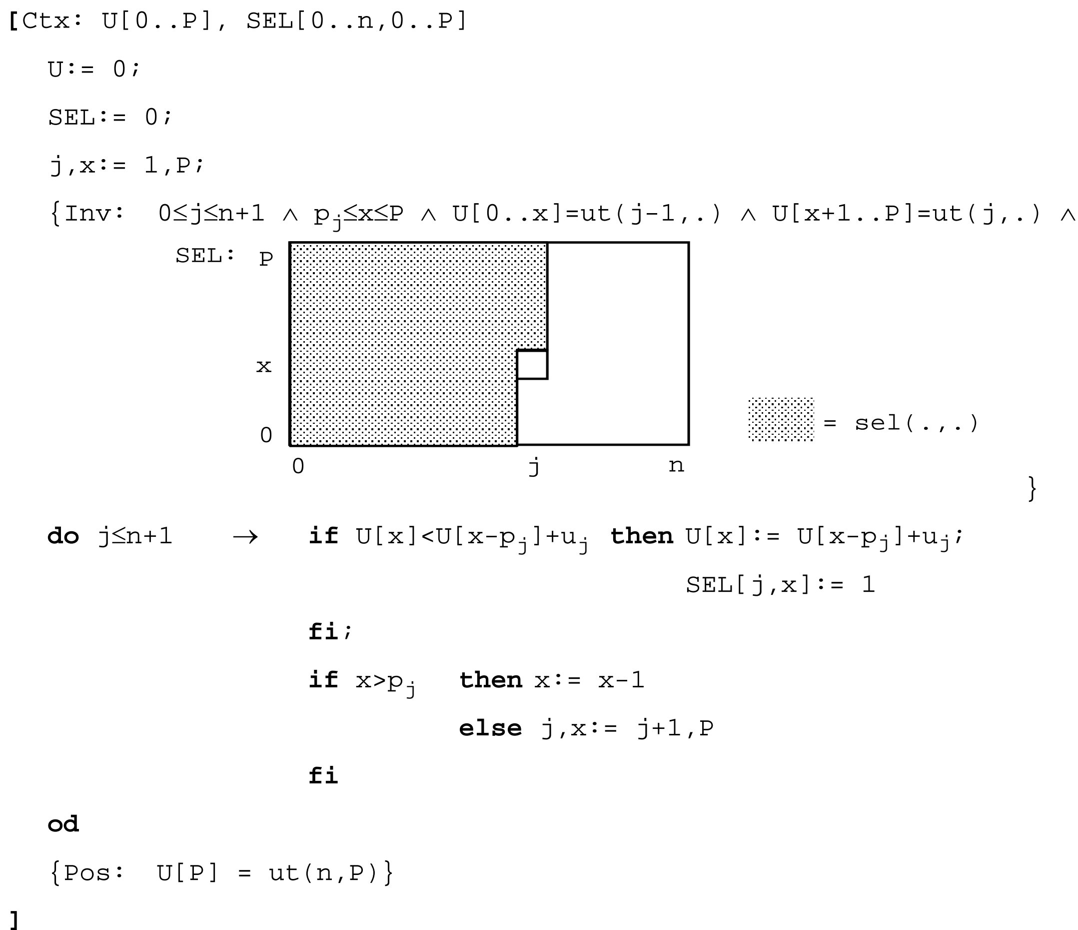
`sel(j,x)` : `1` si o_j se lleva, cuando el peso máximo permitido es `x`

: `0` si o_j no se lleva, cuando el peso máximo permitido es `x`.

