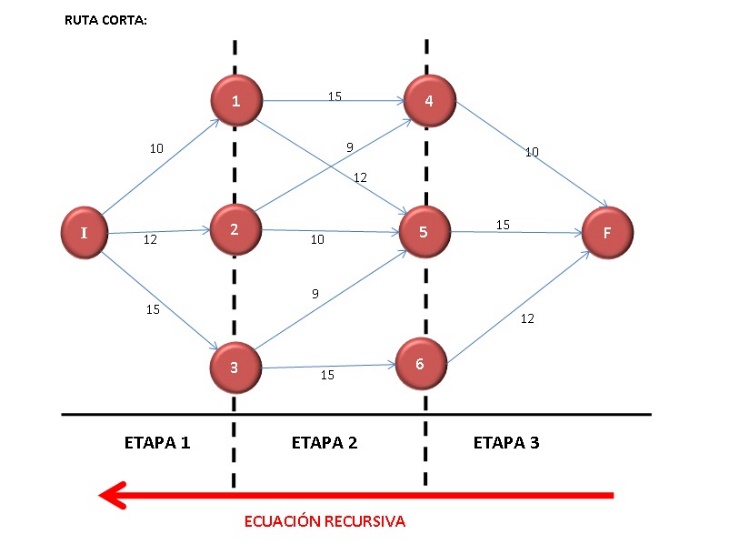
# **ALGORITMOS DE PROGRAMACIÓN DINÁMICA**

# Typesetting

# Andrés Cabero Mata

## Introducción

La programación dinámica se utiliza en la resolución de problemas que requieren tomar decisiones en etapas sucesivas. Las decisiones tomadas en cada etapa del problema condicionan la decisión de la siguiente etapa. El método consiste en dividir el problema principal en sub-problemas más básicos, cuya solución dará la información necesaria para la llegar a la solución del siguiente sub-problema más grande, hasta obtener la solución del problema original. Es una técnica ascendente, partiendo de los problemas menores hasta el problema propuesto.

Es una mejora muy importante desde el punto de vista de la eficiencia en muchos problemas, en concreto aquellos con solución recursiva (Figura 1), pues evita tener que hacer los mismos cálculos varias veces, bajando el coste temporal de algoritmos de coste exponencial a, incluso, coste lineal.

Figura 1

### Problema del Typesetting

El problema del Typesetting o del ajuste de líneas consiste en, dado un párrafo de un texto y un margen límite, encontrar la mejor disposición de palabras en cada línea de texto tal que se minimicen los espacios en blanco finales, sin pasarse en ningún momento del margen límite. Esta disposición de palabras es lo mismo que el conjunto de líneas que hacen que los espacios finales sean mínimos, es decir, de que palabra a que palabra es cada línea del párrafo calculado. El coste de poner esa sucesión de líneas sería la suma de los espacios finales de cada línea, pero lo más común es utilizar algún tipo de penalización, pues la solución dada suele ser de mejor calidad visual, pues cuantos más espacios en cada línea mayor será el coste de poner esa línea.

Es lógico pensar que una solución con un algoritmo voraz debería ser capaz de encontrar esta sucesión de líneas, buscando el mínimo de espacios finales de cada línea. Y aunque da una solución cercana a la óptima:

Texto: “aaa bb cc dddd”.

Salida: aaa bb

cc

ddddd

Margen: 6 (caracteres).

Los espacios finales son 0, 4 y 1. Aplicando a estos una penalización cuadrática el coste de cada línea es 0, 16 y 1 respectivamente, dando un coste total del texto de 17. Aun siendo una solución muy cercana, la óptima es esta:

En esta solución los espacios finales son 3, 1 y 1 en cada línea, aplicando la misma penalización que en el caso anterior da un coste por línea de 9, 1 y 1 respectivamente. El coste total de poner el texto es por tanto de 11, claramente menor.

Salida: aaa

bb cc

ddddd

Aun así es importante decir que debido a la cercanía del algoritmo voraz a la solución óptima, y su mayor rapidez, muchos programas de edición de texto como OppenOffice o Microsoft Word utilizan la solución voraz.

