# Chapter 3.

# Dynamic Programming

Foundations of Algorithms, 5<sup>th</sup> Ed. Richard E. Neapolitan

#### Contents

- 3.1 The Binomial Coefficient
- 3.2 Floyd's Algorithm for Shortest Paths
- 3.3 Dynamic Programming and Optimization Problem
- 3.4 Chained Matrix Multiplication
- 3.5 Optimal Binary Search Trees
- 3.6 The Traveling Salesperson Problem





# **Dynamic Programming**

- Dynamic Programming
  - is similar to divide-and-conquer in that
    - an instance of a problem is divided into smaller instances.
  - However, D.P. solves small instances first,
    - store the results, and later, whenever we need a result,
      - look it up instead of recomputing it.
  - The term "dynamic programming" comes from control theory,
    - "programming" means the use of an array (table)
      - in which a solution is constructed. (called *memoization*)
  - Dynamic Programming is a *bottom-up* approach,
    - whereas the divide-and-conquer is a *top-down* approach.





# **Dynamic Programming**

- The steps in the design of Dynamic Programming:
  - 1. Establish a recursive property that
    - gives the solution to an instance of the problem.
  - 2. Solve an instance of the problem in a bottom-up fashion
    - by solving smaller instances first (using *memoization*).



- The Binomial Coefficient Problem:
  - The definition of binomial coefficient is given by:

$${\binom{n}{k}} = \frac{n!}{k!(n-k)!}, \text{ for } 0 \le k \le n.$$

- We cannot compute directly because n! is very large even for small n.
- Using the recursive property of the binomial coefficient:

$$-\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

- We can eliminate the need to compute n! or k!.



#### **ALGORITHM 3.1**: Binomial Coefficient Using Divide-and-Conquer

```
int bin(int n, int k) {
    if (k == 0 || n == k)
        return 1;
    else
        return bin(n - 1, k - 1) + bin(n - 1, k);
}
```



- The Inefficiency of Algorithm 3.1:
  - Algorithm 3.1 computes  $2 \binom{n}{k} 1$  terms to determine  $\binom{n}{k}$ . (Exercise 3.1.2)
  - The problem is that
    - the same instances are solved in each recursive call.
  - Recall that the *divide-and-conquer* approach is always *inefficient* 
    - when an instance is divided into two smaller instances
      - that are *almost as large as* the original instance.



```
Top-Down
            bin(4, 2)
                    bin(3, 1)
                            bin(2, 0)
                            bin(2, 1)
                    bin(3, 2)
                             bin(2, 2)
                                                      Bottom-Up
```

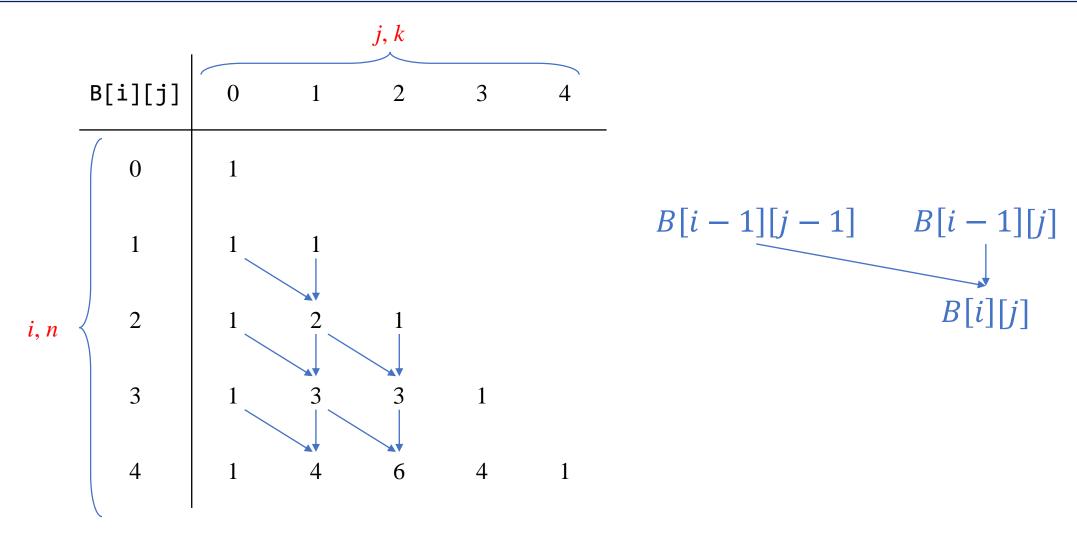


- Design a more efficient algorithm using D.P.
  - *Establish* a recursive property:
    - Let B be an array that B[i][j] contains  $\binom{n}{k}$ .

$$-B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ or } j = i \end{cases}$$

- Solve an instance of the problem in a bottom-up fashion
  - by *computing the rows* in *B* in sequence
    - *starting* with the *first* row.





• You may recognize the array in this figure as *Pascal's triangle*.





#### **ALGORITHM 3.1**: Binomial Coefficient Using Divide-and-Conquer

```
int bin2(int n, int k) {
    int i, j;
    int B[n + 1][k + 1];
    for (i = 0; i <= n; i++)
        for (j = 0; j \leftarrow minimum(i, k); j++)
            if (j == 0 || j == i)
                 B[i][j] = 1;
            else
                 B[i][j] = B[i - 1][j - 1] + B[i - 1][j];
    return B[n][k];
```



- Time Complexity of Algorithm 3.2:
  - The parameters *n* and *k* are *not* the *size* of *input* to this algorithm.
    - Rather, they are the input, and the input size is
      - the *number of symbols* it takes to *encode* them.
  - For given n and k,
    - the number of passes for each value of *i* are:

i	0	1	2	3	 k	k + 1	 n
Number of passes	1	2	3	4	 k + 1	k + 1	 k + 1



- Time Complexity of Algorithm 3.2:
  - The total number of passes is:

$$-1 + 2 + 3 + 4 + \dots + k + (k+1) + (k+1) \dots + (k+1)$$

$$- = \frac{k(k+1)}{2} + (n-k+1)(k+1)$$

$$- = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk).$$

- We can argue that we have developed a *much more efficient* algorithm,
  - by using *dynamic programming* instead of *divide-and-conquer*.



