

Dynamic Programming

R. Bellman began the systematic study of dynamic programming in 1955.

The word “**programming**” refers to the use of a tabular solution method.

DP typically applies to optimization problems in which a **sub-problems of the same form** often arise.

Key technique: store the solution to each subproblem in case it should reappear.

Development of DP

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute value of optimal solution in bottom-up fashion
4. Construct an optimal solution from the value(s) of the optimal solution.

Comments:

- Steps 1-3 (*Forward*) form the basis of a dynamic-programming solution to a problem.
- Step 4 (*Traceback*) can be omitted if only the **value** of an optimal solution is required.

Development of DP (cont'd)

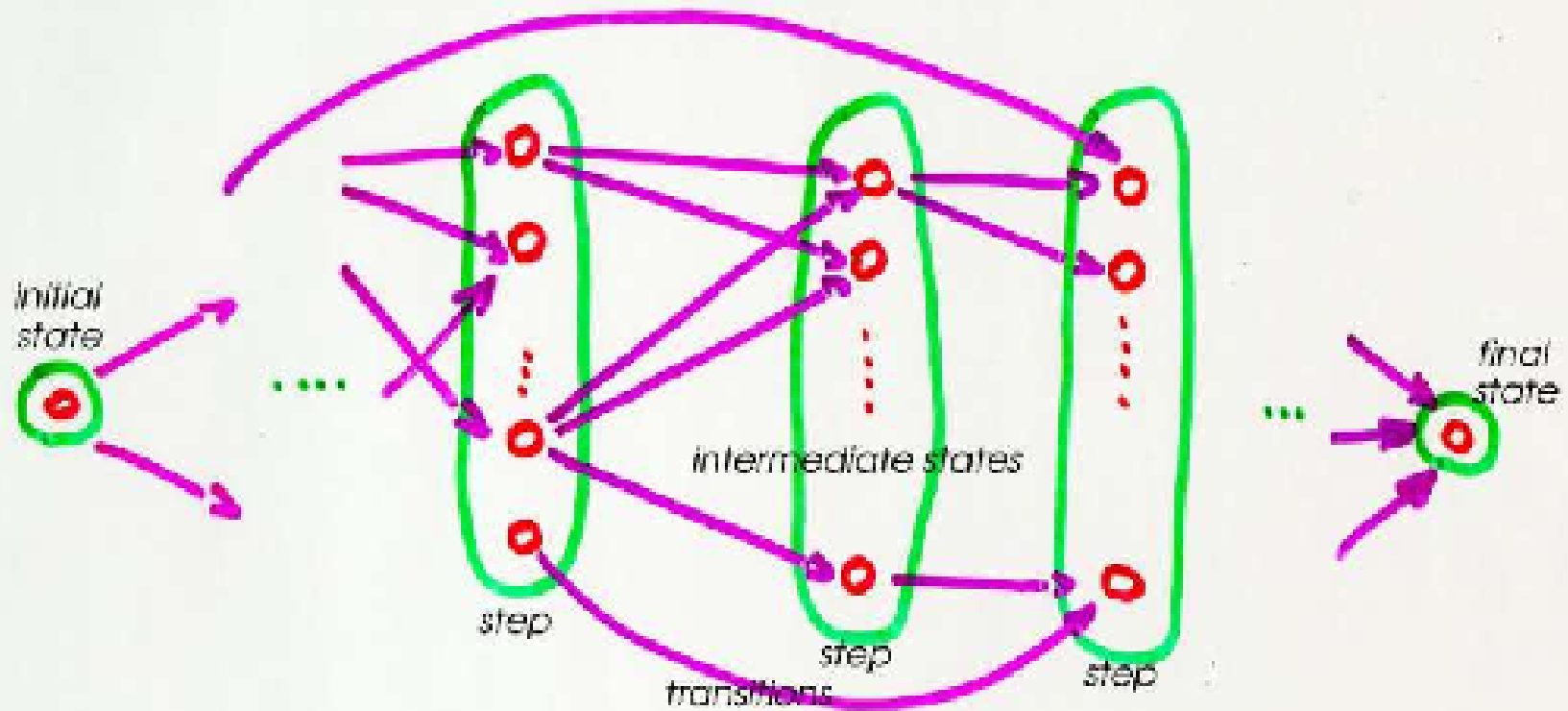
Key ingredients for dynamic programming to be applicable are: **optimal substructure** and **overlapping subproblems**.

Optimal substructure: A problem exhibits optimal substructure if an optimal solution contains optimal solutions to its subproblems.

Overlapping subproblems: Space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over. Typically, the total number of distinct subproblems is a **polynomial in the input size**. When the same subproblem appears over and over again \Rightarrow overlapping subproblems.

The idea of DP in more detail

A view of the computations of dynamic programming



- Compute at each new state the maxima (or minima) of the values **for all** possible transitions **from** previous states.
- If final state is reached, **backtrack** to find the path(s), which lead to the **optimal final state**.
- The set of all transitions **must not have cycles**.
Further assumption: total number of states is polynomial in the size of the problem \Rightarrow computation of each node will require polynomial time.
- Each transition is (possibly) associated with a **cost**. Computation of each state requires the use of **optimal values of previous states** and the **cost of the transitions**.
- Groups of states called **steps** are just used to organize the computation in a sequential way (not needed otherwise).

Examples: Optimal Binary Search Trees

Problem: Construction of **Optimal Binary Search Trees**.

BST : tree where the key values are stored in the internal nodes, the external nodes (leaves) are null nodes, and the keys are ordered lexicographically.

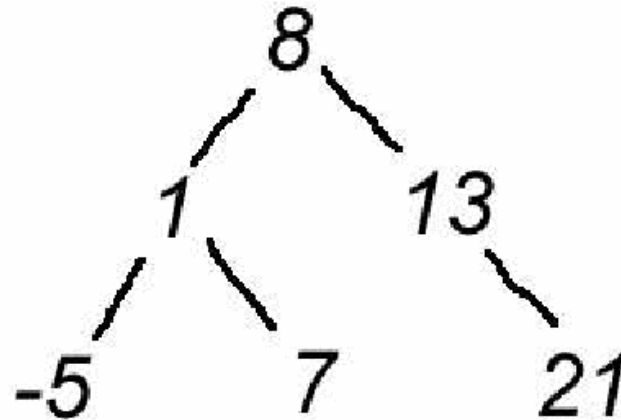
For each internal node all keys in the left subtree are less than the keys in the node, and all the keys in the right subtree are greater.

Knowing the probabilities of searching each one of the keys makes it easy to compute the expected cost of accessing the tree.

An OBST is a BST which has minimal expected cost.

Example

Key	-5	1	8	7	13	21
Probabilities	1/8	1/32	1/16	1/32	1/4	1/2



$$E[\text{cost}] = 1 \cdot 1/16 + 2 \cdot (1/32 + 1/4) + 3 \cdot (1/8 + 1/32 + 1/2)$$

Tree is **not optimal**: if the 21 is closer to the root, given its high probability, the tree will have a lower expected cost.

OBST (cont'd)

Criterion for an optimal tree: Each OBST is composed of a root and (at most) two optimal subtrees (left and right).

Method: For the root (and each node in turn) we select one value to be stored in the node. (We have n possibilities to do that.)

Once this choice is made , the set of keys which go into the left subtree and right subtree is **completely defined**, because the tree is lexicographically ordered.

The left and right subtrees are now constructed recursively
⇒ recursive definition of the optimal cost.

DP for an OBST

Definitions: Let p_i denote the probability of accessing key i ,
let p_{ij} denote the sum of the probabilities from p_i to p_j .

$$T_{ij} = \min_{k=i\dots j} (p_{i,k-1}(1 + T_{i,k-1}) + p_k \cdot 1 + p_{k+1,j}(1 + T_{k-1,j})) \frac{1}{p_{i,j}}$$

First term corresponds to the left subtree.

Second term corresponds to the root,

3rd term to the right subtree.

Every cost is multiplied by its probability.

Simplification: set $p_{i,i-1} = 0$ and $p_{i+1,i} = 0$, so T_{ii} simplifies
to $T_{i,i} = \frac{p_i}{p_{i,i}} = 1$.

DP for an OBST (cont'd)

Procedure is exponential if applied directly. However, the optimal trees are only constructed over **contiguous sets of keys**, and there are at most $\frac{n(n+1)}{2}$ different ones.

Strategy: store optimal cost of a subtree in a matrix \mathbf{T} . T_{ij} : cost of an optimal subtree constructed with the keys i to j .

Then fill the matrix diagonal by diagonal . It is customary to fill the matrix with $p_{ij} \cdot T_{ij}$, \Rightarrow save a lot of multiplications and divisions. Let $T_{ij}^* = p_{ij} \cdot T_{ij}$ then

$$T_{ij}^* = \min_{k=i \dots j} (T_{i,k-1}^* + p_{ij} + T_{k-1,j}^*)$$

An optimal tree with one node is just the node itself (no other choice), so the diagonal of \mathbf{T}^* is easy to fill: $T_{ii}^* = p_i$.