## Solución dinámica del Typesetting

A la hora de plantear la solución del problema del Typesetting por medio de la programación dinámica, es importante encontrar en que sub-problemas se puede dividir. Para ello primero definimos claramente el problema:

Hay que encontrar la sucesión de líneas que conforman un texto en un margen definido, cuyo coste total sea mínimo.

Coste = =Caracteres en cada línea = Margen

Figura 2

La entrada del algoritmo han de ser las palabras que forman al texto, o las longitudes de estas y el margen máximo para las líneas.

Visto esto, una forma de poder plantear el problema como un conjunto de sub-problemas menores podría ser como escribir desde la palabra 1 a la palabra j con menor coste, siendo j una palabra del texto. De esta manera el primer sub-problema a resolver sería escribir la línea que va a la palabra 1 a la 1; escribir la palabra 1 en una línea. Para la línea (1, 2) no hay una única opción, pues podría escribirse la línea (1,1) y (2,2), o únicamente la (1,2). Estas opciones van aumentando por cada palabra incluyas en el sub-problema.

Parece necesario conocer el coste de cada línea posible del texto, incluyendo de esta forma cuales de ellas no se pueden colocar por contener más caracteres que el margen. Necesitamos calcular la tabla de costes: tc(i, j). En esta tabla, tc(i, j) será el coste de poner la línea (i, j). La entrada necesaria para calcular esta tabla son las longitudes de cada palabra y el margen límite, contados ambos como el número de caracteres.

Para construir esta tabla hay que saber primero la longitud de cada línea (i, j) ya que su coste utiliza su número de caracteres (Figura 2), por tanto, es lo primero a calcular. Para saber cuál es la longitud o número de caracteres de cada línea (i, j) es fácil observar que va a ser la suma de j más la línea (i, j-1), además del espacio entre j-1 y j, el cual será un carácter a mayores (Figura 3). Esto es así excepto para la longitud de la línea formada por solo una palabra, que es la longitud de esa palabra (Figura 4). La tabla de dos dimensiones que contenga estas longitudes será la misma que se utilizará para aplicar la fórmula del coste y calcular así el coste de cada posible línea:

Calculo de las longitudes:

Figura 4

Figura 3

Queda calcular los casos no posibles incluidos en la tabla de dos dimensiones, que son en los que j es menor que i, los cuales se evaluaran como coste infinito.

Figura 5

Para optimizar el cálculo de las longitudes lo mejor es parar de calcularlos cuando

Figura 5

evaluando los (i, j) restantes de la fila (i) de la tabla con valor infinito.

Visto esto, lo que parece más sencillo es calcular la tabla de longitudes por filas, siendo una fila los cambios en la última palabra. Cada fila se define por en que palabra se inicia las líneas de la fila, así la primera fila son las líneas que empiezan por la primera palabra, la segunda fila por la segunda palabra, y así en adelante.

La tabla de longitudes para el ejemplo de “aaa bb cc ddddd” y margen 6 sería:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **aaa** | **aaa bb** |  |  |  | **3** | **6** | Inf | Inf |
|  | **bb** | **bb cc** |  |  | Inf | **2** | **5** | Inf |
|  |  | **cc** |  |  | inf | Inf | **2** | inf |
|  |  |  | **ddddd** |  | Inf | Inf | inf | **5** |

Ya calculada la tabla de las longitudes de cada línea, calcular el coste es tan simple como utilizar hacer el cálculo del coste (Figura 2) para cada línea y sustituirlo en la misma tabla. Para el mismo ejemplo:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **3** | **6** | Inf | Inf |  | **9** | **0** | Inf | Inf |
| Inf | **2** | **5** | Inf |  | Inf | **16** | **1** | Inf |
| inf | Inf | **2** | inf |  | inf | Inf | **16** | inf |
| Inf | Inf | inf | **5** |  | Inf | Inf | inf | **1** |

Con la tabla de costes ya calculada, el problema original sigue sin resolverse, pues necesitamos encontrar la sucesión de líneas que den el menor coste, pero gracias a la tabla, es mucho más sencillo ver cuál es la solución. Si miramos la forma recursiva de calcular el menor coste, utilizando la tabla de costes:

Figura 6

En este método recursivo (Figura 6), donde c(j) es el coste mínimo de poner las palabras de la 1 a la j, hay superposición de sub-problemas; por ejemplo, la solución del sub-problema c(2) es usada en el c(3) y c(4). Aquí entra la programación dinámica, la cual guarda los resultados de los sub-problemas, y los calcula empezando por c(1).

A la vez que guarda el coste menor de poner hasta la palabra j, guarda también la palabra desde la que escribe (i, j); i, formando la lista r(j).

La lista c(i) y r(j) para el ejemplo “aaa bb cc ddddd” y margen 6, serían respectivamente:

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 0 | 10 | 11 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 2 | 4 |

Esta lista r(j) indica para cada j cual es la línea que llega a la palabra j que está incluida en la sucesión de líneas que hacen el coste mínimo. Si se recorre la lista empezando r(j), siendo j la última palabra y por tanto el último elemento de la lista r, se puede obtener la información (r(j), j), que es la última línea del texto. Si seguimos buscando por la anterior última línea, es decir, la anterior a la ya encontrada; r(r(j)-1), hasta llegar a la primera palabra, se puede formar la salida deseada del algoritmo: Tras que palabra hay que hacer saltos de línea para construir el texto con las líneas que hacen el menor coste posible. A la hora de encontrar los saltos de línea se encuentran por orden contrario al texto, así que es necesario dar la vuelta a la lista donde se almacenan.

Para el mismo ejemplo estos saltos de línea son los siguientes:

|  |  |  |
| --- | --- | --- |
| 1 | 3 | 4 |

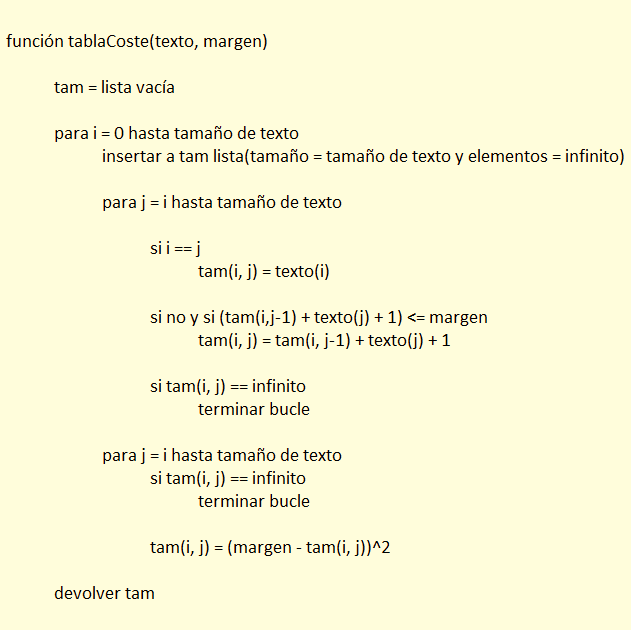
Y, como ya hemos visto estos saltos de línea dan forma al texto de la siguiente manera:

Salida: aaa (\n)

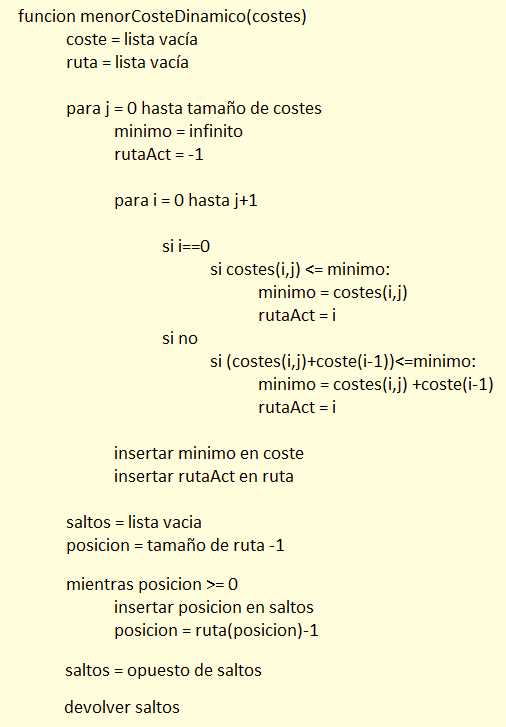
bb cc (\n)

ddddd (\n)