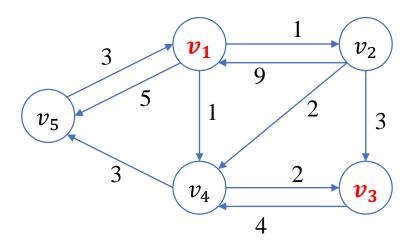
- Improving the Performance of Algorithm 3.2
  - Take advantage of the fact that

$$-\binom{n}{k} = \binom{n}{n-k}.$$

- Use only a *one-dimensional array* indexed from 0 to k,
  - instead of creating the *entire two-dimensional* array *B*.
  - Once a row is computed,
    - we no longer need the values in the row that proceeds it.



- Finding the Shortest Paths in a Graph:
  - from each vertex to all other vertices (All Pairs Shortest Paths)



- Shortest path from  $v_1$  to  $v_3$ ?
  - $length[v_1, v_2, v_3] = 1 + 3 = 4$
  - $length[v_1, v_4, v_3] = 1 + 2 = 3$
  - $length[v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$



- The Shortest Paths as an *Optimization Problem*:
  - There can be more than one *candidate solution* 
    - to an instance of an optimization problem.
  - Each candidate solution has a value associated with it,
    - and a solution to the instance is any candidate solution
      - that has an optimal value (either minimum or maximum).
  - There can be more than one shortest paths from one vertex to another.
    - Our problem is to find *any one* of the shortest paths.



- The **Brute-Force** Approach:
  - Determine, for each vertex,
    - the *lengths of all the paths* from that vertex to each other vertex,
      - and compute the *minimum* of these lengths.
  - This algorithm is *worse* than *exponential time*.
    - The total number of paths from one vertex to another vertex is
      - $(n-2)(n-3)\cdots 1 = (n-2)!$  (factorial time)
  - This is often the case with many optimization problems.
    - Our goal is to find a more efficient algorithm for this problem.



#### • Floyd's Algorithm:

- A *cubic-time* algorithm for the problem of (*All Pairs*) *Shortest Paths*.
  - not *exponential*, but *polynomial*.
- Using dynamic programming,
  - First, we develop an algorithm
    - that determines *only the lengths* of the shortest paths.
  - After, we modify it to *produce shortest paths* as well.





- Data Structure for the Floyd's Algorithm
  - We represent a *weighted* (*directed*) graph containing *n* vertices
    - by an array (*adjacency matrix*) W where

• 
$$W[i][j] = \begin{cases} weight\ on\ edge \end{cases}$$
 if there is an edge from  $v_i$  to  $v_j$  if there is no edge from  $v_i$  to  $v_j$  or  $if\ i=j$ 

- We also define an array *D* as a matrix that
  - contains the lengths of the shortest paths in the graph.

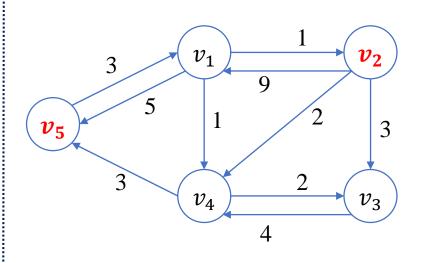


W	1	2	3	4	5	D	1	2	3	4	5
1	0	1	$\infty$	1	5	1	0	1	3	1	4
		0				2	8	0	3	2	5
3	$\infty$	$\infty$	0	4	$\infty$	3	10	11	0	4	7
4	$\infty$	$\infty$	2	0	3	4	6	7	2	0	3
5	3	$\infty$	$\infty$	$\infty$	0	5	3	4	6	4	0



- Solving the Problem:
  - If we can develop a way
    - to calculate the values in D from those in W,
      - we will have an algorithm for the Shortest Paths problem.
  - Let us create a sequence of n+1 arrays  $D^{(k)}$ ,
    - where  $0 \le k \le n$  and where
      - $D^{(k)}[i][j] = \text{length of a shortest path from } v_i \text{ to } v_j \text{ using only vertices}$ in the set  $\{v_1, v_2, \dots, v_k\}$  as intermediate vertices.





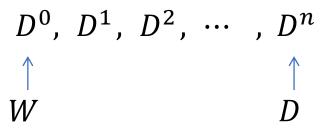
- $D^{(0)}[2][5] = length[v_2, v_5] = \infty$
- $D^{(1)}[2][5] = minimum(length[v_2, v_5], length[v_2, v_1, v_5])$  $= minimum(\infty, 14) = 14.$
- $D^{(2)}[2][5] = D^{(1)}[2][5] = 14.$
- $D^{(3)}[2][5] = D^{(2)}[2][5] = 14.$
- $D^{(4)}[2][5] = minimum(length[v_2, v_1, v_5], length[v_2, v_4, v_5],$  $length[v_2, v_1, v_4, v_5], length[v_2, v_3, v_4, v_5]$ = minimum(14,5,13,10) = 5.
- $D^{(5)}[2][5] = D^{(4)}[2][5] = 5.$



- Solving the Problem:
  - Note that  $D^{(n)}[i][j]$  is the length of a shortest path from  $v_i$  to  $v_j$ ,
    - because it is the length of a shortest path from  $v_i$  to  $v_j$ 
      - that is allowed to pass through any of the other vertices.
  - Therefore, we have established that
    - $D^{(0)} = W \text{ and } D^{(n)} = D.$
  - To determine *D* from *W*,
    - we need only find a way to obtain  $D^{(n)}$  from  $D^{(0)}$ .



