DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

DYNAMIC PROGRAMMING

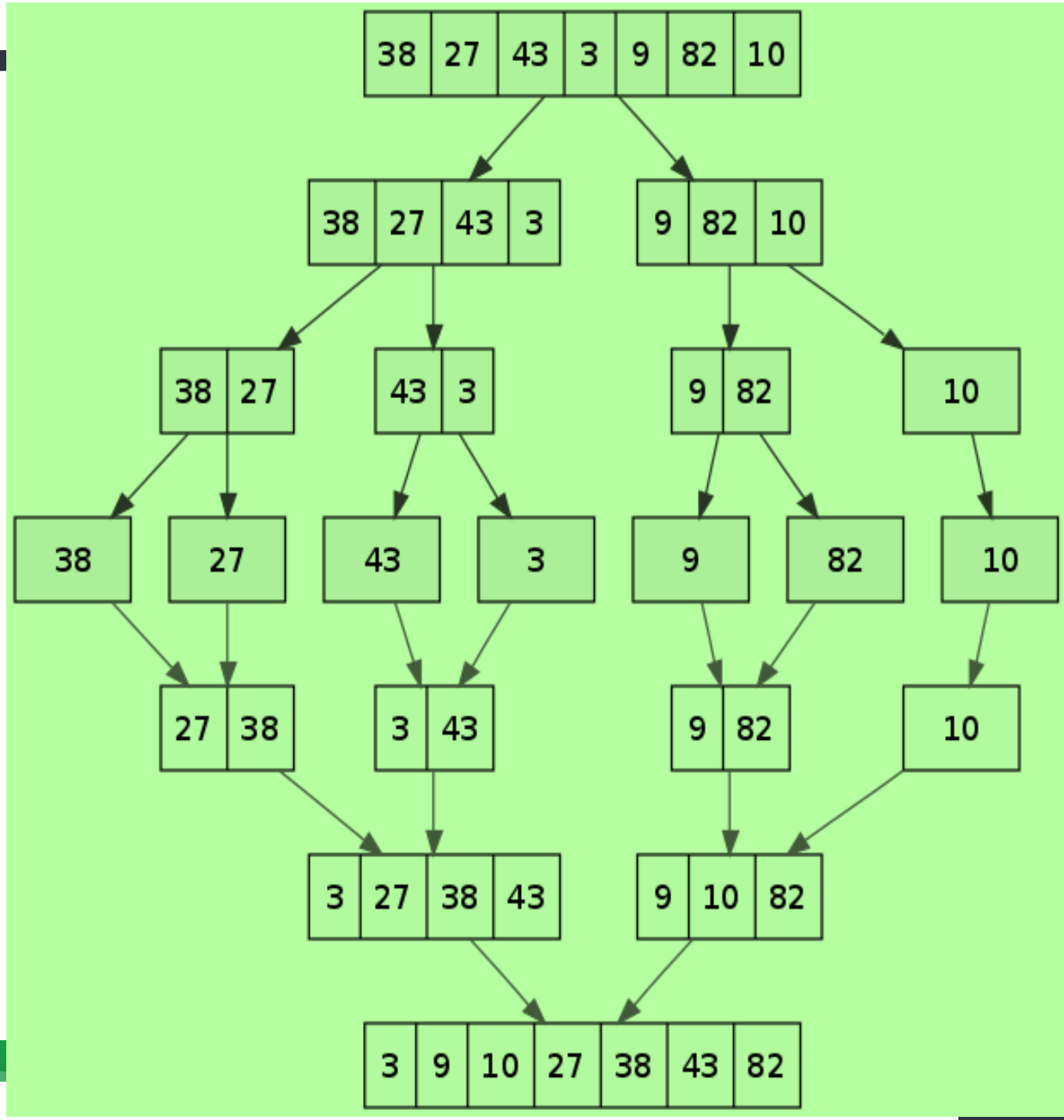
- It was invented by a prominent U.S. mathematician, Richard Bellman
- Thus, the word “programming” in the name of this technique stands for “planning” / “tabulation” and does not refer to computer programming.
- Dynamic programming is a technique for solving problems with overlapping subproblems
  - Typically, these subproblems arise from a recurrence relating a given problem’s solution **to** solutions of its smaller subproblems
- Rather than solving overlapping subproblems again and again,
  - dynamic programming suggests solving each of the smaller subproblems only once , and
  - recording the results in a table from which a solution to the original problem can then be obtained.

DYNAMIC PROGRAMMING

- A widely used technique to solve optimization problems
- Optimal=maximization/minimization
- The main difference between divide and conquer and dynamic programming is that
  - the divide and conquer combines the solutions of the sub-problems to obtain the solution of the main problem
  - while dynamic programming uses the result of the sub-problems to find the optimum solution of the main problem.
  - A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table

