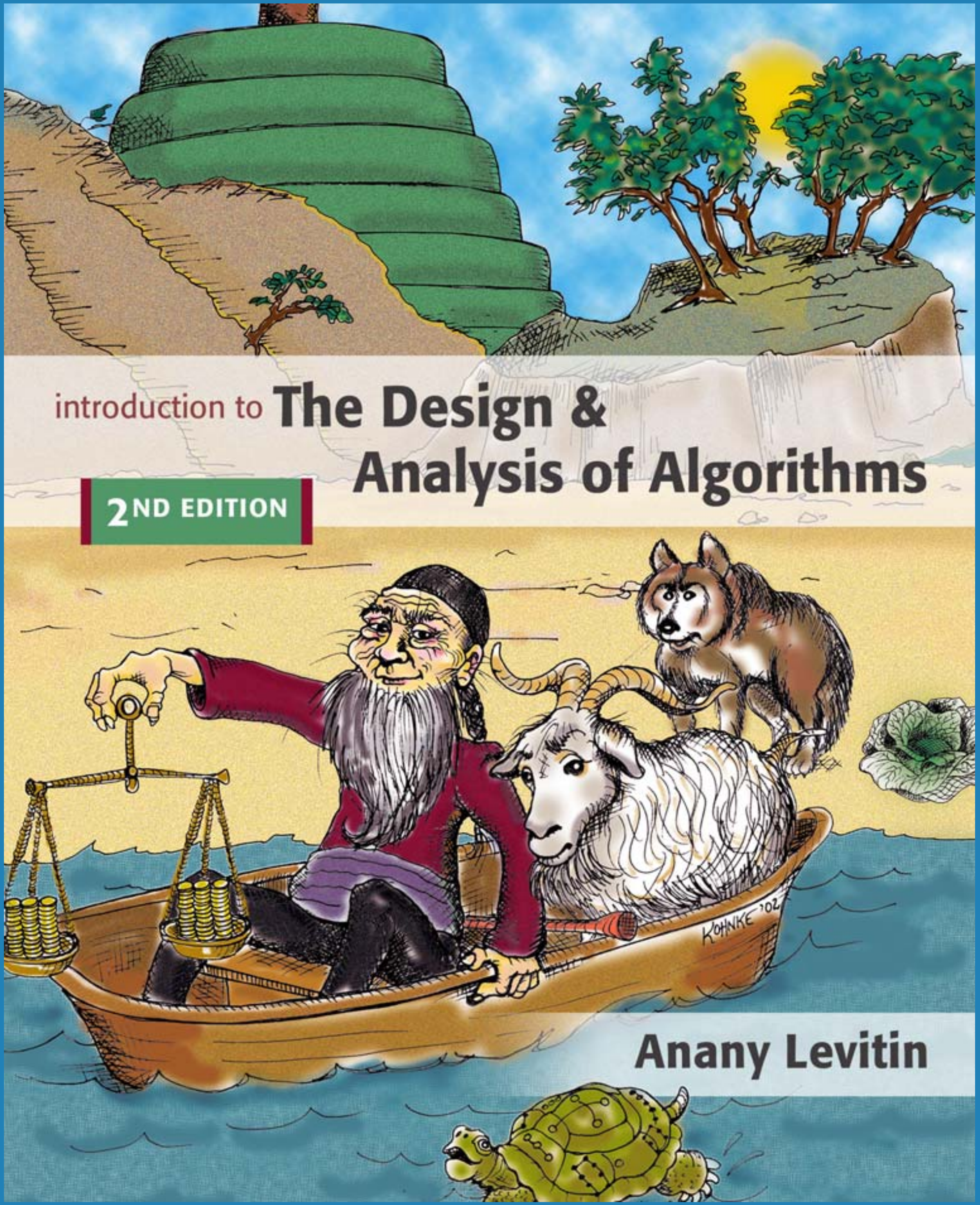
Chapter 8

Dynamic Programming



Dynamic Programming

Dynamic Programming **is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems**

- **Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS**

- **"Programming" here means "planning"**

- **Main idea:**
  - **set up a recurrence relating a solution to a larger instance to solutions of some smaller instances**
  - **solve smaller instances once**
  - **record solutions in a table**
  - **extract solution to the initial instance from that table**

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the n^{th} Fibonacci number recursively (top-down):**

$$F(n)$$

$$F(n-1) \quad + \quad F(n-2)$$

$$F(n-2) \quad + \quad F(n-3) \qquad F(n-3) \quad + \quad F(n-4)$$

...