- The steps for Dynamic Programming:
  - Establish a recursive property
    - with which we compute  $D^{(k)}$  from  $D^{(k-1)}$ .
  - Solve an instance of the problem in a bottom-up fashion
    - by repeating the process for k = 1 to n.
    - This process creates the sequence:

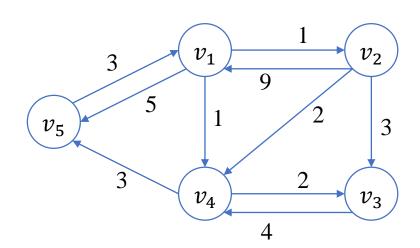




- Two cases of Establishing Recursive Property:
  - 1. At least one shortest path from  $v_i$  to  $v_i$  does not use  $v_k$ ,
    - using only vertices in  $\{v_1, v_2, \dots, v_k\}$  as intermediate vertices.
    - Then,  $D^{(k)}[i][j] = D^{(k-1)}[i][j]$ .
  - 2. All shortest paths from  $v_i$  to  $v_i$  do use  $v_k$ ,
    - using only vertices in  $\{v_1, v_2, \dots, v_k\}$  as intermediate vertices.
    - Then,  $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$ .

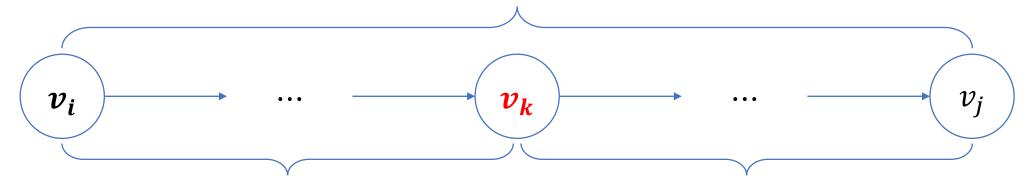


- Examples of Two Cases:
  - case 1:  $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$ ,
    - because when we include vertex  $v_5$ ,
      - the shortest path from  $v_1$  to  $v_3$  is still  $[v_1, v_4, v_3]$ .
  - case 2:  $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$ ,
    - because  $v_k$  cannot be an intermediate vertex on the subpath from  $v_i$  to  $v_k$ ,
      - that subpath uses only vertices in  $\{v_1, v_2, \dots, v_{k-1}\}$  as intermediates.





A shortest path from  $v_i$  to  $v_j$  using only vertices in  $\{v_1, v_2, \dots, v_k\}$ 



A shortest path from  $v_i$  to  $v_k$ using only vertices in  $\{v_1, v_2, \cdots, v_k\}$ 

A shortest path from  $v_k$  to  $v_i$ using only vertices in  $\{v_1, v_2, \cdots, v_k\}$ 



- Considering Two Cases:
  - Let  $D^{(k)}[i][j]$  be the *minimum* of the values from either Case 1 or Case 2.
    - $D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]).$
  - Now we have established a recursive property (Step 1).
    - To accomplish Step 2,
      - create the sequence of arrays:  $D^0$ ,  $D^1$ ,  $D^2$ , ...,  $D^n$ .



$$D^0$$
 1
 2
 3
 4
 5

 1
 0
 1
  $\infty$ 
 1
 5

 2
 9
 0
 3
 2
  $\infty$ 

 3
  $\infty$ 
 $\infty$ 
 0
 4
  $\infty$ 

 4
  $\infty$ 
 $\infty$ 
 2
 0
 3

 5
 3
  $\infty$ 
 $\infty$ 
 $\infty$ 
 0

- $D^{(1)}[2][4] = minimum(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$ = minimum(2, 9 + 1) = 2.
- $D^{(1)}[5][2] = minimum(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$ =  $minimum(\infty, 3 + 1) = 4.$
- $D^{(1)}[5][4] = minimum(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$ =  $minimum(\infty, 3 + 1) = 4.$
- $D^{(2)}[5][4] = minimum(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$ = minimum(4, 4 + 2) = 4.



#### **ALGORITHM 3.3**: Floyd's Algorithm for Shortest Paths

```
void floyd(int n, const int W[][MAX], int D[][MAX]) {
    int i, j, k;
    D = W;
    for (k = 1; k \le n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
```





- Why can we use only one array *D*?
  - because the *values* in the *k*th row and the *k*th column
    - are *not changed* during the *k*th iteration of the loop.
  - In the *k*th iteration, the algorithm assigns
    - D[i][k] = minimum(D[i][k], D[i][k] + D[k][k]), and
    - D[k][j] = minimum(D[k][j], D[k][k] + D[k][j])
  - During the kth iteration, D[i][j] is computed
    - from only its own value and values in the kth row and the kth column.
    - These values have maintained their values from the (k-1)st iteration.
- Time Complexity of Floyd's algorithm:  $T(n) = n^3 \in \Theta(n^3)$ .



- Producing the Shortest Paths:
  - Define an array *P*, where

$$P[i][j] = \begin{cases} \text{highest index of an itermediate vertex on the shortest path} \\ \text{from } v_i \text{ to } v_j, \text{ if at least one intermediate vertex exists.} \\ 0, \text{ if no intermediate vertex exists.} \end{cases}$$

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





#### **ALGORITHM 3.4:** Floyd's Algorithm for Shortest Paths 2

```
void floyd2(int n, const int W[][MAX], int D[][MAX], int P[][MAX]) {
    int i, j, k;
    for (i = 1; i <= n; i++)
        for (j = 1; j \le n; j++)
            P[i][i] = 0;
    D = W;
    for (k = 1; k \le n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (D[i][k] + D[k][j] < D[i][j]) {</pre>
                    P[i][j] = k;
                    D[i][j] = D[i][k] + D[k][j];
```



#### **ALGORITHM 3.5**: Print Shortest Path

```
void path(int q, int r) {
    if (P[q][r] != -1) {
        path(q, P[q][r]);
        cout << "v" << P[q][r] << " ";
        path(P[q][r], r);
```





# 3.3 Dynamic Programming and Optimization Problems

- The design of a D.P. algorithm for an *optimization problem*:
  - 1. Establish a recursive property
    - that gives the optimal solution to an instance of the problem.
  - 2. Compute the value of an optimal solution in a bottom-up fashion.
  - 3. Construct an optimal solution in a bottom-up fashion.
  - Note that *Steps 2* and *3* are ordinarily accomplished
    - at *about the same point* in the algorithm





# 3.3 Dynamic Programming and Optimization Problems

#### • The Principle of Optimality:

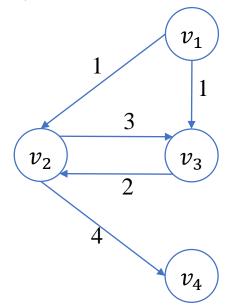
- An optimization problem can be solved using dynamic programming,
  - if and only if the *principle of optimality applies* in the problem.
- The principle of optimality is said to *apply* in a problem
  - if an *optimal solution* to an instance of a problem
    - always contains optimal solutions to all subinstances.





## 3.3 Dynamic Programming and Optimization Problems

- Example: the *Longest Paths* problem
  - Finding the *longest simple paths* from each vertex to all other vertices.
  - We restrict the path to be *simple*,
    - because *a cycle* can always create an arbitrarily long path
      - by repeatedly passing through the cycle.
  - Then, we can show that the principle of optimality *does not apply*.



- The optimal (longest) simple path from  $v_1$  to  $v_4$ is  $[v_1, v_3, v_2, v_4]$ .
- However, the subpath  $[v_1, v_3]$  is not an optimal path from  $v_1$  to  $v_4$ 
  - length  $[v_1, v_3] = 1$  and length  $[v_1, v_2, v_3] = 4$ .



- The Multiplication of Chained Matrices:
  - Suppose we want to multiply a  $2 \times 3$  matrix and a  $3 \times 4$  matrix:

$$-\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

- The resultant matrix is a  $2 \times 4$  matrix.
- If we use the standard method of multiplying matrices
  - it takes three elementary multiplications.
  - For 29 in the product: 3 multiplications  $(1 \times 7 + 2 \times 2 + 3 \times 6)$ .
- Because there  $2 \times 4 = 8$  entries in the product,
  - the total number of elementary multiplication is  $2 \times 3 \times 4 = 24$ .



- The number of *elementary multiplications*:
  - In general, to multiply an  $i \times j$  matrix with a  $j \times k$  matrix,
    - the number of elementary multiplications is  $i \times j \times k$ .
  - Consider the multiplication of the following four matrices:

```
- A \times B \times C \times D
-(20 \times 2) (2 \times 30) (30 \times 12) (12 \times 8)
```

- Matrix multiplication is an associative operation,
  - meaning the *order* in which we multiply *does not matter*.
  - for example, A(B(CD)) = (AB)(CD).



- The number of elementary multiplications:
  - There are *five different orders* with different number multiplications.

```
- A(B(CD)): 3,680
```

-(AB)(CD): 8,880

- A((BC)D): 1,232 (optimal order)

-((AB)C)D: 10,320

-(A(BC))D: 3,120

- *Our goal* is to develop an algorithm that
  - determines the *optimal order* for multiplying *n* chained matrices.



- The Brute-Force Approach:
  - Consider *all possible orders* and *take* the *minimum*.
  - We will show that this algorithm is at least exponential-time.
    - For a given *n* matrices:  $A_1, A_2, \dots, A_n$ ,
    - let  $t_n$  be the number of different orders.
    - Then,  $t_n \ge t_{n-1} + t_{n-1} = 2t_{n-1}$ , and  $t_2 = 1$ .
    - Therefore,  $t_n \ge 2^{n-2}$ . (at least exponential)



- Does the principle of optimality apply?
  - Show that the optimal order for multiplying *n* matrices includes
    - the optimal order for multiplying any subset of the *n* matrices.
  - Then, we can use dynamic programming to construct a solution.

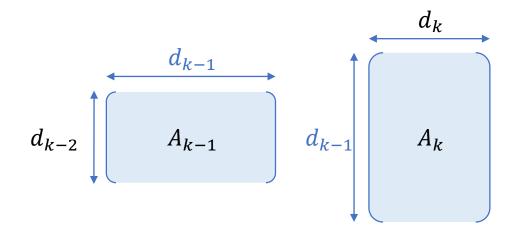
$$A_1\left(\left(\left((A_2A_3)A_4\right)A_5\right)A_6\right)$$

- If this order is *optimal* for multiplying  $A_1A_2A_3A_4A_5A_6$ ,
- then the *optimal order* for multiplying  $A_2A_3A_4$  is this.

$$\left( (A_2 A_3) A_4 \right)$$



- Multiplying  $A_{k-1}$  times  $A_k$ :
  - The number of columns in  $A_{k-1}$ 
    - must equal to the number of rows in  $A_k$ .
  - If we let  $d_0$  be the number of rows in  $A_1$ 
    - and  $d_k$  be the number of columns in  $A_k$  for  $1 \le k \le n$ ,
    - then, the dimension of  $A_k$  is  $d_{k-1} \times d_k$ .





- Creating a sequence of arrays:
  - For  $1 \le i \le j \le n$ , let

$$\mathbf{M}[i][j] = \begin{cases} \text{minimum number of multiplications needed to multiply} \\ A_i \text{ through } A_j, \text{ if } i < j. \\ 0, \text{ if } i = j. \end{cases}$$

$$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$$
  
(5 × 2) (2 × 3) (3 × 4) (4 × 6) (6 × 7) (7 × 8)  
 $d_0 \ d_1 \ d_2 \ d_3 \ d_4 \ d_5 \ d_6$ 

- To multiply  $A_4$ ,  $A_5$ , and  $A_6$ , we have two orders:
  - $(A_4A_5)A_6$ :  $d_3d_4d_5 + d_3d_5d_6 = 392$
  - $A_4(A_5A_6)$ :  $d_4d_5d_6 + d_3d_4d_6 = 528$
- Therefore, M[4][6] = minimum(392, 528) = 392.

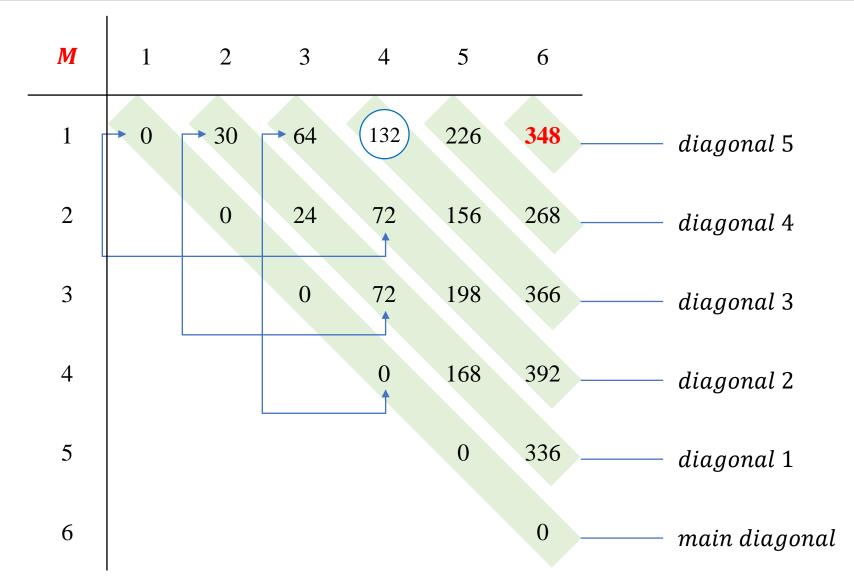


- Establish the recursive property:
  - The *optimal order* for multiplying *six* matrices
    - must have *one of these factorizations*:
    - 1.  $(A_1)(A_2A_3A_4A_5A_6)$
    - $(A_1A_2)(A_3A_4A_5A_6)$
    - 3.  $(A_1A_2A_3)(A_4A_5A_6)$
    - 4.  $(A_1A_2A_3A_4)(A_5A_6)$
    - 5.  $(A_1A_2A_3A_4A_5)(A_6)$
  - Of these *factorizations*,
    - the one that *yields* the *minimum* number of multiplications
      - must be the optimal one.



- Establish the recursive property:
  - The number of multiplications for the *k*th factorization
    - is the *minimum number* needed to *obtain each factor* 
      - *plus* the *number* needed to *multiply two factors*.
    - that is,  $M[1][k] + M[k+1][6] + d_0d_kd_6$ .
  - We have established that
    - $M[1][6] = \min_{1 \le k \le 5} (M[1][k] + M[k+1][6] + d_0 d_k d_6).$
  - We can generalize this result into:
    - $M[i][j] = \min_{i \le k \le j-1} (M[i][k] + M[k+1][j] + d_{i-1}d_kd_j)$ , if i < j.
    - -M[i][i] = 0.









```
M[i][i] = 0 for 1 \le i \le 6
M[1][2] = \min_{1 \le k \le 1} (M[1][k] + M[k+1][2] + d_0 d_k d_2)
           = M[1][1] + M[2][2] + d_0d_1d_2 = 0 + 0 + 5 \times 2 \times 3 = 30.
M[1][3] = \min_{1 \le k \le 2} (M[1][k] + M[k+1][3] + d_0 d_k d_3)
           = minimum(M[1][1] + M[2][3] + d_0d_1d_3, M[1][2] + M[3][3] + d_0d_2d_3)
           = minimum(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64.
M[1][4] = \min_{1 \le k \le 3} (M[1][k] + M[k+1][4] + d_0 d_k d_4)
           = minimum(M[1][1] + M[2][4] + d_0d_1d_4, M[1][2] + M[3][4] + d_0d_2d_4, M[1][3] + M[4][4] + d_0d_3d_4)
           = minimum(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, 64 + 0 + 5 \times 4 \times 6) = 132.
M[1][6] = 348.
```



#### **ALGORITHM 3.6**: Minimum Multiplications

```
int minmult(int n, int d[], int P[][MAX], int M[][MAX]) {
    int i, j, k, diagonal;
    for (i = 1; i <= n; i++)
        M[i][i] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)</pre>
        for (i = 1; i <= n - diagonal; i++) {
            j = i + diagonal;
            M[i][j] = \min_{i < k < i-1} (M[i][k] + M[k+1][j] + d_{i-1}d_kd_j);
             P[i][j] = a value of k that gave the minimum;
    return M[1][n];
```





```
int minimum(int i, int j, int &minK, int M[][MAX], int d[MAX]) {
   int minV = INF;
   for (int k = i; k <= j - 1; k++) {
        int value = M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j];
       if (minV > value) {
           minV = value;
           minK = k;
   return minV;
```





- Time Complexity of Algorithm 3.6 (Every-Case)
  - Basic Operation: the *instructions* executed for each value of *k*.
  - Input Size: n, the number of matrices to be multiplied.
  - Since we have a nested loop,
    - the number of passes through the k loop is
      - j 1 i + 1 = i + diagonal 1 i + 1 = diagonal.
    - the total number of times that the basic operation is done equals

$$\sum_{\substack{diagonal=1}}^{n-1} \left[ (n - diagonal) \times diagonal \right] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$



• Obtaining the *Optimal Order* from the array P.

P	1	2	3	4	5	6	
1		1	1	1	1	1	-
2			2	3	4	5	
3				3	4	5	
4					4	5	
5						5	
6							

$$P[2][6] = 6: P[2][5] & P[6][6]$$

• 
$$P[2][5] = 4: P[2][4] \& P[5][5]$$

• 
$$P[2][4] = 3: P[2][3] & P[4][4]$$

• 
$$P[2][3] = 2: P[2][2] & P[3][3]$$

$$\bullet (A_1 A_2 A_3 A_4 A_5 A_6)$$

$$\bullet$$
( $A_1$ )( $A_2A_3A_4A_5A_6$ )

$$\bullet (A_1) ((A_2 A_3 A_4 A_5) A_6)$$

$$\bullet (A_1) \left( \left( (A_2 A_3 A_4) A_5 \right) A_6 \right)$$

$$\bullet (A_1) \left( \left( ((A_2 A_3) A_4) A_5 \right) A_6 \right)$$



#### **ALGORITHM 3.7**: Print Optimal Order

```
void order(int i, int j, int P[][MAX]) {
    if (i == j)
        cout << "A" << i;
    else {
        int k = P[i][j];
        cout << "(";
        order(i, k, P);
        order(k + 1, j, P);
        cout << ")";
```



- A binary search tree (BST) is
  - a *binary tree* of items (*keys*) that come from an *ordered* set, such that
  - 1. Each node contains *one key*.
  - 2. The keys in the *left subtree* of a given node
    - are *less than* or *equal to* the key in that node.
  - 3. The keys in the *right subtree* of a given node
    - are *greater than* or *equal to* the key in that node.





- **Optimal** Binary Search Tree
  - Our goal is to organize the keys in a BST so that
    - the *average time* it takes to locate a key is *minimized*.
  - Optimal BST is a tree that is organized in this fashion.
  - This problem is not hard
    - if all keys have the *same probability* of being the *search key*.
  - We are concerned with the case
    - where the keys do **not** have the same probability.
  - We will discuss the case in which
    - it is known that the *search key is* **in** *the tree*.



#### **ALGORITHM 3.8**: Search Binary Tree

```
void search(node pointer tree, int keyin, node pointer& p) {
    bool found;
    p = tree;
    found = false;
    while (!found) {
        if (p->key == keyin)
            found = true;
                                           typedef struct node *node pointer;
        else if (keyin < p->key)
                                           typedef struct node {
            p = p->left;
                                               int key;
        else
                                               node pointer left;
            p = p->right;
                                               node pointer right;
                                           } nodetype;
```

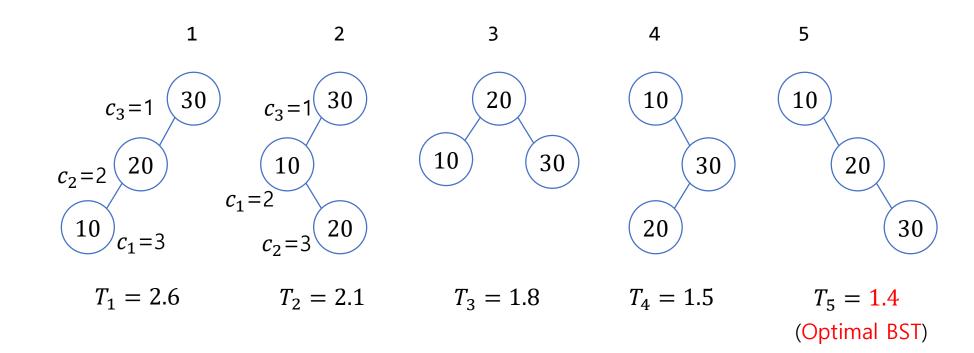




- Evaluating the search time of a BST
  - The search time is the number of comparisons
    - done by procedure *search* to locate a key.
  - Our goal is to determine a binary search tree
    - for which the *average search time* is *minimal*.
  - Let  $K_1, K_2, \dots, K_n$  be the *n* keys in order,
    - and let  $p_i$  be the *probability* that  $K_i$  is the *search key*.
  - The average search time is  $T_{avg} = \sum_{i=1}^{n} c_i p_i$ 
    - where  $c_i$  be the *number of comparisons* needed to find  $K_i$  in a given tree.



- An example:
  - n = 3, K = [10, 20, 30], p = [0.7, 0.2, 0.1]





- Finding an Optimal BST
  - by considering all binary search trees.
  - The number of *different* BSTs with a depth of n-1 is  $2^{n-1}$ .
    - If a BST's depth is n-1,
      - a node can be either to the left or to the right of its parent.
    - It means there are two possibilities at each of those levels.
  - The time complexity of this brute-force approach is *exponential*.

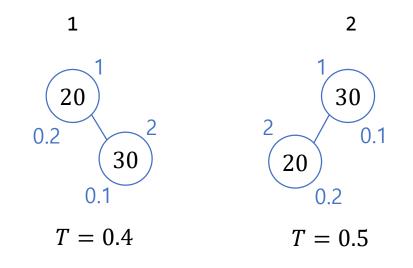




- To apply the Dynamic Programming
  - Suppose that keys  $K_i$  through  $K_i$  are arranged in a tree
    - that minimizes  $\sum_{m=1}^{J} c_m p_m$ .
  - We will call such a tree **optimal** for those keys
    - and denote the *optimal value* by A[i][j].
  - Because it takes one comparison to locate a key containing one key,
    - $A[i][i] = p_i$ .



- An example:
  - n = 3, K = [10, 20, 30], p = [0.7, 0.2, 0.1], then determine A[2][3].

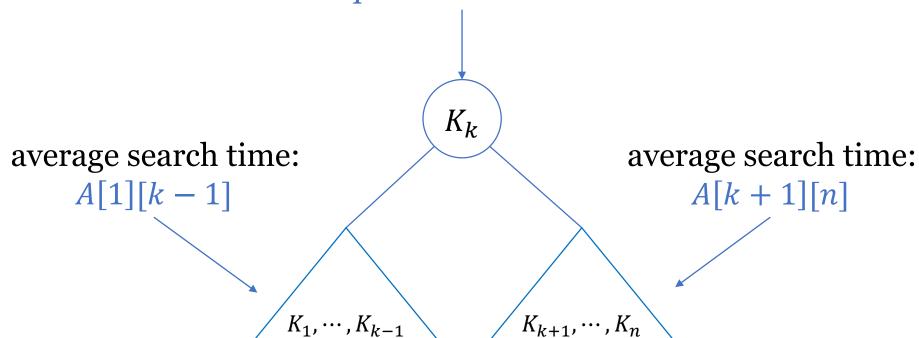


- A[2][3] = 0.4
  - 1.  $1 \times p_2 + 2 \times p_3 = 0.4$  (optimal)
  - 2.  $2 \times p_2 + 1 \times p_3 = 0.5$

- Note that the *optimal tree* is
  - the *right subtree* of the optimal tree obtained from the previous tree.
- The *principle of optimality* applies:
  - Any subtree of an optimal tree must be *optimal* for the key in that subtree.



For each key, there is *one additional* comparison at the root.





- Establish the recursive property:
  - The average search time for tree k is
    - $-A[1][k-1]+A[k+1][n]+\sum_{m=1}^{n}p_{m}$ .
  - The average search time for the *optimal tree* is given by
    - $A[1][n] = \min_{1 \le k \le n} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m,$ 
      - where A[1][0] and A[n+1][n] are defined to be 0.
  - In general,
    - $A[i][j] = minimum_{i \le k \le i} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m$ , for i < j.
    - $-A[i][j] = p_i,$
    - -A[i][i-1] = A[i+1][j] = 0.



- Determining an Optimal BST:
  - We proceed by computing *in sequence* the values on *each diagonal*.
    - because A[i][j] is computed
      - from entries in the *i*th row but to the left of A[i][j],
      - and from entries in the jth column but beneath A[i][j].
  - Let an array *R* contain
    - the indices of the keys chosen for the root at each step.
    - R[i][j]: the index of the key in the root of an optimal tree
      - containing the *i*th through the *j*th keys.



#### **ALGORITHM 3.9:** Optimal Binary Search Tree

```
void optsearchtree(int n, float p[], float &minavg, int R[][MAX]) {
    int i, j, k, diagonal;
    float A[MAX][MAX];
    for (i = 1; i <= n; i++) {
        A[i][i] = p[i]; A[i][i - 1] = 0;
        R[i][i] = i; R[i][i - 1] = 0;
    A[n + 1][n] = 0;
    R[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)</pre>
        for (i = 1; i <= n - diagonal; i++) {
             j = i + diagonal;
            A[i][j] = minimum_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m;
            R[i][j] = a value of k that gave the minimum;
    minavg = A[1][n];
```



#### **ALGORITHM 3.9**: Optimal Binary Search Tree

```
node_pointer tree(int i, int j, int R[][MAX], int keys[MAX]) {
   int k;
   node pointer p;
   k = R[i][j];
    if (k == 0)
        return NULL;
   else {
        p = (node pointer)malloc(sizeof(nodetype));
        p->key = keys[k];
        p->left = tree(i, k - 1, R, keys);
        p->right = tree(k + 1, j, R, keys);
        return p;
```

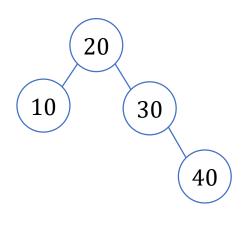




• An example:

• 
$$n = 4, K = [10, 20, 30, 40], p = [3, 3, 1, 1].$$

A	1	2	3	4	R	1	2	3	4
1	3			14	1	1	1	2	2
2		3	5	8	2		2	2	2
3			1	3	3			3	3
4				1	4				4



preorder: 20 10 30 40

inorder: 10 20 30 40





- Time Complexity of Algorithm 3.9:
  - Basic Operation: the instructions executed for *each value of k*.
    - This include a comparison to test for the minimum.
  - Input Size: *n*, the *number of keys*.
  - The control of this algorithm is almost identical to Algorithm 3.6.
    - The only difference is that,
      - the basic operation is done diagonal + 1 times.
  - Therefore,

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$



- The Traveling Salesperson Problem (TSP)
  - Suppose a *salesperson* is planning a *sales trip* that includes *20 cities*.
    - *Each city* is *connected* to some of the other cities by a road.
  - To *minimize* travel time,
    - we want to determine a *shortest route* that
      - *starts* at the salesperson's *home city*,
      - *visits* each of the cities *once*,
      - and ends up at the home city.





#### • Generalization of the TSP:

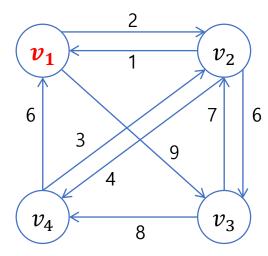
- In a weighted, directed graph,
  - find an *optimal tour* when at least one tour exists.
- A *tour* in a directed graph is a *path* from a *vertex* to *itself* 
  - that *passes through* each of the other vertices *exactly once*.
  - also called a *Hamiltonian circuit*.
- An *optimal tour* in a *weighted*, directed graph is
  - such a path of *minimum length*.





#### • An example:

- We will consider  $v_1$  to be the *starting vertex*.
- There are *three tours* and the lengths are:
  - $length[v_1, v_2, v_3, v_4, v_1] = 22$
  - $length[v_1, v_3, v_2, v_4, v_1] = 26$
  - $length[v_1, v_3, v_4, v_2, v_1] = 21$  (optimal tour)



- In general,
  - there can be an edge from *every vertex* to *every other vertex*.
  - the total number of tours is  $(n-1)(n-2)\cdots 1=(n-1)!$
  - Factorial time complexity: worse than exponential!



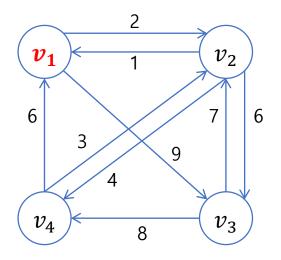
- The *principle of optimality* applies:
  - Notice that if  $v_k$  is the *first vertex* after  $v_1$  on an *optimal tour*,
    - the *subpath* of the tour from  $v_k$  to  $v_1$
    - must be a *shortest path* from  $v_k$  to  $v_1$
    - that passes through each of the other vertices *exactly once*.
  - Therefore, we can design an algorithm for the TSP
    - using the dynamic programming.





#### Formalization of the TSP:

- Let W be an *adjacency matrix* for a given graph G = (V, E).
- Let V be a set of all the vertices.
- Let A be a subset of V.
- Let  $D[v_i][A]$  be the length of a shortest path from  $v_i$  to  $v_1$ 
  - passing through each vertex in A exactly once.

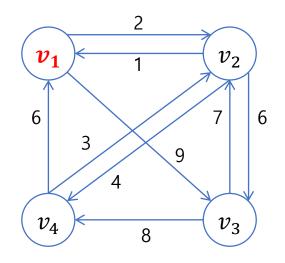


- $V = \{v_1, v_2, v_3, v_4\}$
- If  $A = \{v_3\}$ ,
  - $D[v_2][A] = lengh[v_2, v_3, v_1] = \infty.$
- If  $A = \{v_3, v_4\}$ ,
  - $D[v_2][A] = \min(lengh[v_2, v_3, v_4, v_1], lengh[v_2, v_4, v_3, v_1])$ =  $\min(20, \infty) = 20.$



