Chapter 8

Dynamic Programming

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Dynamic Programming

Dynamic Programming **is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances**

- **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**

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the nth Fibonacci number recursively (top-down):**

$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

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

$$\ldots$$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Example: Fibonacci numbers (cont.)

Computing the nth 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** n
- **space** n

What if we solve it recursively?

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- 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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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 $n > k > 0$

$C(n,0) = 1, \quad C(n,n) = 1$ 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						$C(n,k)$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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]
```

universitystudy.in

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Knapsack Problem by DP

Given n items of

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

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

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 an instance. Then

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

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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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

capacity j

		0	1	2	3	4	5
	0	0	0	0			
$w_1 = 2, v_1 = 12$	1	0	0	12			
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

Backtracing finds the actual optimal subset, i.e. solution.

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Memory function

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 capacity

//Output: The value of an optimal feasible subset of the first *i items*

//Note: Uses as global variables input arrays *Weights[1..n], V alues[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]***

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of "president": pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.

- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

LCS

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

LCS

Another example:

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm.

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

How to compute LCS?

- Let A=$a_1a_2\ldots a_m$ and $B=b_1b_2\ldots b_n$.
- *len*(*i*, *j*): the length of an LCS between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$
- With proper initializations, *len*(*i*, *j*) can be computed as follows.

$$len(i,j)=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ len(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j\ ,\\ \max(len(i,j-1),len(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j.\end{cases}$$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

procedure *LCS-Length(A, B)*

1. **for** $i \leftarrow 0$ **to** m **do** $len(i,0) = 0$
2. **for** $j \leftarrow 1$ **to** n **do** $len(0,j) = 0$
3. **for** $i \leftarrow 1$ **to** m **do**
4. **for** $j \leftarrow 1$ **to** n **do**
5. **if** $a_i = b_j$ **then** $\left[\begin{array}{l} len(i,j) = len(i-1,j-1)+1 \\ prev(i,j) = \text{"}\nwarrow\text{"} \end{array}\right.$
6. **else if** $len(i-1,j) \geq len(i,j-1)$
7. **then** $\left[\begin{array}{l} len(i,j) = len(i-1,j) \\ prev(i,j) = \text{"}\uparrow\text{"} \end{array}\right.$
8. **else** $\left[\begin{array}{l} len(i,j) = len(i,j-1) \\ prev(i,j) = \text{"}\leftarrow\text{"} \end{array}\right.$
9. **return** *len* and *prev*

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

i	j	0	1	2	3	4	5	6	7	8	9	10
			p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0		0	0	0	0	0	0	0	0	0	0	0
1	*p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	*r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	*e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	*s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	*i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	*d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	*e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	*n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	*t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

universitystudy.in

Running time and memory: O(*mn*) and O(*mn*).

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

The backtracing algorithm

procedure *Output-LCS(A, prev, i, j)*

1 **if** $i = 0$ **or** $j = 0$ **then return**

2 **if** *prev(i, j)=”* $\nwarrow$ *“* **then** $\left[\begin{array}{l} Output-LCS(A, prev, i-1, j-1) \\ \text{print } a_i \end{array}\right.$

3 **else if** *prev(i, j)=”* $\uparrow$ *“* **then** *Output-LCS(A, prev, i-1, j)*

4 **else** *Output-LCS(A, prev, i, j-1)*

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

i		j 0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0		0	0	0	0	0	0	0	0	0	0	0
1	p	0	1	1	1	1	1	1	1	1	1	1
2	r	0	1	2	2	2	2	2	2	2	2	2
3	e	0	1	2	2	2	2	2	3	3	3	3
4	s	0	1	2	2	2	2	2	3	3	3	3
5	i	0	1	2	2	2	3	3	3	3	3	3
6	d	0	1	2	2	2	3	4	4	4	4	4
7	e	0	1	2	2	2	3	4	5	5	5	5
8	n	0	1	2	2	2	3	4	5	6	6	6
9	t	0	1	2	2	2	3	4	5	6	6	6

universitystudy.in

Output: *priden*

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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:

$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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 the 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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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

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

Initial condition?

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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 $R^{(k-1)}$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Warshall's Algorithm (example)

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

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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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)
```

universitystudy.in

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

Space efficiency: Matrices can be written over their predecessors (with some care), so it's $\Theta(n$^2).

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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:

$$\begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 4 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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,…,*k* as intermediate

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

Initial condition?

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Floyd's Algorithm (example)

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

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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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
```

If $D[i,k] + D[k,j] < D[i,j]$ then $P[i,j] \leftarrow k$

universitystudy.in

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

Since the superscripts k or k-1 make no difference to $D[i,k]$ and $D[k,j]$.

Space efficiency: Matrices can be written over their predecessors

Note: Works on graphs with negative edges but without negative cycles. Shortest paths themselves can be found, too. How?

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Optimal Binary Search Trees

Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1, \ldots, p_n$ 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?

Average # of comparisons
= 1*0.4 + 2*(0.2+0.3) + 3*0.1
= 1.7

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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.

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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

DP for Optimal BST Problem (cont.)

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

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

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

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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

universitystudy.in

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
```

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

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 expanded to include unsuccessful searches

universitystudy.in

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.