Nótese que[2]:

$$sel(j,x) = 1 \quad \Leftrightarrow \quad 0<j\leq P \wedge x\geq p_j \textbf{ cand } ut(j-1,x) < ut(j-1,x-p_j)+u_j$$

Con esta definición, el algoritmo puede modificarse para calcular `sel`:



```
[Ctx: U[0..P], SEL[0..n,0..P]
   U:= 0;
   SEL:= 0;
   j,x:= 1,P;
   {Inv:  0≤j≤n+1 ∧ pj≤x≤P ∧ U[0..x]=ut(j-1,.) ∧ U[x+1..P]=ut(j,.) ∧
           SEL: (diagrama: P, x, 0 / 0, j, n;  ░ = sel(.,.) )
                                                                  }
   do j≤n+1   →   if U[x]<U[x-pj]+uj then U[x]:= U[x-pj]+uj;
                                          SEL[j,x]:= 1
                  fi;
                  if x>pj   then x:= x-1
                            else j,x:= j+1,P
                  fi
   od
   {Pos:  U[P] = ut(n,P)}
]
```

Obsérvese que la estructura del algoritmo que calcula el óptimo se respeta y se utiliza como plataforma sobre la que se puede, adicionalmente, escribir código que permita recordar las decisiones tomadas.

Finalmente, para determinar qué objetos se pueden llevar para conseguir la utilidad máxima, defínase, para $1\leq i\leq n$:

---

[2] Si se tiene igualdad en la comparación, daría lo mismo llevar a o_j o no. Aquí se supone que no se lleva.

```
x[i] =1 ⇔ "oi se lleva, para alcanzar el óptimo"
```

El siguiente algoritmo, que calcula en el arreglo `X[1..n]` el vector `x[1..n]`:

```
[Ctx: X[1..n]
   {Pre:  SEL = sel}
   j, CapRes:= n,P;
   {Inv:  0≤j≤n ∧ X[j+1..n]=x[j+1..n] ∧  CaPRes = P-⟨+: j<i≤n: piX[i]⟩ }
   do j≠0 →    X[j]:= SEL[j,CapRes];
               if   X[j]=1    then CapRes:= CapRes-pj fi;
               j:= j-1
   od
  {Pos:     "X[1..n] ya calculado"
  ]
```

4.2 EL PROBLEMA DEL ANÁLISIS SINTÁCTICO

Dados un alfabeto V, una palabra α y una gramática G, el *problema del análisis sintáctico* consiste en decidir si la palabra pertenece al lenguaje definido por la gramática L(G).

Más generalmente, se trata de determinar $A = \{a \in V \mid a \Rightarrow^* \alpha\}$.

Para el planteo actual, se supone que la gramática está en forma normal binaria, i.e., toda producción es de la forma

$$a \rightarrow bc$$

para ciertos $a,b,c \in V$.

El lenguaje
Supóngase calculable una función

$$letras: V \times V \rightarrow 2^V$$

tal que:

$$letras(b,c) = \{a \in V \mid \text{Existe una producción } a \rightarrow bc \text{ en } G\}$$

Esta función se puede considerar extendida a subconjuntos de parejas de letras, i.e.,

$$letras(S) = \langle \cup : (b,c) \in S : letras(b,c) \rangle$$

La palabra α es tal que:

$$\alpha = a_1 \ldots a_n.$$

Si para $1 \le i \le j \le n$, se denota con $lg(i,j)$ el conjunto de las letras que generan la subpalabra $a_i \ldots a_j$, se tiene que:

$$lg(i,j) = \{v \in V \mid v \Rightarrow^* a_i \ldots a_j\} \quad , \ 1 \le i \le j \le n$$

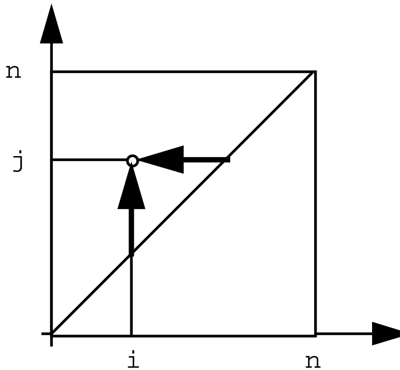
Entonces, la respuesta buscada es `lg(1,n)`.