## Pseudocódigo

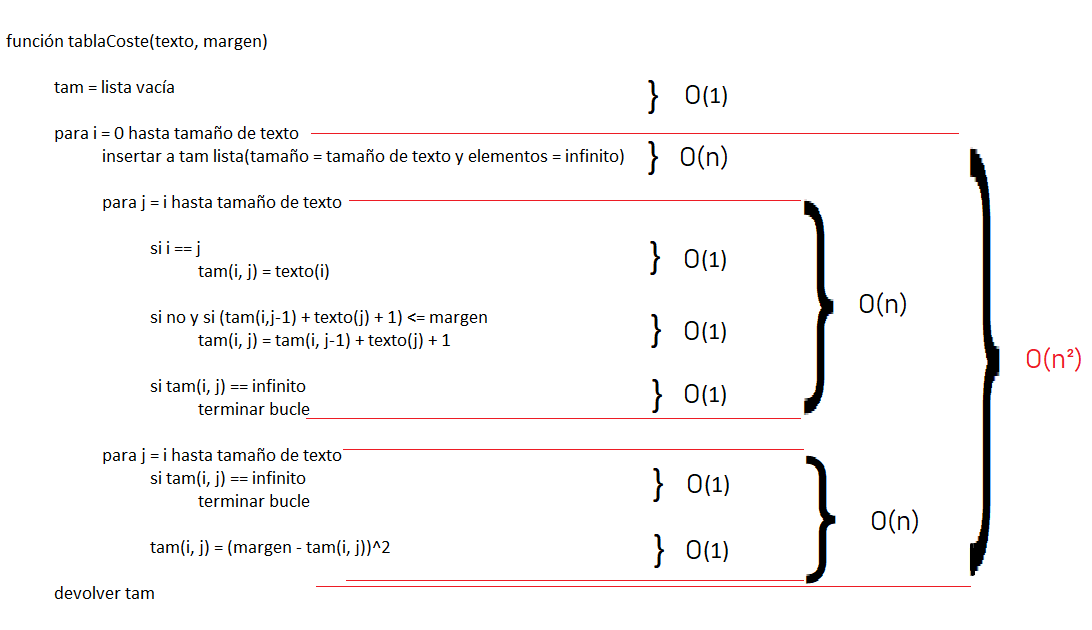
Construcción de la tabla de costes:

Búsqueda de disposición del texto con coste mínimo:

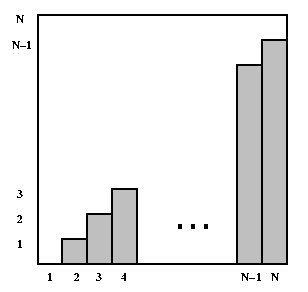


## Análisis de coste

### 4.1. Construcción de la tabla de costes

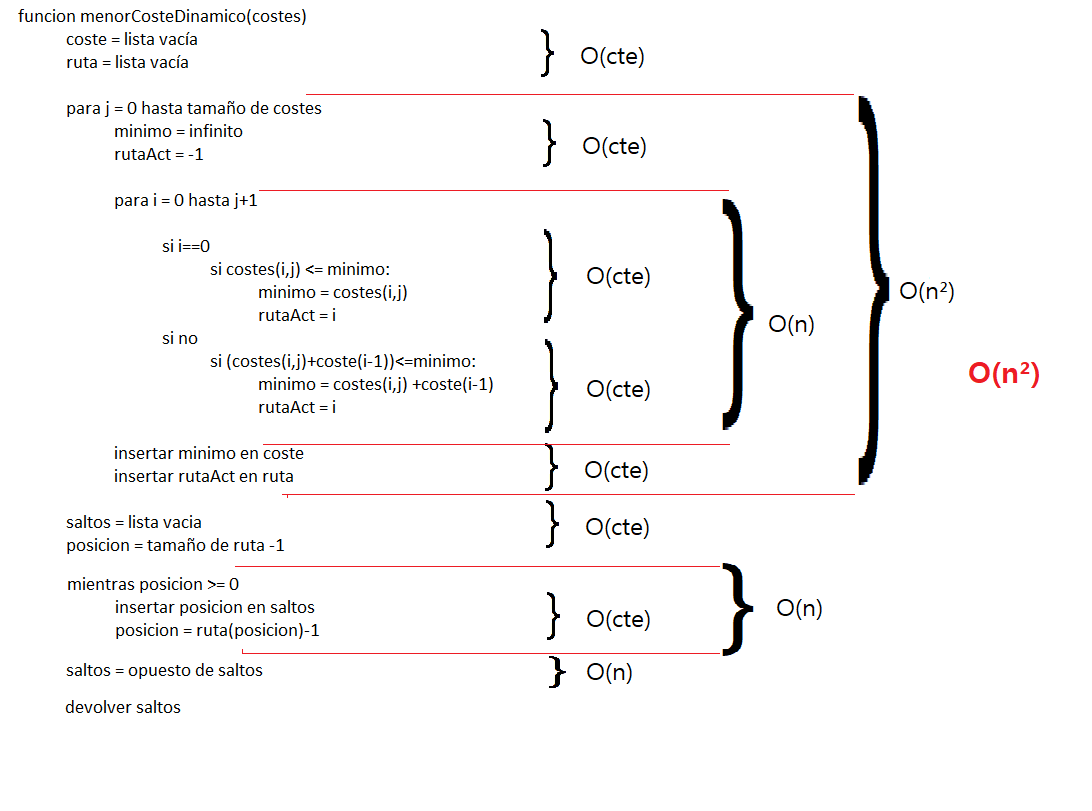
El coste total de construir la tabla de coste es del orden cuadrático.

Con la implementación mostrada en el pseudo código se hacen un bucle principal el cual se hace tantas veces como palabras haya en el texto.

Dentro de este, para cada palabra utilizada como inicio de línea se calculan las longitudes de toda la fila que tiene como inicio de línea la misma palabra en un bucle y tras ello el coste de esa línea en otro. En el peor caso, el caso en el que el margen sea tan grande como para que todas las posibles líneas necesiten ser calculadas, el primer bucle que calcula las longitudes lo hará N+N-1+N-2…+N-(N-1) veces cuyo valor es equivalente a N2/2

De la misma manera lo hace el segundo bucle, por tanto, la suma de ambos da un valor cercano a N2. Esto hace un orden cuadrático a la construcción de la tabla.

### Búsqueda de la disposición del texto con coste mínimo



El coste total de encontrar los saltos de línea que dan la mejor disposición al texto es de orden cuadrático.

El algoritmo se compone de dos bucles, el primero que encuentra los costes mínimos de escribir todas las palabras y la procedencia de esas palabras, y el segundo, que buscando en la procedencia de las palabras encuentra el conjunto de líneas y por tanto los saltos de línea necesarios. Este primer bucle contiene otro que tiene la misma dependencia que los bucles vistos en la construcción de la tabla de costes, haciéndose 1+2+3+…+(N-1) veces, lo cual hace que el primer bucle tenga un coste de orden cuadrático.

El segundo bucle, el que se encarga de crear la lista de saltos a través de la información del primer bucle, tiene un peor caso muy concreto, en el que cada palabra del texto entra en una línea. Esto hace que a la hora de encontrar los saltos de línea sea uno tras cada palabra, por tanto, teniendo en cuenta que lo que se encuentra dentro del bucle es de orden constante, el coste del bucle es de orden lineal.

### Coste espacial

El coste espacial de todo el algoritmo es el coste de la tabla de longitudes y costes, que es la misma, junto con el coste de las listas necesarias en el segundo algoritmo, que son 2 listas de tantos elementos como palabras tiene el texto. El coste es la suma de las tres, y teniendo en cuenta que la que mayor coste espacial conlleva es la primera, siendo una tabla de orden cuadrático, el coste del algoritmo es por tanto cuadrático.

## Bibliografía

* <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>
* <https://www.geeksforgeeks.org/word-wrap-problem-space-optimized-solution/>
* <https://en.wikipedia.org/wiki/Line_wrap_and_word_wrap>
* <http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html>
* <https://wiki.python.org/moin/TimeComplexity>
* Apuntes Programación Dinámica, asignatura Algoritmos y Computación, UVA, 18/19
* Introduction to Algorithms 3rd edition, Thomas H. Cormen