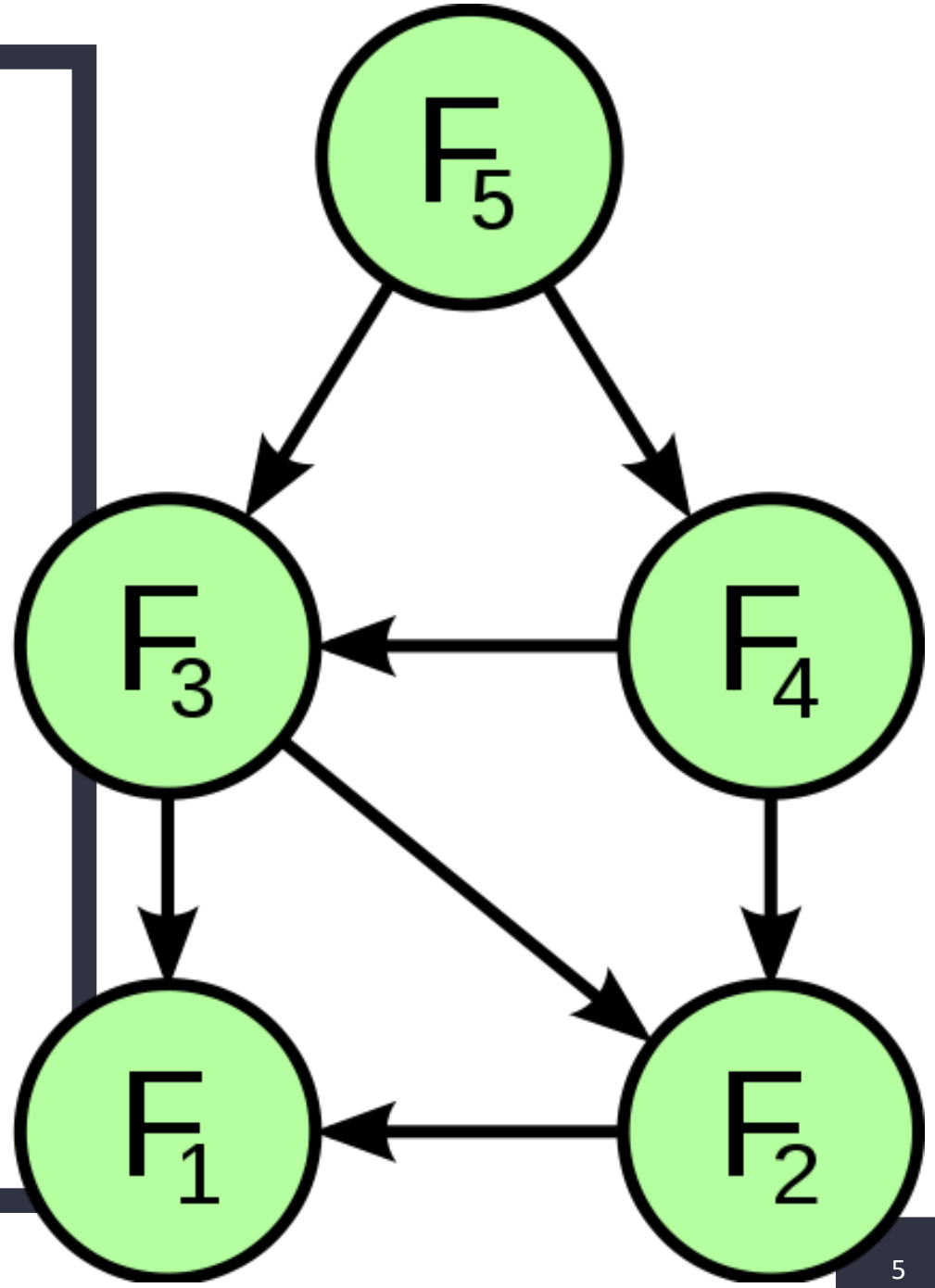
WHAT IS DIVIDE AND CONQUER ?

- Divide and conquer divides the main problem into small subproblems.
- The subproblems are divided again and again. At one point, there will be a stage where we cannot divide the subproblems further.
- Then, we can solve each subproblem independently.
- Finally, we can combine the solutions of all subproblems to get the solution to the main problem.



WHAT IS DYNAMIC PROGRAMMING

- Dynamic programming divides the main problem into smaller subproblems, but it does not solve the subproblems independently.
- It stores the results of the subproblems to use when solving similar subproblems. Storing the results of subproblems is called memorization.
- Before solving the current subproblem, it checks the results of the previous subproblems.
- Finally, it checks the results of all subproblems to find the best solution or the optimal solution. This method is effective as it does not compute the answers again and again. Usually, dynamic programming is used for optimization.



DIVIDE AND CONQUER VERSUS DYNAMIC PROGRAMMING

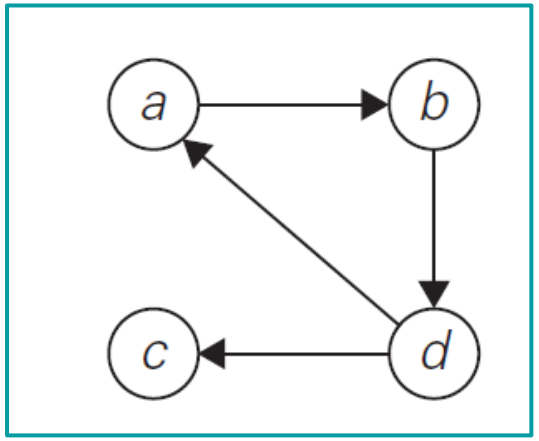
DIVIDE AND CONQUER	DYNAMIC PROGRAMMING
An algorithm that recursively breaks down a problem into two or more sub-problems of the same or related type until it becomes simple enough to be solved directly	An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property
Subproblems are independent of each other	Subproblems are interdependent
Recursive	Non-recursive
More time-consuming as it solves each subproblem independently	Less time-consuming as it uses the answers of the previous subproblems
Less efficient	More efficient
Used by merge sort, quicksort, and binary search	Used by matrix chain multiplication, optimal binary search tree

Visit www.PEDIAA.com

DYNAMIC PROGRAMMING

Warshall's Algorithm

- An algorithm for computing the transitive closure of a directed graph
- The ***transitive closure*** of a directed graph with *n* vertices can be defined as the ***n* × *n*** boolean matrix ***T* = {t_{ij}}**, in which the element in the ***i*th** row and the ***j*th** column is ***1*** if there exists a nontrivial path (i.e., directed path of a positive length) from the ***i*th** vertex to the ***j*th** vertex; otherwise, t_{ij} is ***0***.



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Transitive closure of the digraph, would allow us to determine in constant time whether the j^{th} vertex is reachable from the i^{th} vertex.
- We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search.?
- Doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.
- But this method traverses the same digraph several times.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Transitive closure of the digraph, would allow us to determine in constant time whether the j^{th} vertex is reachable from the i^{th} vertex.
- We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search.?
- Doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.
- But this method traverses the same digraph several times.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

- Each of these matrices provides certain information about directed paths in the digraph.
- The element $r^{(k)}_{ij}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$ $_{(i, j)}$= 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to 1 if and only if there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k