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- time
- space

Examples of DP algorithms

- Computing a binomial coefficient
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a+b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

$C(n,0) = 1$, $C(n,n) = 1$ for $n \geq 0$

Value of $C(n,k)$ **can be computed by filling a table:**

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

Computing $C(n,k)$: pseudocode and analysis

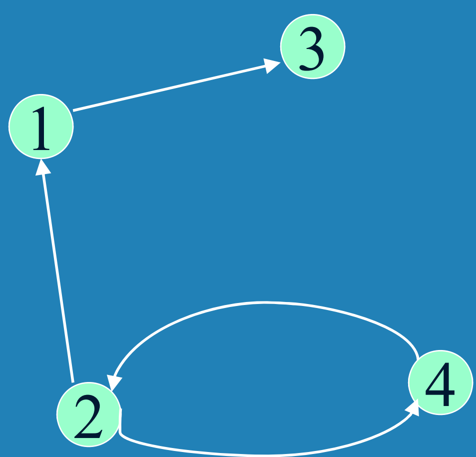
```
ALGORITHM Binomial(n, k)
    //Computes C(n, k) by the dynamic programming algorithm
    //Input: A pair of nonnegative integers n ≥ k ≥ 0
    //Output: The value of C(n, k)
    for i ← 0 to n do
        for j ← 0 to min(i, k) do
            if j = 0 or j = i
                C[i, j] ← 1
            else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
    return C[n, k]
```

Time efficiency: $\Theta(nk)$

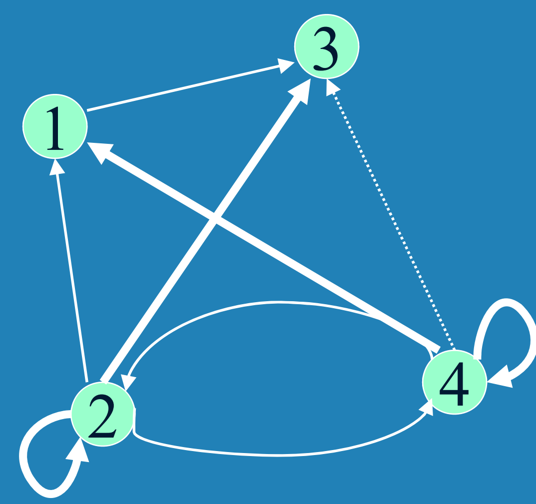
Space efficiency: $\Theta(nk)$

Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph
- Example of transitive closure:



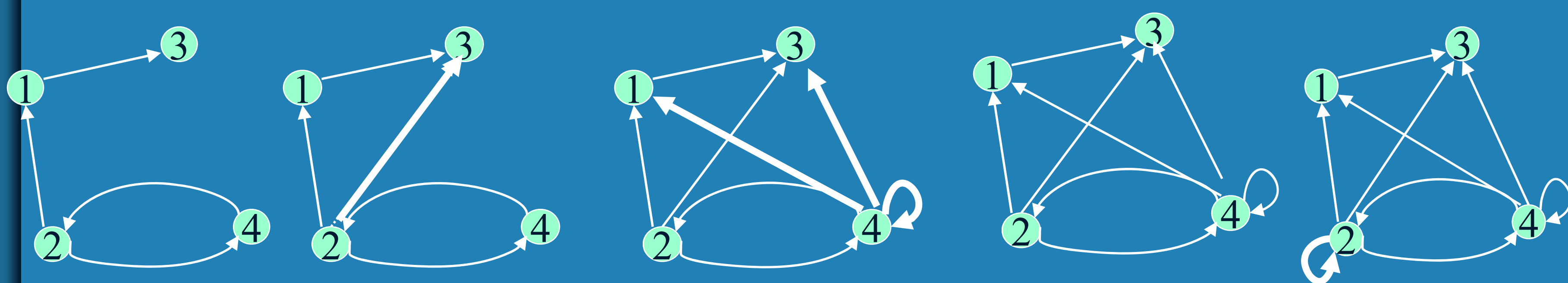
0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0