La recurrencia

Ahora se puede aprovechar el hecho de que la gramática sea binaria:

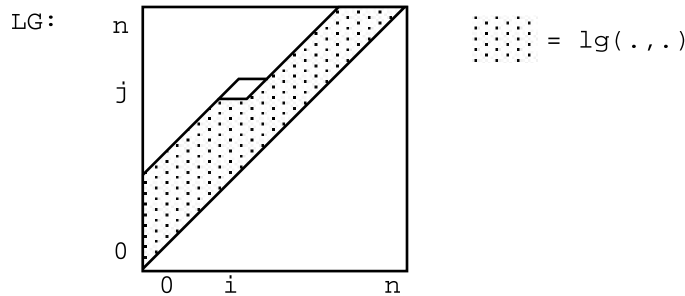
```
lg(i,j)  =   {aᵢ}                                          , si 1≤i=j≤n
         =   ⟨∪: i≤k<j : letras(lg(i,k) x lg(k+1,j))⟩      , si 1≤i<j≤n
```

Diagrama de necesidades



El algoritmo (de Younger):

```
[Ctx: LG[1..n,1..n]
   for k:=1 to n → LG[k,k]:= {aₖ};
   i,j:= 1,2;
   {inv:
```




```
                                                   }
   do j-i≠n   →   LG[i,j]:= ∅;
                  for k:= i to j-1 →
                       LG[i,j]:= LG[i,j] ∪ letras(LG[i,k]xLG[k+1,j])
                  rof;
                  if   j=n  →   i,j:= 1,j-i+2
                  []   j≠n  →   i,j:= i+1,j+1
                  fi
```

```
od
{Pos:     LG[1,n] = lg(1,n) }
]
```

De este modo (suponiendo constante el cálculo de `letras`):

$T(n) = O(n^2)\ O(n) = O(n^3)$

$S(n) = O(n^2)$

4.3 EJERCICIOS

1 Estime la complejidad del algoritmo de Younger si se consideran como variables de entrada la gramática binaria G y una palabra de longitud n. (En otros términos: cuánto cuestan, dependiendo de G, los cálculos de la función `letras`?)

2 El algoritmo de Younger descrito en el texto tiene la desventaja de exigir tener toda la palabra α disponible para empezar a generar los valores de `lg`. Modifique este algoritmo para diseñar un analizador sintáctico con programación dinámica que tome $O(n^3)$ operaciones básicas y genere todos los cálculos necesarios para la palabra $a_1 \dots a_i$ antes de generar los cálculos para la palabra $a_1 \dots a_{i+1}$.

3 Encuentre el subarreglo ascendente más largo contenido en un arreglo dado de longitud n.

4 Encuentre la longitud de la subsucesión ascendente más larga que se puede encontrar en un arreglo dado de longitud n. Indique cómo modoficar el algoritmo para encontrar una subsucesión de longitud máxima.

5 Suponga disponible un algoritmo para multiplicar una matriz de dimensiones $p \times q$ por otra de dimensiones $q \times r$ con pqr multiplicaciones elementales. Dado un conjunto de n matrices numéricas $A_1, A_2, \dots, A_n$, tal que, para $i=1,\dots n$, A_i tiene dimensiones $d_{i-1} \times d_i$, encuentre la forma más eficiente de asociar el producto $A_1A_2 \dots A_n$ de modo que, utilizando el algoritmo disponible para multiplicar pares de matrices, el número de multiplicaciones elementales sea mínimo.

6 Sea $S = \{s_1, \dots, s_n\}$ un conjunto de n números naturales. Diseñe un algoritmo que utilice programación dinámica par determinar si S se puede partir en dos conjuntos cuyos elementos sumen la misma cantidad.