DYNAMIC PROGRAMMING

Warshall's Algorithm

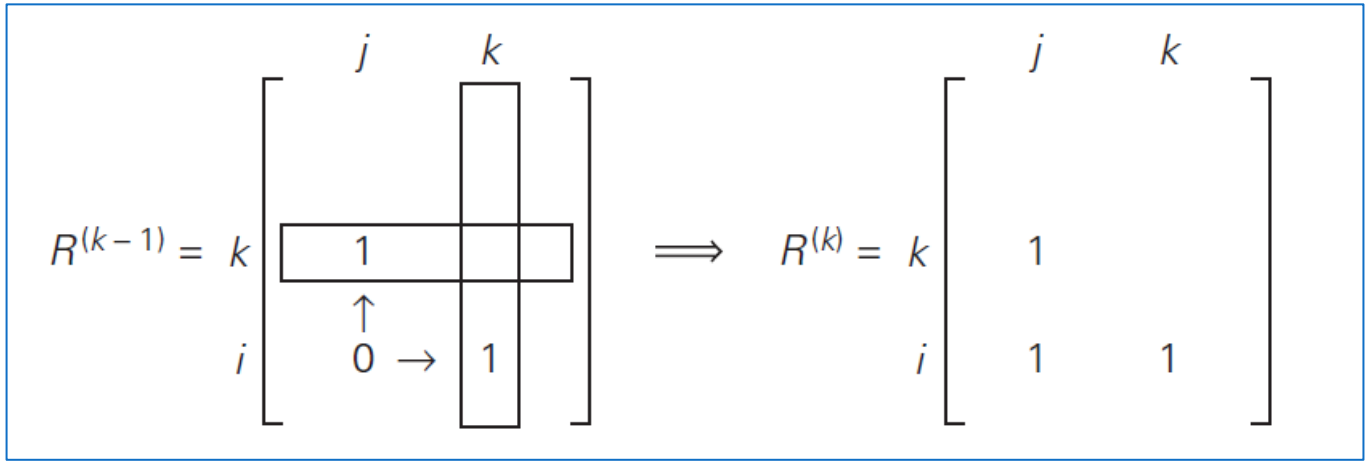
- Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$
- In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's.
- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series

$$r_{ij}^{k} = r_{ij}^{k-1} \; or \; (r_{ik}^{k-1} \text{ and } r_{kj}^{k-1})$$



DYNAMIC PROGRAMMING

Warshall's Algorithm

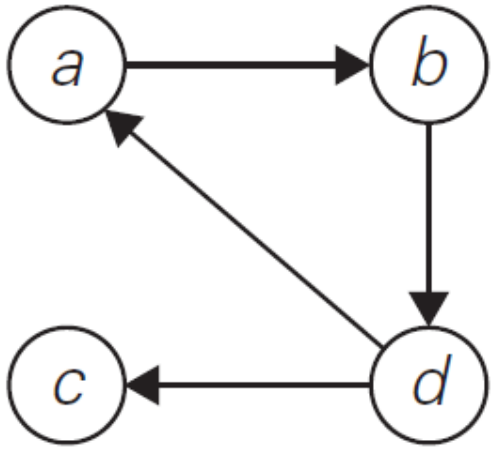
$$r_{ij}^{k} = r_{ij}^{k-1} \, or \, (r_{ik}^{k-1} \text{ and } r_{kj}^{k-1})$$

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} 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 $R^{(k-1)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Obtain the transitive closure for the following digraph



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

DYNAMIC PROGRAMMING

Warshall's Algorithm

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths);
boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

DYNAMIC PROGRAMMING

Warshall's Algorithm

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

$R^{(0)} \leftarrow A$

for $k \leftarrow 1$ **to** n **do**

  for $i \leftarrow 1$ **to** n **do**

    for $j \leftarrow 1$ **to** n **do**

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

return $R^{(n)}$

Time Complexity? $\theta(n^3)$

DYNAMIC PROGRAMMING

Warshall's Algorithm

- Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix:

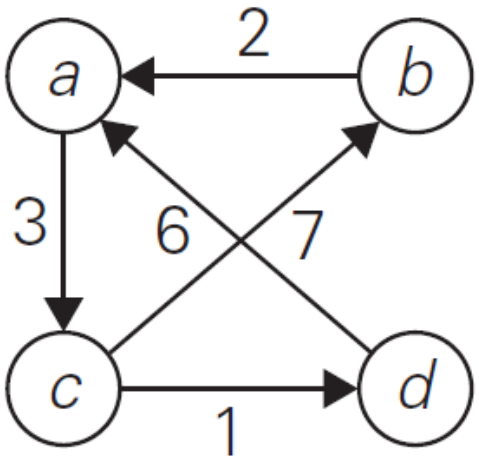
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.
- **Major applications**
  - communications,
  - transportation networks,
  - operations research
  - precomputing distances for motion planning in computer games. Etc.

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***:
  - the element d^{ij} in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex.



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- The distance matrix can be generated by an algorithm that is very similar to **Warshall's algorithm**,
- It is **Floyd's algorithm** named after Robert W Floyd
- It is applicable to both undirected and directed weighted graphs provided they do not contain a cycle of a negative length
- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$$

- Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

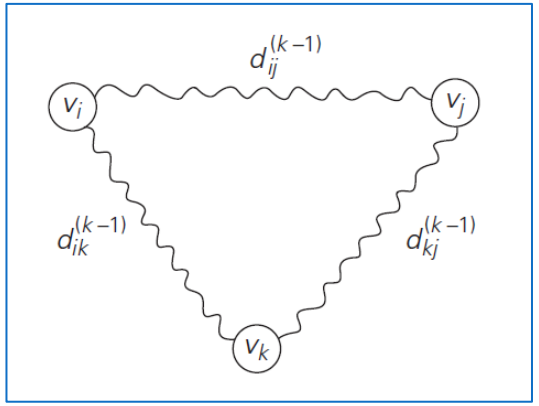
- The element $d^{(k)}_{ij}$ in the i^{th} row and the j^{th} column of matrix $D(k)$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to the length of the shortest path among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than **k**.
- The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence,
  - $D^{(0)}$ is simply the weight matrix of the graph.
- The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate.
- We can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- The recurrence relation for the Flovd's algorithm is given bv

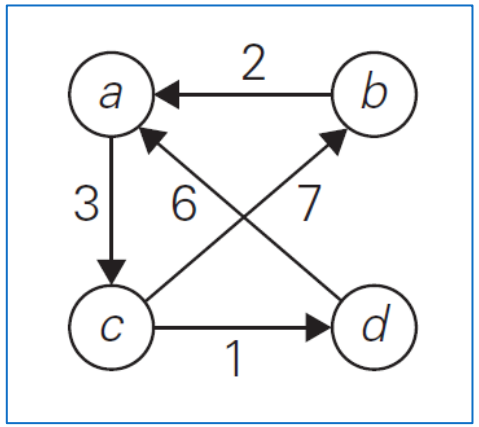
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}.$$

- the element in row ***i*** and column ***j*** of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row ***i*** and the column ***k*** and in the same column ***j*** and the row ***k*** if and only if the latter sum is smaller than its current value.



FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- **Obtain the shortest paths from each vertex to all other vertices in the graph shown**



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*)

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

- Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

FLOYD'S ALGORITHM FOR THE ALL-PAIRS SHORTEST-PATHS PROBLEM

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

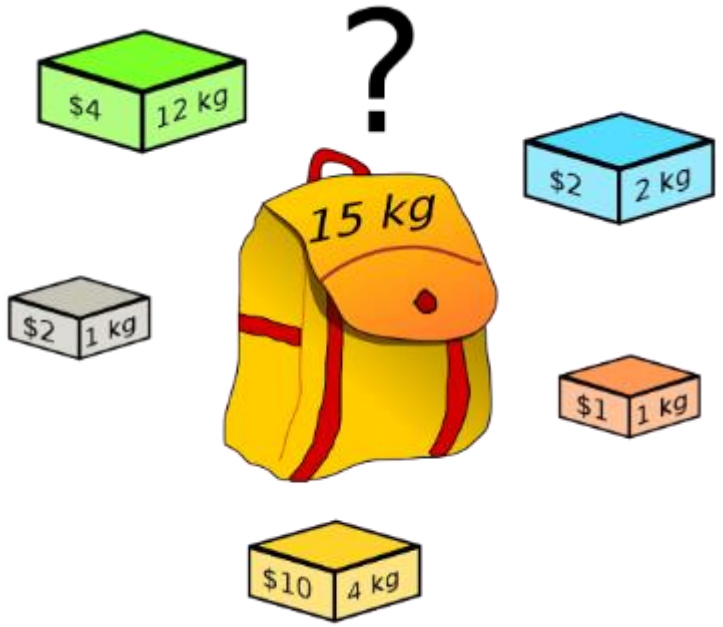
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
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

Time Complexity? $\theta(n^3)$

DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

0/1 KNAPSACK PROBLEM



Items are indivisible, you either select item or not !!!!!

Problem Statement :

Given n items of

integer weights: w_1 w_2 … w_n

values: v_1 v_2 … v_n

and a knapsack of integer capacity W ,

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

Assumptions: Weights and capacity are positive integers, values need not have to be integers

0/1 KNAPSACK PROBLEM

- To design a dynamic programming algorithm, recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub instances is needed
- Consider such a smaller sub instance defined by first i items, $1 \leq i \leq n$, with weights $w1, \ldots, wi$, values $v1, \ldots, vi$, and knapsack capacity j, $1 \leq j \leq W$.
- Let *F(i, j)* be the value of an optimal solution to this instance,
  - i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

0/1 KNAPSACK PROBLEM

- We can divide all the subsets of the first ***i*** items that fit the knapsack of capacity ***j*** into two categories: those that do not include the i^{th} item and those that do.
- Among the subsets that **do not include** the i^{th} item, the value of an optimal subset is, by definition, ***F(i – 1, j).***
- Among the subsets that **do include** the i^{th} item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first ***i – 1*** items that fits into the knapsack of capacity $j - w_i$.
  - The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

0/1 KNAPSACK PROBLEM

- Hence, the value of an optimal solution among all feasible subsets of the first ***i*** *i*tems is the maximum of these two values.
- If the i^{th} item does not fit into the knapsack, the value of an optimal subset selected from the first ***i*** items is the same as the value of an optimal subset selected from the first ***i* – 1** items.
- With these observations we can arrive at following recurrence

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

- Following are the initial conditions

$F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$.

0/1 KNAPSACK PROBLEM

- The goal is to find ***F(n, W),*** the maximal value of a subset of the ***n*** given items that fit into the knapsack of capacity ***W,*** and an optimal subset itself.

F=

w_i, v_i		0	$j - w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
	i	0		$F(i, j)$	
	n	0			goal

0/1 KNAPSACK PROBLEM

- Solve the following instance of knapsack problem to obtain the optimal solution using dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

knapsack capacity = 5 kg,

0/1 KNAPSACK PROBLEM

Solution

- Draw a table say with **(n+1) = 4 + 1 = 5** number of rows and **(w+1) = 5 + 1 = 6** number of columns.
- Fill all the boxes of **0th** row and **0th** column with 0. (initial condition)

F=

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

- Start filling this table using the recurrence relation

0/1 KNAPSACK PROBLEM

- $\boldsymbol{F(1,1) = ?}$
- $i = 1, j = 1, v_1 = 3, w_1 = 2$
- Find whether $\boldsymbol{j - wi < 0?}$ **YES**
- Then as per the recurrence $\boldsymbol{F(1,1)} = F(1-1,1) = F(0,1) = \boldsymbol{0}$
- $\boldsymbol{F(1,2) = ?}$
- $i = 1, j = 2, v_1 = 3, w_1 = 2$
- Find whether $\boldsymbol{j - wi < 0?}$ **NO**
- $\boldsymbol{F(1,2)} = \max\{F(0,2), 3 + F(0,0)\} = \boldsymbol{3}$

0/1 KNAPSACK PROBLEM

- $\boldsymbol{F(1,3) = ?}$
- $i = 1, j = 3, v_1 = 3, w_1 = 2$
- Find whether $\boldsymbol{j - wi < 0?}$ **NO**
- $\boldsymbol{F(1,3) = \max\{F(0,3), 3 + F(0,1)\} = 3}$
- $\boldsymbol{F(1,4) = ?}$
- $i = 1, j = 4, v_1 = 3, w_1 = 2$
- Find whether $\boldsymbol{j - wi < 0?}$ **NO**
- $\boldsymbol{F(1,4) = \max\{F(0,4), 3 + F(0,2)\} = 3}$

0/1 KNAPSACK PROBLEM

- $\boldsymbol{F(1,5) = ?}$
- $i = 1, j = 5, v_1 = 3, w_1 = 2$
- Find whether $\boldsymbol{j - wi < 0?}$ **NO**
- $\boldsymbol{F(1,5) = \max\{F(0,5), 3 + F(0,3)\} = 3}$
- $\boldsymbol{F(2,1) = ?}$
- $i = 2, j = 1, v_2 = 4, w_2 = 3$
- Find whether $\boldsymbol{j - wi < 0?}$ **YES**
- $\boldsymbol{F(2,1) = F(1,1) = 0}$