0	0	1	0
1	**1**	**1**	1
0	0	0	0
1	1	**1**	**1**

Warshall's Algorithm

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where
$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate
Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)



$R^{(0)}$

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)}$

0	0	1	0
1	0	**1**	1
0	0	0	0
0	1	0	0

$R^{(2)}$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	**1**	**1**

$R^{(3)}$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

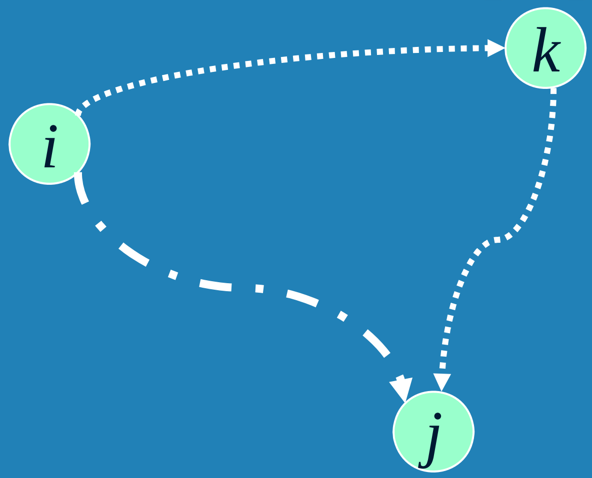
$R^{(4)}$

0	0	1	0
1	**1**	1	1
0	0	0	0
1	1	1	1

Warshall's Algorithm (recurrence)

On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices $1,\ldots,k$ allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-1)} \\ \text{or} & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1,\ldots,k\text{-1)} \end{cases}$$



Warshall's Algorithm (matrix generation)

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

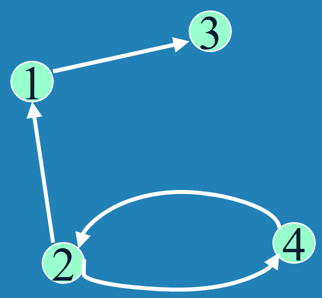
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$

Warshall's Algorithm (example)



$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^{(2)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \qquad R^{(3)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \qquad R^{(4)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm (pseudocode and analysis)

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: $\Theta(n^3)$

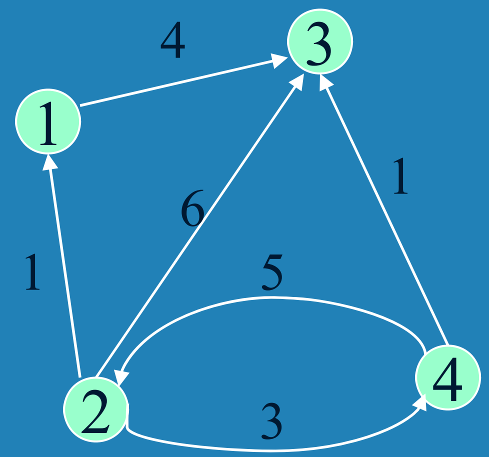
Space efficiency: Matrices can be written over their predecessors

Floyd's Algorithm: All pairs shortest paths

Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

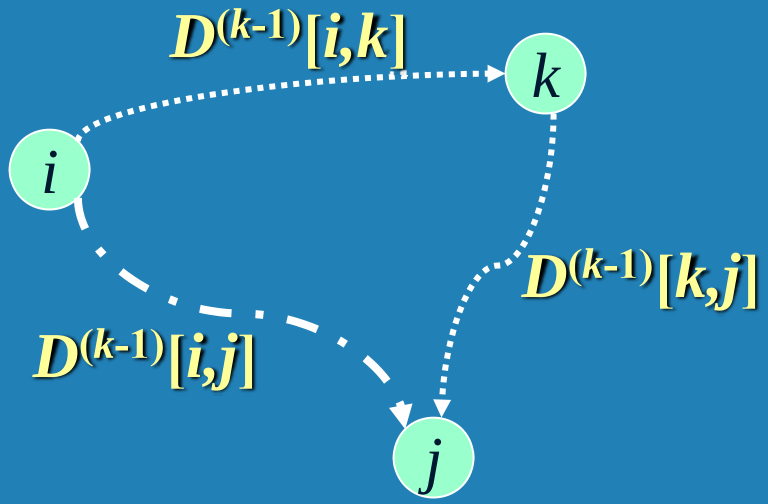
Example:



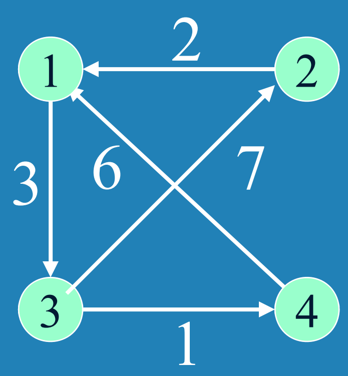
Floyd's Algorithm (matrix generation)

On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among $1,\ldots,k$ as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Floyd's Algorithm (example)



$$D^{(0)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{matrix}$$

$$D^{(1)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{matrix}$$

$$D^{(2)} = \begin{matrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{matrix}$$

$$D^{(3)} = \begin{matrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{matrix}$$

$$D^{(4)} = \begin{matrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{matrix}$$

Floyd's Algorithm (pseudocode and analysis)

ALGORITHM *Floyd*($W[1..n, 1..n]$)

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: Matrices can be written over their predecessors

Note: Shortest paths themselves can be found, too

Optimal Binary Search Trees

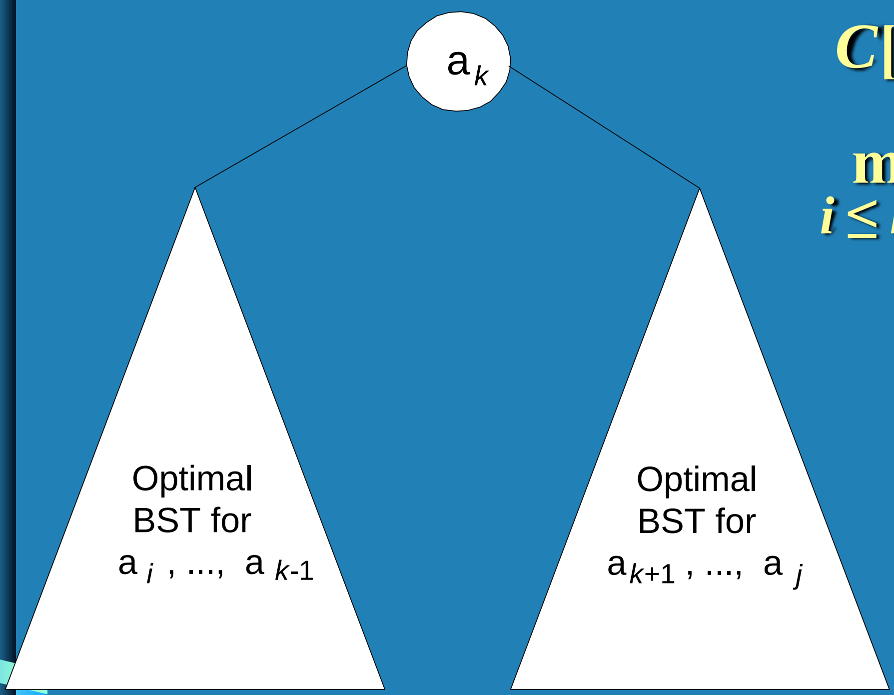
Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \le \ldots \le p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with n nodes is given by C(2n,n)/(n+1), which grows exponentially, brute force is hopeless.

Example: What is an optimal BST for keys *A*, *B*, *C*, and *D* with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?

DP for Optimal BST Problem

Let $C[i,j]$ be minimum average number of comparisons made in T[i,j], optimal BST for keys $a_i < \ldots < a_j$, where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some a_k ($i \le k \le j$) as their root; T[i,j] is the best among them.



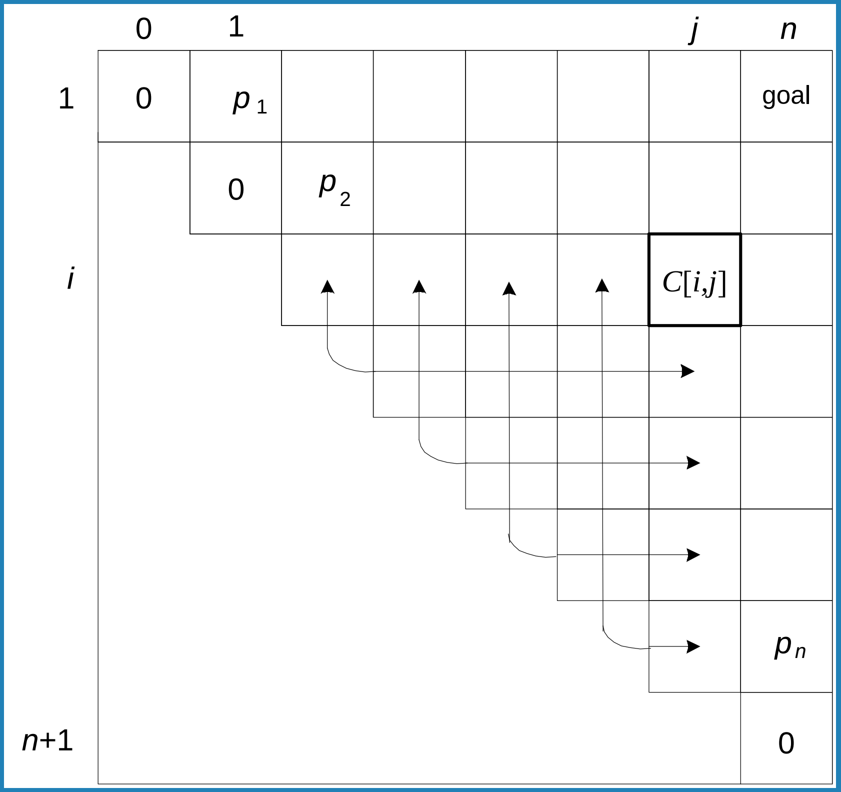
$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \text{ (level } a_s \text{ in T}[i,k-1] + 1) + \sum_{s=k+1}^{j} p_s \text{ (level } a_s \text{ in T}[k+1,j] + 1)\}$$

DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for $C[i,j]$:

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

$$C[i,i] = p_i \quad \text{for } 1 \le i \le j \le n$$



Example:	key	*A*	*B*	*C*	*D*
	probability	0.1	0.2	0.4	0.3

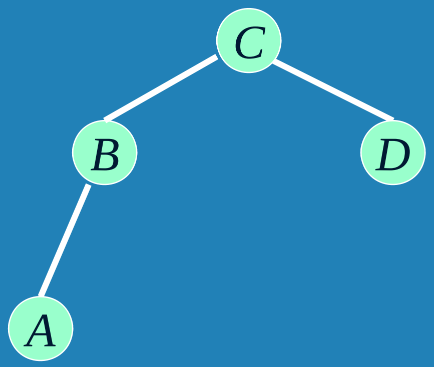
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records *k*'s values giving the minima

i \ *j*	0	1	2	3	4
1	0	.1	.4	1.1	1.7
2		0	.2	.8	1.4
3			0	.4	1.0
4				0	.3
5					0

i \ *j*	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



optimal BST

Optimal Binary Search Trees

```
ALGORITHM  OptimalBST(P[1..n])
    //Finds an optimal binary search tree by dynamic programming
    //Input: An array P[1..n] of search probabilities for a sorted list of n keys
    //Output: Average number of comparisons in successful searches in the
    //           optimal BST and table R of subtrees' roots in the optimal BST
    for i ← 1 to n do
        C[i, i − 1] ← 0
        C[i, i] ← P[i]
        R[i, i] ← i
    C[n + 1, n] ← 0
    for d ← 1 to n − 1 do //diagonal count
        for i ← 1 to n − d do
            j ← i + d
            minval ← ∞
            for k ← i to j do
                if C[i, k − 1] + C[k + 1, j] < minval
                    minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
            R[i, j] ← kmin
            sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
            C[i, j] ← minval + sum
    return C[1, n], R
```

Analysis DP for Optimal BST Problem

Time efficiency: $\Theta(n^3)$ but can be reduced to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., *R*[*i,j*] is always in the range between *R*[*i,j*-1] and R[*i*+1,j]

Space efficiency: $\Theta(n^2)$

Method can be expended to include unsuccessful searches

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \quad \dots \quad w_n$

values: $v_1 \quad v_2 \quad \dots \quad v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

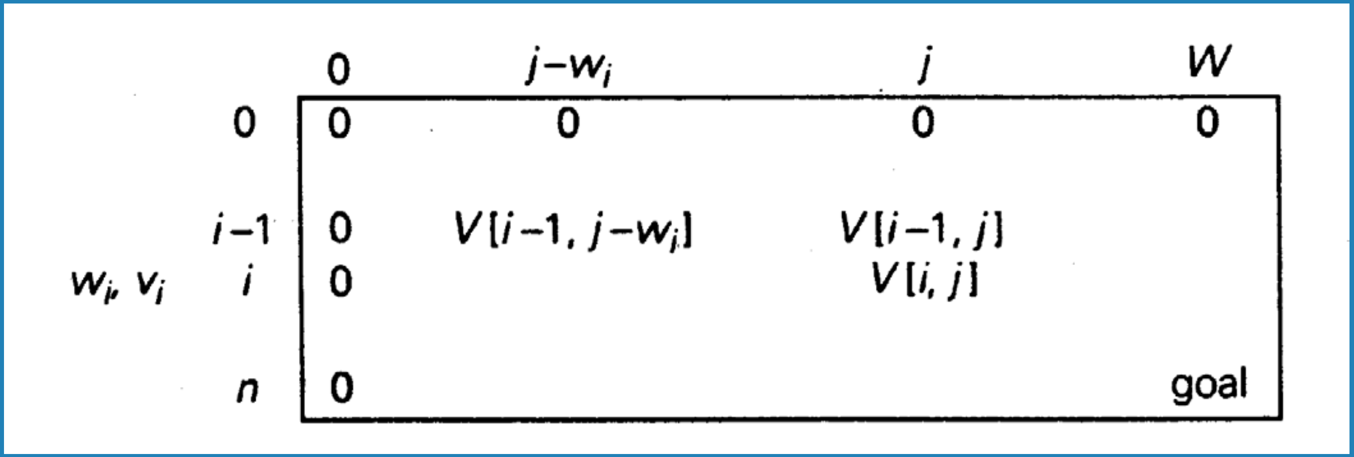
Consider instance defined by first i items and capacity j ($j \leq W$).

Let $V[i,j]$ be optimal value of such instance. Then

$$V[i,j] = \begin{cases} \max \{V[i\text{-}1,j],\ v_i + V[i\text{-}1,j\text{-}\ w_i]\} & \text{if } j\text{-}\ w_i \geq 0 \\ V[i\text{-}1,j] & \text{if } j\text{-}\ w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

Visualizing this Relationship



$$V[i,j] = \begin{cases} \max\{V[i\text{-}1,j],\ v_i + V[i\text{-}1,j\text{-}\ w_i]\} & \text{if } j\text{-}\ w_i \geq 0 \\ V[i\text{-}1,j] & \text{if } j\text{-}\ w_i < 0 \end{cases}$$

- **So we can build up the table, left to right by repeatedly applying the result of this expression**
- **The initial conditions are shown by the first row and first column with 0 values**

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

We know the solution is 37; the next question is how we find the items actually involved.

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

i \ capacity j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Top Down Approach with Memorization

- We can use a top down approach by storing all results
- When we need a value we first ask have we stored it

```
ALGORITHM  MFKnapsack(i, j)
    //Implements the memory function method for the knapsack problem
    //Input: A nonnegative integer i indicating the number of the first
    //        items being considered and a nonnegative integer j indicating
    //        the knapsack's capacity
    //Output: The value of an optimal feasible subset of the first i items
    //Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
    //and table V[0..n, 0..W] whose entries are initialized with −1's except for
    //row 0 and column 0 initialized with 0's
    if V[i, j] < 0
        if j < Weights[i]
            value ← MFKnapsack(i − 1, j)
        else
            value ← max(MFKnapsack(i − 1, j),
                        Values[i] + MFKnapsack(i − 1, j − Weights[i]))
        V[i, j] ← value
    return V[i, j]
```

Example of Memorization

	capacity *j*					
i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–	12	22	–	22
3	0	–	–	22	–	32
4	0	–	–	–	–	37

- **Notice that not all values need to be calculated**
- **Only eleven out of twenty of the nontrivial values (the zeros) need to be computed**