7 Un conjunto $C = \{a_1, \dots, a_n\}$, ordenado totalmente, puede representarse con un árbol binario con n nodos, en el que para cada nodo (representante de un elemento del conjunto) los nodos del subárbol izquierdo (derecho) son menores (mayores) que él. Se dice que un tal árbol binario está *ordenado*. Sea $f\colon C \to]0,1]$ una distribución de frecuencias (probabilidades) para los elementos de C.
Para $A \subseteq C$, sea $f(.|A)\colon A \to]0,1]$ la distribución de frecuencias condicionales de A, i.e.,

$$f(x|A) = \frac{f(x)}{f(A)}$$

donde $f(A) = \langle +:\ a \in A\ :\ f(a) \rangle$. Si B es un árbol binario ordenado que representa el conjunto A, el costo (medio) de la consulta de un nodo de B se define como

$$cc_B = <+:\ x \in A\ :\ h_B(x) f(x|A)>$$

en donde $h_B(x)$ es la altura del nodo que representa a x en B. Para un árbol vacío este costo es 0. Esta forma de medir las consultas enfatiza el hecho de que, si los nodos más consultados están más cerca de la raíz, el costo será menor. Entonces, tiene sentido hablar de árboles de consulta óptima en cuanto a que de un árbol a otro, aunque representen el mismo conjunto, puede haber diferencias de costos de consulta.
Diseñe un algoritmo que calcule un árbol de consulta óptima que represente el conjunto C.

8 Diseñe un algoritmo polinomial para expresar una suma entera de dinero con monedas cuyo peso total sea mínimo. Suponga que se cuenta con cantidades ilimitadas de monedas de denominaciones $d_1, d_2, \ldots, d_n$, enumeradas en orden ascendente y con $d_1 = 1$, con pesos para cada denominación respectivamente $p_1, p_2, \ldots, p_n$. (no se supone una relación particular de orden entre estos pesos).

9 Usando técnicas de programación dinámica, dé una versión iterativa para un algoritmo de exponenciación entera b^n, de complejidad $O(\log n)$ en el peor caso.

10 Sean B_1 y B_2 dos bodegas de donde se envían mercancías a los destinos D_j, con $1 \le j \le n$. Sea d_j la demanda de D_j y r_i el inventario en B_i $(i=1,2)$.
Suponga $r_1 + r_2 = \langle +:\ 1 \le j \le n\ :\ d_j \rangle$. Sea $C_{ij}(x_{ij})$ el costo de enviar x_{ij} unidades de B_i a D_j. El problema de las bodegas consiste en encontrar enteros no negativos x_{ij} con $i=1,2$ y $1 \le j \le n$, tales que $x_{1j} + x_{2j} = d_j$ con $1 \le j \le n$, de modo que

$$CT = \langle +:\ 1 \le i \le 2,\ 1 \le j \le n:\ C_{ij}(x_{ij}) \rangle$$

sea mínimo. Diseñe un algoritmo que resuelva este problema y calcule sus complejidades temporal y espacial.

11 Dado un grafo dirigido $G=(V,E,c)$ (con $V=\{1,2,_,n\}$), etiquetado con costos positivos en los arcos mediante una función $c:\ E \to R^+$, representado mediante una matriz de costos (c'_{ij}), $1 \le i,j \le n$, tal que

$$
\begin{aligned}
c'_{ij} &= 0 &&, \text{ si } i=j \\
&= c(i,j) &&, \text{ si } i \ne j,\ (i,j) \in E, \\
&= \infty &&, \text{ si } i \ne j,\ (i,j) \notin E
\end{aligned}
$$

Considere el problema de calcular, para cada vértice $v \in V$, una ruta de costo óptimo que conecta al nodo 1 al nodo v. Se desea una solución obtenida mediante el uso de métodos de programación dinámica. Para ello, defina:

$Op(v)$ ≈ "costo óptimo en G de una ruta de 1 a v ($v \in V$)"

$Opt(k,v)$ ≈ "costo óptimo en G para rutas de a lo sumo k arcos que vayan de 1 a v".