0/1 KNAPSACK PROBLEM

- $\boldsymbol{F(2,2,)} = ?$
- $i = 2, j = 2, v_2 = 4, w_2 = 3$
- Find whether $\boldsymbol{j - wi < 0}$? **YES**
- $\boldsymbol{F(2,2)} = F(1,2) = \boldsymbol{3}$
- $\boldsymbol{F(2,3)} = ?$
- $\boldsymbol{F(2,3)} = ?$ **and so on till F(4,5)**

0/1 KNAPSACK PROBLEM

- The final table

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

0/1 KNAPSACK PROBLEM

- Solve the following instance of knapsack problem to obtain the optimal solution using dynamic programming approach.

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

knapsack capacity = 5 kg,

0/1 KNAPSACK PROBLEM

- Solution

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	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**

Optimal solution

DYNAMIC PROGRAMMING

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

TRAVELING SALESMAN PROBLEM

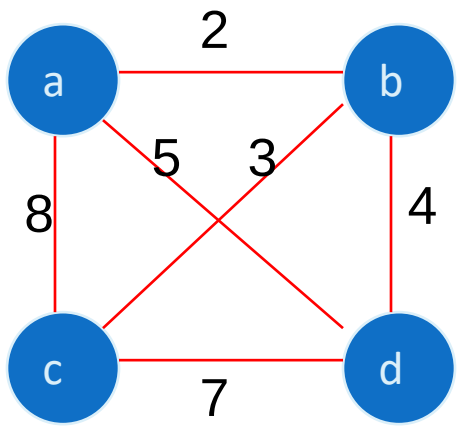
- 0/1 knapsack problem was an example for subset selection problem.
- Talking about permutation problems, they are much more harder to solve compared to subset selection problem
- as there are **n!** different permutations of **n** objects where as there are only $\mathbf{2^n}$ different subsets of **n** objects $\mathbf{(n!>2^n)}$
- Travelling salesman problem is a perfect example for permutation problem

TRAVELING SALESMAN PROBLEM

- Given **n** vertices the salesman wishes to make a ***tour***, visiting each city exactly once and finishing at the city he starts from
- Given a graph **G= (V, E)** with **n** vertices and the **source** vertex, the salesman wishes to make a ***tour***, starting from the **source** vertex, visiting each vertex **exactly once** and **finishing** at the vertex he started from (**source**).
- Principle of Optimality ?
- Different techniques to solve this problem
  - Brute Force approach
  - Branch and bound
  - Dynamic programming

TRAVELING SALESMAN PROBLEM

- **Brute Force approach**

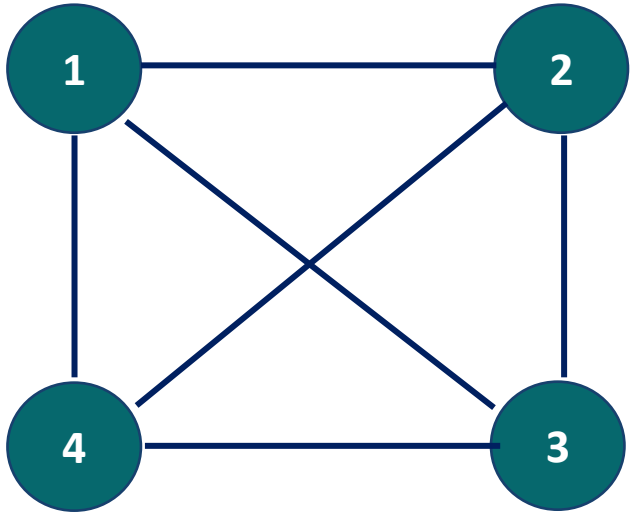


Let source be **a**

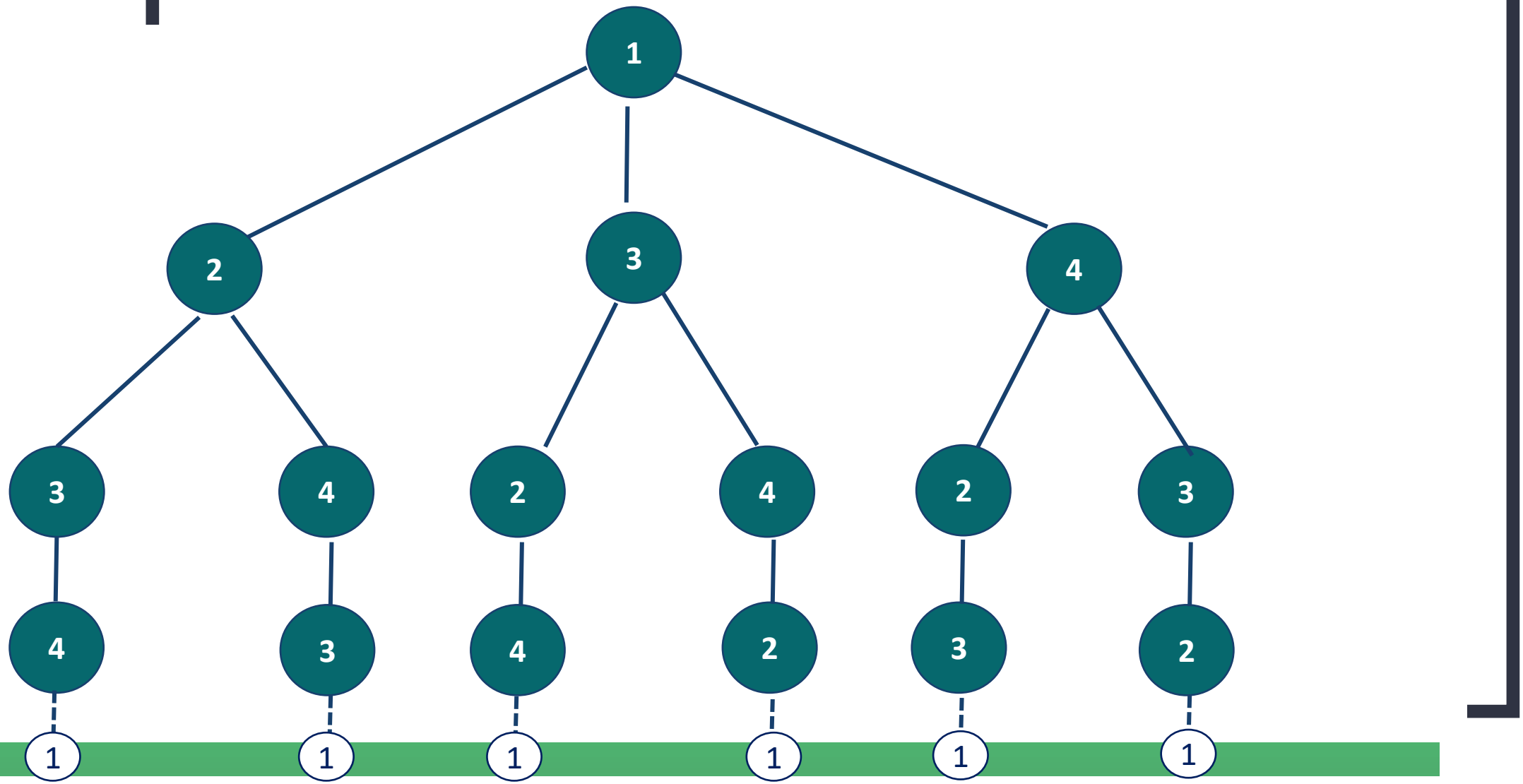
Tour	Cost	
a→b→c→d→a	2+3+7+5 = 17	Best Tour
a→b→d→c→a	2+4+7+8 = 21	
a→c→b→d→a	8+3+4+5 = 20	
a→c→d→b→a	8+7+4+2 = 21	
a→d→b→c→a	5+4+3+8 = 20	
a→d→c→b→a	5+7+3+2 = 17	Best Tour

