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

$$T(i; i_1, j_2, \ldots, j_i) = Min_{1 \le m \le k} C[i, j_m] T(j_m; j_1, j_2, \ldots, j_k)$$

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

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

Still, with other ideas (some type of pruning or best-first search) it can be effective for combinatorial search.

SHOW PICTURE OF PRUNING TREE

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

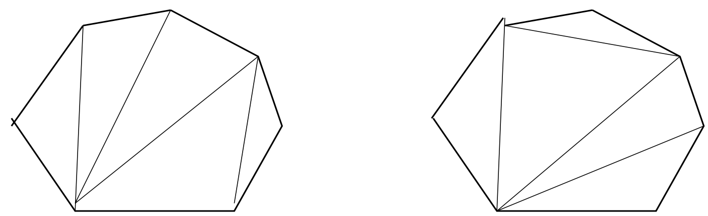
There are only $n(n-1)/2$ possible subtrees of a binary search tree, each described by a maximum and minimum key, so we can use it for optimizing binary search trees.

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!

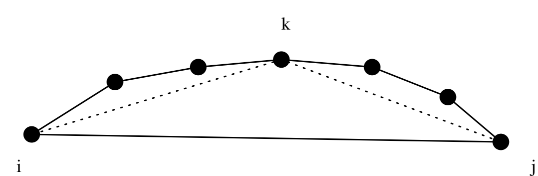
Minimum Length Triangulation

A triangulation of a polygon is a set of non-intersecting diagonals which partiions the polygon into diagonals.



The length of a triangulation is the sum of the diagonal lengths.

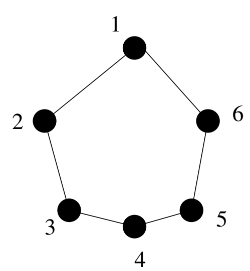
We seek to find the minimum length triangulation. For a convex polygon, or part thereof:



Once we identify the correct connecting vertex, the polygon is partitioned into two smaller pieces, both of which must be triangulated optimally!

$$
\begin{aligned}
t[i, i+1] &= 0 \\
t[i, j] &= \min_{k=i}^{j} t[i, k] + t[k, j] + |ik| + |kj|
\end{aligned}
$$

Evaluation proceeds as in the matrix multiplication example - $\binom{n}{2}$ values of t, each of which takes $O(j-i)$ time if we evaluate the sections in order of increasing size.



J-i = 2

13, 24, 35, 46, 51, 62

J-i = 3

14, 25, 36, 41, 52, 63

J-i = 4

15, 26, 31, 42, 53, 64

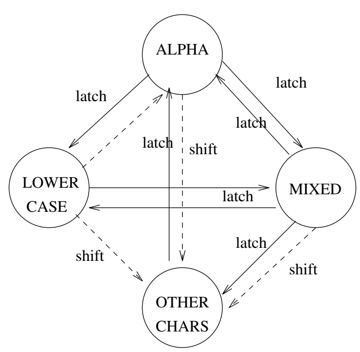
Finish with 16

What if there are points in the interior of the polygon?

Dynamic Programming and High Density Bar Codes

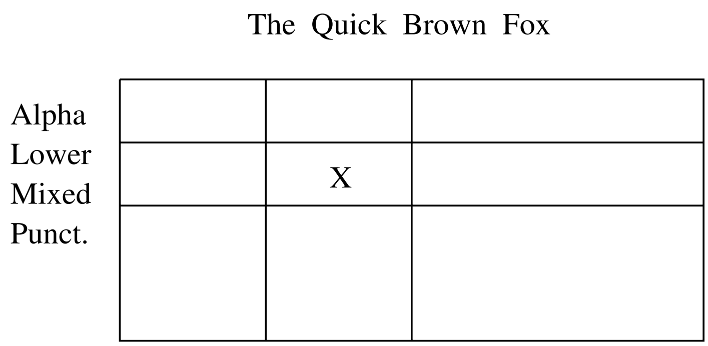
Symbol Technology has developed a new design for bar codes, PDF-417 that has a capacity of several hundred bytes. What is the best way to encode text for this design?

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.

Originally, Symbol used a greedy algorithm to encode a string, making local decisions only. We realized that for any prefix, you want an optimal encoding which might leave you in every possible mode.



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

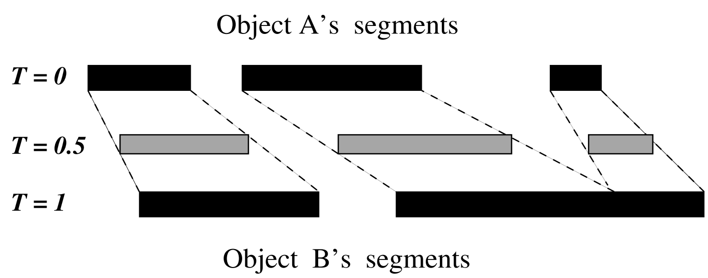
Our simple dynamic programming algorithm improved to capacity of PDF-417 by an average of 8%!

Dynamic Programming and Morphing

Morphing is the problem of creating a smooth series of intermediate images given a starting and ending image.

The key problem is establishing a correspondence between features in the two images. You want to morph an eye to an eye, not an ear to an ear.

We can do this matching on a line-by-line basis:



This should sound like string matching, but with a different set of operations:

- *Full run match:* We may match run i on top to run j on bottom for a cost which is a function of the difference in the lengths of the two runs and their positions.

- *Merging runs:* We may match a string of consecutive runs on top to a run on bottom. The cost will be a function of the number of runs, their relative positions, and lengths.

- *Splitting runs:* We may match a big run on top to a string of consecutive runs on the bottom. This is just the converse of the merge. Again, the cost will be a function of the number of runs, their relative positions, and lengths.

This algorithm was incorporated into a morphing system, with the following results: