

# Lecture 18: Applications of Dynamic Programming

**Steven Skiena**

Department of Computer Science  
State University of New York  
Stony Brook, NY 11794-4400

<http://www.cs.sunysb.edu/~skiena>

## Problem of the Day

---

A certain string processing language allows the programmer to break a string into two pieces. Since this involves copying the old string, it costs  $n$  units of time to break a string of  $n$  characters into two pieces.

Suppose a programmer wants to break a string into many pieces. The order in which the breaks are made can affect the total amount of time used.

For example suppose we wish to break a 20 character string after characters 3,8, and 10:

- If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 17 units of time and the third break costs 12 units of time, a

total of 49 steps.

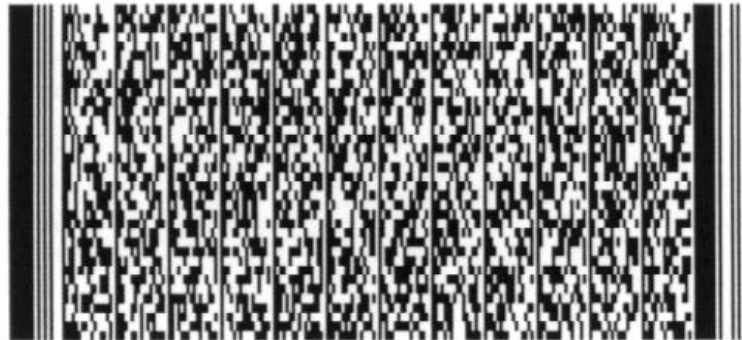
- If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, a total of only 38 steps.

Give a dynamic programming algorithm that, given the list of character positions after which to break, determines the cheapest break cost in  $O(n^3)$  time.

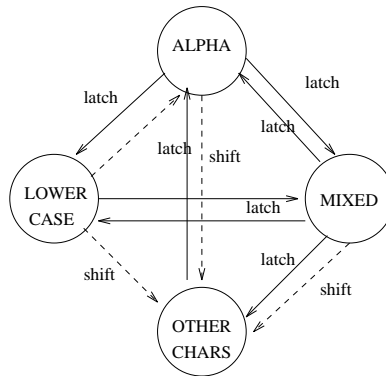
# Dynamic Programming and High Density Bar Codes

---

Symbol Technology has developed a new design for bar codes, PDF-417 that has a capacity of several hundred bytes. What is the best way to encode text for this design?



They developed a complicated mode-switching data compression scheme.



Latch commands permanently put you in a different mode. Shift commands temporarily put you in a different mode. Symbol used a greedy algorithm to encode a string, making local decisions only. But we realized that for any prefix, you want an optimal encoding which might leave you in every possible mode.

## The Quick Brown Fox

Alpha		
Lower		
Mixed	X	
Punct.		

$M[i, j] = \min(M[i - 1, k] + \text{the cost of encoding the } i\text{th character and ending up in node } j).$

Our simple dynamic programming algorithm improved the capacity of PDF-417 by an average of 8%!

## Dividing the Work

---

Suppose the job scanning through a shelf of books is to be split between  $k$  workers. To avoid the need to rearrange the books or separate them into piles, we can divide the shelf into  $k$  regions and assign each region to one worker.

What is the fairest way to divide the shelf up?

If each book is the same length, partition the books into equal-sized regions,

100 100 100 | 100 100 100 | 100 100 100

But what if the books are not the same length? This partition would yield

100 200 300 | 400 500 600 | 700 800 900

Which part of the job would you volunteer to do?  
How can we find the fairest possible partition, i.e.

100 200 300 400 500 | 600 700 | 800 900



# The Linear Partition Problem

---

Input: A given arrangement  $S$  of nonnegative numbers  $\{s_1, \dots, s_n\}$  and an integer  $k$ .

Problem: Partition  $S$  into  $k$  ranges, so as to minimize the maximum sum over all the ranges.

Does fixed partition positions always work?

Does taking the average value of a part  $\sum_{i=1}^n s_i/k$  from the left always work?

## Recursive Idea

---

Consider a recursive, exhaustive search approach. Notice that the  $k$ th partition starts right after we placed the  $(k - 1)$ st divider.

Where can we place this last divider? Between the  $i$ th and  $(i + 1)$ st elements for some  $i$ , where  $1 \leq i \leq n$ .

What is the cost of this? The total cost will be the larger of two quantities, (1) the cost of the last partition  $\sum_{j=i+1}^n s_j$  and (2) the cost of the largest partition cost formed to the left of  $i$ .