TRAVELING SALESMAN PROBLEM

- $$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$



TRAVELING SALESMAN PROBLEM



TRAVELING SALESMAN PROBLEM

- Let $\boldsymbol{G = (V, E)}$ be a directed graph with edge costs $\mathbf{c_{ij}}$.
- The variable $\boldsymbol{c_{ij}}$ is defined such that $\boldsymbol{c_{ij} > 0}$ for all i and j and $\boldsymbol{c_{ij} > \infty}$ $\boldsymbol{if (i, j) \notin E}$.
- Let $\boldsymbol{|V|}$**=n** and assume **n>1**
- A tour of **G** is a directed simple cycle that includes every vertex in **V**.
- The cost of the tour is the sum of the cost of the edges on the tour.
- The TSP is to find a tour of minimum cost.
- Applications
  - Postal van routing
  - Production environment

TRAVELING SALESMAN PROBLEM

The solution approach

- Consider a tour to be a simple path that starts and ends at vertex **1**
- Every tour consists of an edge **(1,k)** for some $\boldsymbol{k} \in \boldsymbol{V} - \{\mathbf{1}\}$ and a path from vertex **k to 1**
- The path from vertex k to vertex 1 goes through each vertex in $\boldsymbol{V} - \{\mathbf{1}, \boldsymbol{k}\}$ exactly once
- Its is easy to see that if the tour is optimal, then the path from k to **1** must be the shortest k to 1 path going through all vertices in $\boldsymbol{V} - \{\mathbf{1}, \boldsymbol{k}\}$

TRAVELING SALESMAN PROBLEM

- Let $g(i, S)$ be the length of a shortest path starting at vertex i ,going through all vertices in S, and terminating at vertex 1
- The function $g(1, V - \{1\})$ is the length of an optimal salesman tour
- So

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \qquad \text{A}$$

generalizing the above equation we have

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \qquad \text{B}$$

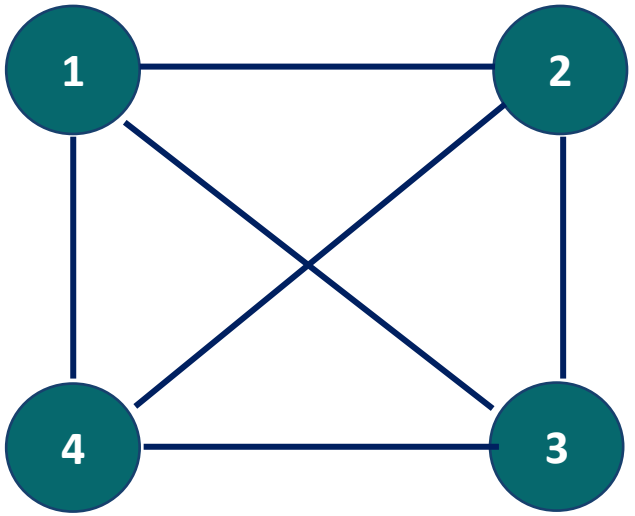
- $g(i, \phi) = c_{i1}$, $1 \le i \le n$.
- Using equation **B** we can get **g(i,S)** for all $|S|$ **=1, 2** and so on
- When $|S| < n - 1$, the values of i and S for which **g(i,S)** is needed are such that

$$i \neq 1, 1 \notin S \; and \; i \notin S$$

TRAVELING SALESMAN PROBLEM

- Solve the following instance of **TSP** using dynamic programming

- $$\begin{array}{c} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$



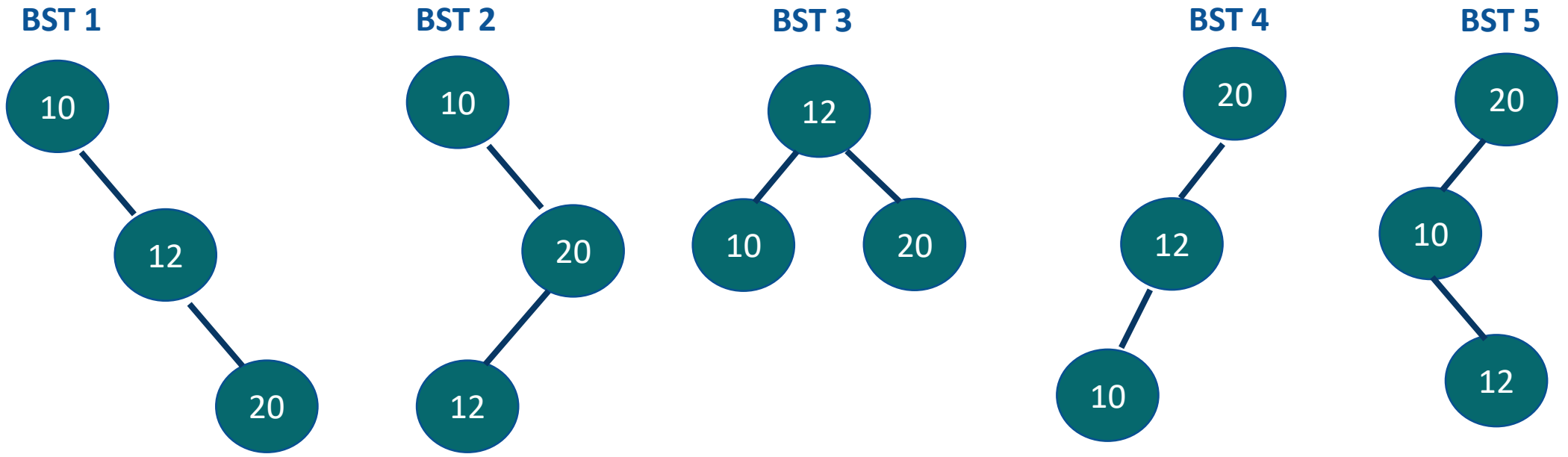
OPTIMAL BINARY SEARCH TREES

- A binary search tree is one of the most important data structures in computer science
- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes.
  - The external nodes are null nodes.
  - The keys are ordered **lexicographically**, i.e. for each internal node all the keys in the **left sub-tree** are **less than** the keys in the node, and all the keys in the **right sub-tree** are **greater**.
- The major applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion
- If probabilities of searching for elements of a set are known
  - e.g., from accumulated data about past searches
  - it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible

OPTIMAL BINARY SEARCH TREES

Example

- Keys={10, 12, 20} Frequency=(34, 8, 50}
- Possible BST are

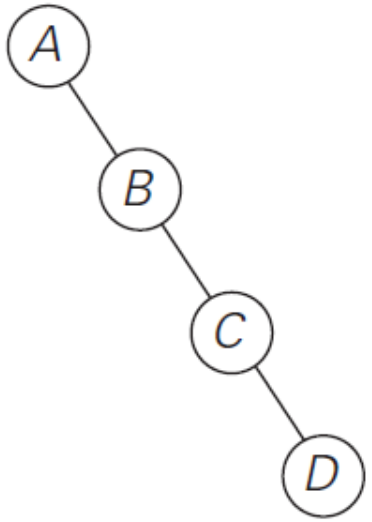


OPTIMAL BINARY SEARCH TREES

- Example

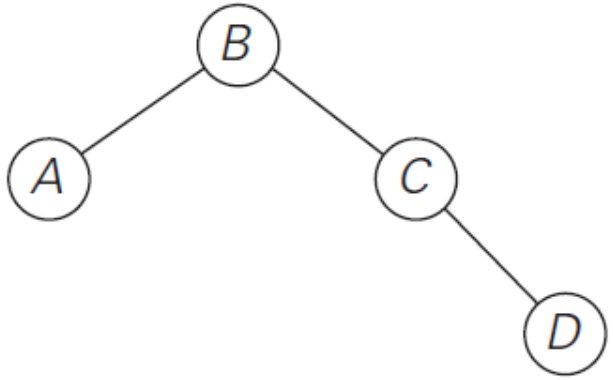
Keys	Probability
A	0.1
B	0.2
C	0.4
D	0.3

BST 1



Avg. no. of successful comparison
0.1 . 1+ 0.2 . 2 + 0.4 .3+ 0.3 . 4 = 2.9

BST 2



Avg. no. of successful comparison
0.1 . 2 + 0.2 . 1+ 0.4 . 2 + 0.3 . 3= 2.1.

OPTIMAL BINARY SEARCH TREES

- Total number of binary search trees with *n* keys is equal to the *n*th ***Catalan number.***

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1.$$

which grows to infinity as fast as $4^n/n^{1.5}$

$$C(n,r) = \binom{n}{r} = \frac{n!}{(r!(n-r)!)}$$

OPTIMAL BINARY SEARCH TREES

Alternate approach to find number of BST possible with n nodes

$$t(n) = \sum_{i=1}^{n} t(i-1)\ t(n-i).$$

- The base case is t(0) = 1 and t(1) = 1, i.e. there is one empty BST and there is one BST with one

$$t(2) = t(0)t(1) + t(1)t(0) = 2$$

$$t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 2 + 1 + 2 = 5$$

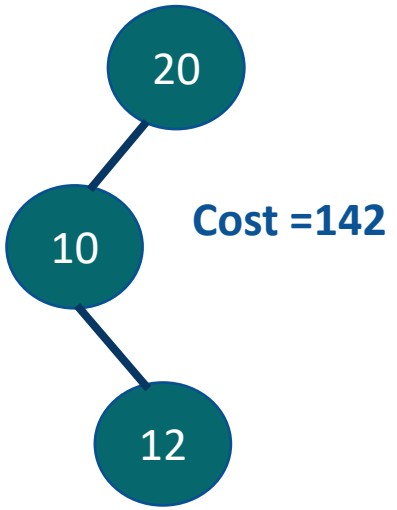
$$t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 5 + 2 + 2 + 5 = 14$$

OPTIMAL BINARY SEARCH TREES

Definitions

- An Optimal Binary Search Tree **(OBST)** is a Binary Search Tree **(BST)**, which has minimal expected cost of locating each node
- An **OBST** is a **BST** for which the nodes are arranged on levels such that the tree cost is minimum
- Example

  Keys={10, 12, 20} Frequency=(34, 8, 50}
- For n = 0, 1, 2, 3, … values of Catalan numbers are
  - 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ….
  - So are numbers of Binary Search Trees.