DP for an OBST

$$\mathbf{T}^* = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & \frac{1}{8} & \frac{3}{16} & \frac{9}{32} & \frac{15}{32} & \frac{31}{32} & \frac{63}{32} \\ 2 & 0 & \frac{1}{32} & \frac{3}{32} & \frac{7}{32} & \frac{19}{32} & \frac{47}{32} \\ 3 & & 0 & \frac{1}{32} & \frac{1}{8} & \frac{15}{32} & \frac{21}{16} \\ 4 & & & 0 & \frac{1}{16} & \frac{3}{8} & \frac{19}{16} \\ 5 & & & & 0 & \frac{1}{4} & 1 \\ 6 & & & & & 0 & \frac{1}{2} \end{array} = \frac{1}{32} \times \begin{pmatrix} 4 & 6 & 9 & 15 & 31 & 63 \\ & 1 & 3 & 7 & 19 & 47 \\ & & 1 & 4 & 15 & 42 \\ & & & 2 & 12 & 38 \\ & & & & 8 & 32 \\ & & & & & 16 \end{pmatrix}$$

Comments:

- The cost of the *OBST* is in $T_{1,n}$ ($T_{1,6}$ in our example).
- It is practical to work with frequencies rather than with probabilities to avoid fractions as matrix-entries.

Optimal Order of Matrix Multiplications

Given: a product of matrices $ABCDE$. There are many ways of evaluating this product (using the **associativity of matrix multiplication**). Let the dimensions of the matrices be:

$$\mathbf{A} : \text{col}(\mathbf{A}) \cdot \text{row}(\mathbf{A}) \quad \text{with } \text{row}(\mathbf{A}) = \text{col}(\mathbf{B})$$

$$\mathbf{B} : \text{col}(\mathbf{B}) \cdot \text{row}(\mathbf{B}) \quad \text{with } \text{row}(\mathbf{B}) = \text{col}(\mathbf{C})$$

$$\mathbf{C} : \text{col}(\mathbf{C}) \cdot \text{row}(\mathbf{C}) \quad \quad \quad \vdots$$

$$\vdots \quad \quad \quad \vdots$$

Definition: $N(n)$ = number of possible orderings to multiply n matrices.

Recursive calculations

Define $N(n)$ **recursively** by looking at last product (which can be done in $n - 1$ different ways)

$$N(n) := \sum_{i=1}^{n-1} N(i) \cdot N(n - i) \text{ for } n \geq 2, \quad (1)$$

$$N(0) := 0, \text{ and } N(1) := 1.$$

We have $N(2) = N(1) \cdot N(1) = 1$, $N(3) = N(1) \cdot N(2) + N(2) \cdot N(1) = 2$, etc.

Recursive calculations (cont'd)

Such convolution sums are normally solved using **generating functions**:

$$N(z) := \sum_{i=0}^{\infty} N(i)z^i = \sum_{i=1}^{\infty} N(i)z^i = z + \sum_{i=2}^{\infty} N(i)z^i. \quad (2)$$

It can be shown that:

$$N(n) = \frac{1}{2n-1} \binom{2n-1}{n},$$

which is exponential in n .

n	$N(n)$
2	1
3	2
4	5
5	14
6	42
7	132
8	429
9	1430
10	4862
15	2674440
20	1767263190
25	1289904147324
30	1002242216651368

The cost of calculating $\mathbf{A} \cdot \mathbf{B}$ with dimensions $\text{row}(\mathbf{A}) \times \text{col}(\mathbf{A})$ and $\text{row}(\mathbf{B}) \times \text{col}(\mathbf{B})$ and $\text{col}(\mathbf{A}) = \text{row}(\mathbf{B})$ is: $\text{col}(\mathbf{A}) \cdot \text{row}(\mathbf{A}) \cdot \text{col}(\mathbf{B})$.

So, given the dimensions, the optimal cost is

$$T_{ij} = \min_{k=i \dots j-1} (T_{ik} + T_{k+1,j} + \text{row}(i) \cdot \text{col}(k) \cdot \text{col}(j)).$$

The cost of $T_{ii} = 0$.

Finding the optimal solution is now completely analogous to finding an Optimal Binary Search Tree.