What is the size of this left partition? To partition the elements  $\{s_1, \dots, s_i\}$  as equally as possible. *But this is a smaller instance of the same problem!*

# Dynamic Programming Recurrence

---

Define  $M[n, k]$  to be the minimum possible cost over all partitionings of  $\{s_1, \dots, s_n\}$  into  $k$  ranges, where the cost of a partition is the largest sum of elements in one of its parts. Thus defined, this function can be evaluated:

$$M[n, k] = \min_{i=1}^n \max(M[i, k-1], \sum_{j=i+1}^n s_j)$$

with the natural basis cases of

$$M[1, k] = s_1, \text{ for all } k > 0 \text{ and,}$$

$$M[n, 1] = \sum_{i=1}^n s_i$$

What is the running time?

It is the number of cells times the running time per cell.

A total of  $k \cdot n$  cells exist in the table.

Each cell depends on  $n$  others, and can be computed in linear time, for a total of  $O(kn^2)$ .

# Implementation

---

To evaluate this efficiently, we must make sure we do the smaller cases before the larger cases that depend upon them.

Partition[ $S, k$ ]

(\* compute prefix sums:  $p[k] = \sum_{i=1}^k s_i$  \*)

$p[0] = 0$

for  $i = 1$  to  $n$  do  $p[i] = p[i - 1] + s_i$

(\* initialize boundary conditions \*)

for  $i = 1$  to  $n$  do  $M[i, 1] = p[i]$

for  $i = 1$  to  $k$  do  $M[1, j] = s_1$

(\* evaluate main recurrence \*)

```
for  $i = 2$  to  $n$  do
  for  $j = 2$  to  $k$  do
     $M[i, j] = \infty$ 
    for  $x = 1$  to  $i - 1$  do
       $s = \max(M[x, j - 1], p[i] - p[x])$ 
      if ( $M[i, j] > s$ ) then
         $M[i, j] = s$ 
         $D[i, j] = x$ 
```

# DP Matrices

---

For the input  $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$M$	$k$				$D$	$k$		
$n$	1	2	3		$n$	1	2	3
1	1	1	1		1	-	-	-
2	3	2	2		2	-	1	1
3	6	3	3		3	-	2	2
4	10	6	4		4	-	3	3
5	15	9	6		5	-	3	4
6	21	11	9		6	-	4	5
7	28	15	11		7	-	5	6
8	36	21	15		8	-	5	6
9	45	24	17		9	-	6	7

# DP Matrix: Uniform Series

---

For the input  $\{1, 1, 1, 1, 1, 1, 1, 1, 1\}$

$M$	$k$				$D$	$k$		
$n$	1	2	3		$n$	1	2	3
1	1	1	1		1	-	-	-
1	2	1	1		1	-	1	1
1	3	2	1		1	-	1	2
1	4	2	2		1	-	2	2
1	5	3	2		1	-	2	3
1	6	3	2		1	-	3	4
1	7	4	3		1	-	3	4
1	8	4	3		1	-	4	5
1	9	5	3		1	-	4	6



# When can you use Dynamic Programming?

---

Dynamic programming computes recurrences efficiently by storing partial results. Thus dynamic programming can only be efficient when there are not too many partial results to compute!

There are  $n!$  permutations of an  $n$ -element set – we cannot use dynamic programming to store the best solution for each subpermutation. There are  $2^n$  subsets of an  $n$ -element set – we cannot use dynamic programming to store the best solution for each.

However, there are only  $n(n-1)/2$  contiguous substrings of a string, each described by a starting and ending point, so we can use it for string problems.

There are only  $n(n-1)/2$  possible subtrees of a binary search tree, each described by a maximum and minimum key, so we can use it for optimizing binary search trees.

*Dynamic programming works best on objects which are linearly ordered and cannot be rearranged – characters in a string, matrices in a chain, points around the boundary of a polygon, the left-to-right order of leaves in a search tree.*

Whenever your objects are ordered in a left-to-right way, you should smell dynamic programming!

# The Principle of Optimality

---

To use dynamic programming, the problem must observe the *principle of optimality*, that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the first decision.

Combinatorial problems may have this property but may use too much memory/time to be efficient.

## Example: The Traveling Salesman Problem

---

Let  $T(i; j_1, j_2, \dots, j_k)$  be the cost of the optimal tour for  $i$  to 1 that goes thru each of the other cities once

$$T(i; j_1, j_2, \dots, j_k) = \text{Min}_{1 \leq m \leq k} C[i, j_m] + T(j_m; j_1, j_2, \dots, j_k)$$

$$T(i, j) = C(i, j) + C(j, 1)$$

Here there can be any subset of  $j_1, j_2, \dots, j_k$  instead of any subinterval - hence exponential.