OPTIMAL BINARY SEARCH TREES

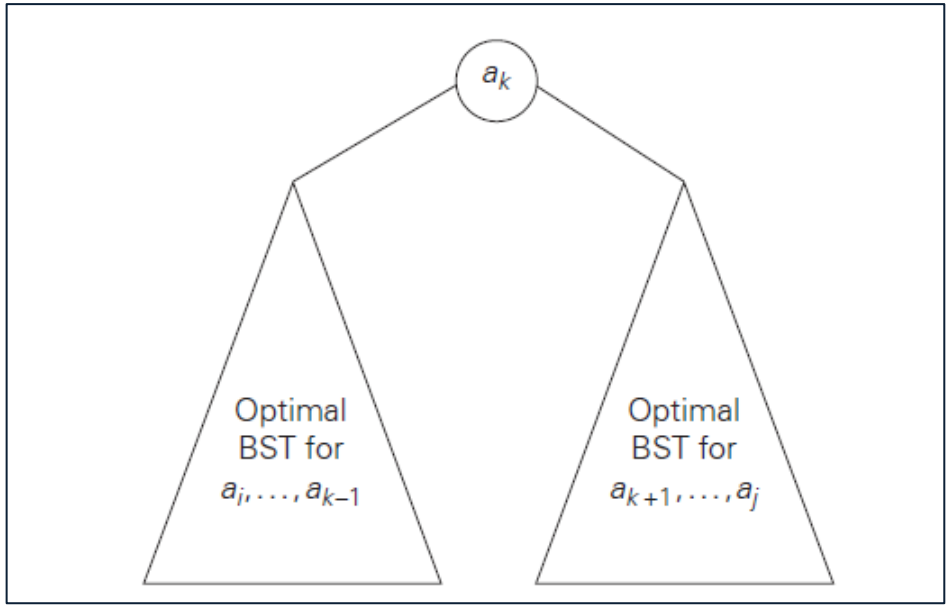
Dynamic Programming Approach

- let $a_1, \ldots, a_n$ be distinct keys ordered from the smallest to the largest
- let $p_1, \ldots, p_n$ be the probabilities of searching for them
- Let $C(i, j)$ be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys $a_i, \ldots, a_j$ where i, j are some integer indices, $1 \le i \le j \le n$.
  - To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$.

OPTIMAL BINARY SEARCH TREES

Dynamic Programming Approach

- For such a binary search tree (shown below) the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, the right subtree T_{k+1}^{j} contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged



OPTIMAL BINARY SEARCH TREES

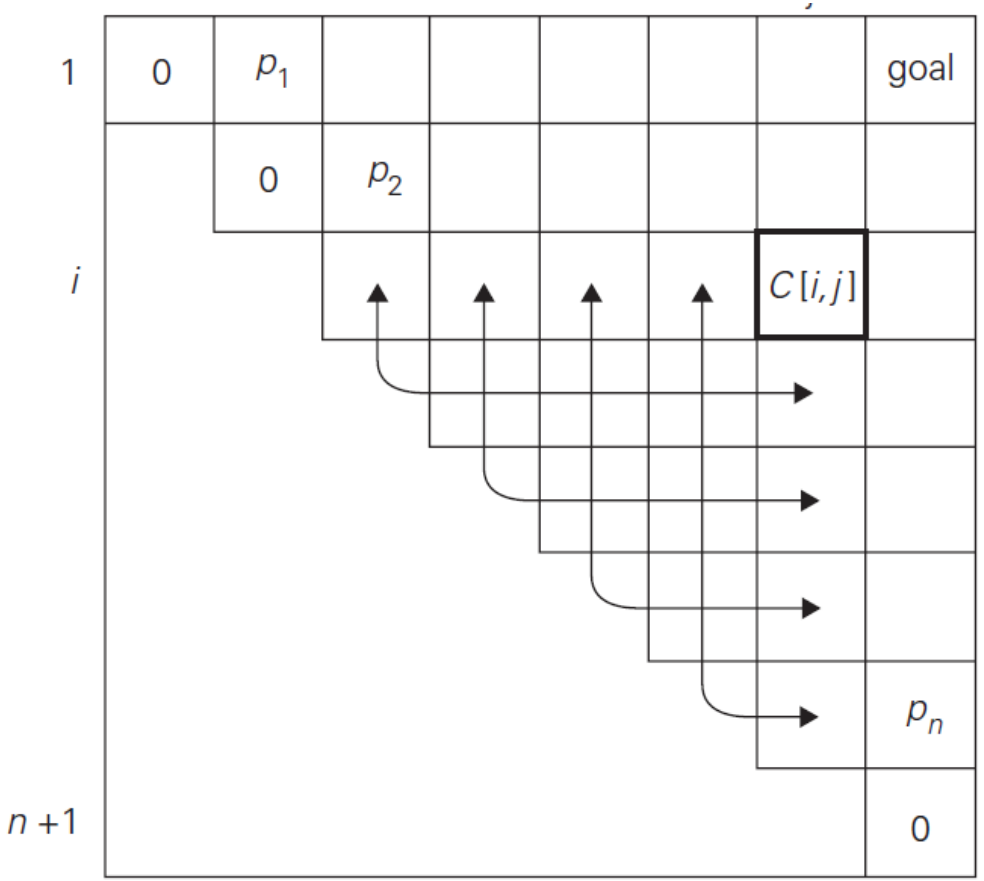
- The recurrence for solving the problem of finding OBST is

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

- Assumptions
  - *C(i, i – 1)* = 0 for 1≤ *i* ≤ *n* + 1, *(*number of comparisons in the empty tree)
  - *C(i, i) = pi* for 1≤ *i* ≤ *n, (BST with one node* a_i*)*

OPTIMAL BINARY SEARCH TREES

- Table structure
-



OPTIMAL BINARY SEARCH TREES

- Obtain the optimal binary search tree for the following given keys and their probabilities
-

Keys	Probability
A	0.1
B	0.2
C	0.4
D	0.3

OPTIMAL BINARY SEARCH TREES

- Solution

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

C(i, i – 1) **= 0 for 1≤** ***i*** **≤** ***n*** **+ 1**
C(i, i) **=** ***pi*** **for 1≤** ***i*** **≤** ***n,***

R(i, i) **=** ***i*** **for 1≤** ***i*** **≤** ***n***

OPTIMAL BINARY SEARCH TREES

- Solution is elaborated in the lecture notes

TRAVELING SALESMAN PROBLEM

- Jg

TRAVELING SALESMAN PROBLEM

DYNAMIC PROGRAMMING

Dr.Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

THANK YOU