- Establish the recursive property:
  - Because  $V \{v_1, v_i\}$  contains
    - all the vertices except  $v_1$  and  $v_i$  and the principle of optimality applies,
  - The length of an optimal tour is
    - $\min_{2 \le j \le n} (W[1][j] + D[v_j][V \{v_1, v_j\}]).$
  - In general, for  $i \neq 1$  and  $v_i \notin A$ ,
    - $D[v_i][A] = \min_{j:v_j \in A} (W[i][j] + D[v_j][A \{v_j\}])$ , if  $A \neq \phi$
    - $D[v_i][\phi] = W[i][1].$





$$A = \phi$$
: •  $D[v_2][\phi] = W[2][1] = 1$   
•  $D[v_3][\phi] = W[3][1] = \infty$   
•  $D[v_4][\phi] = W[4][1] = 6$ 

$$A = \{v_2\}: \quad \bullet \ D[v_3][A] = \min_{j:v_j \in A} (W[3][j] + D[v_j][A - \{v_j\}])$$
$$= W[3][2] + D[v_2][\phi] = 7 + 1 = 8$$



#### **ALGORITHM 3.11**: The Dynamic Programming Algorithm for the TSP

```
void travel(int n, int W[][MAX], int P[][MAX], int &minlength) {
    int i, j, k, D[MAX][MAX];
    for (i = 2; i <= n; i++)
         D[i][\phi] = W[i][1];
    for (k = 1; k \le n - 2; k++)
         for (all subsets A \subseteq V - \{v_1\} containing k vertices)
              for (i such that i \neq 1 and v_i is not in A) {
                  D[i][A] = \min_{j:v_j \in A} (W[i][j] + D[j][A - \{v_j\}]);
                  P[i][A] = value of j that gave the minimum;
    D[1][V - \{v_1\}] = \min_{2 \le j \le n} (W[1][j] + D[j][V - \{v_1, v_j\}]);
    P[1][V - \{v_1\}] = value of j that gave the minimum;
    minlength = D[1][V - \{v_1\}];
```



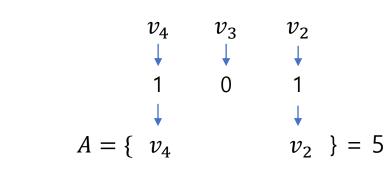


• Handling Subsets using Bitwise Operations:

 $2^{n-1}$ 

• All subsets  $A \subseteq V - \{v_1\}$ : 0, 1, 2, 3, 4, 5, 6, 7

$$A = \phi$$
: 000 = 0  
 $A = \{v_2\}$ : 001 = 1  
 $A = \{v_3\}$ : 010 = 2  
 $A = \{v_4\}$ : 100 = 4  
 $A = \{v_2, v_3\}$ : 011 = 3  
 $A = \{v_2, v_4\}$ : 101 = 5  
 $A = \{v_3, v_4\}$ : 110 = 6  
 $A = \{v_2, v_3, v_4\}$ : 111 = 7





- Handling Subsets using Bitwise Operations:
  - All subsets A containing k vertices

```
int subset_size = pow(2, n - 1);
for (k = 1; k \le n - 2; k++)
    for (int A = 0; A < subset_size; A++) {
        if (count(A) != k) continue;
                       int count(int A) {
                           int cnt = 0;
                           for (; A != 0; A >>= 1)
                               if (A & 1) cnt++;
                           return cnt;
```



- Handling Subsets using Bitwise Operations:
  - for i such that  $i \neq 1$  and  $v_i$  is not in A

```
for (i = 2; i <= n; i++) {
    if (isIn(i, A)) continue;
    .....
}</pre>
```

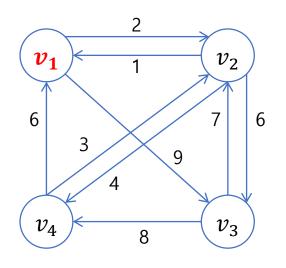
```
bool isIn(int i, int A) {
    return (A & (1 << (i - 2))) != 0;
}</pre>
```



- Handling Subsets using Bitwise Operations:
  - Minimum & Set Difference:  $A \{v_i\}$

```
int minimum(int n, int i, int &minJ, int A, int W[][MAX], int D[][MAX]) {
    int minV = INF;
    for (int j = 2; j <= n; j++) {
        if (!isIn(j, A)) continue;
        int value = W[i][j] + D[j][diff(A, j)];
        if (minV > value) {
            minV = value;
            minJ = j;
    return minV;
                              int diff(int A, int j) {
                                  return (A \& \sim (1 << (j - 2)));
```





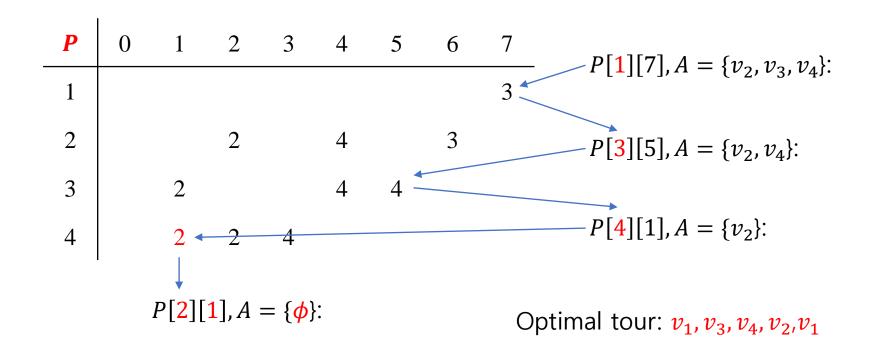
W	1	2	3	4
1	0	2	9	$\infty$
2	1	0	6	4
3	$\infty$	7	0	8
4	6	3	$\infty$	0

D	0	1	2	3	4	5	6	7
1								21
2	1		$\infty$		10		20	
3	$\infty$	8			14	12		
4	1 ∞ 6	4	$\infty$	$\infty$				

P	0	1	2	3	4	5	6	7
1	0 0 0							3
2	0		2		4		3	
3	0	2			4	4		
4	0	2	2	4				



- Retrieving an *optimal tour* from the array *P*:
  - P[i][A]: the index of the first vertex after  $v_i$ 
    - on a shortest path from  $v_i$  to  $v_1$
    - that passes through all vertices in *A* exactly once.







```
void tour(int v, int A, int P[][MAX]) {
    if (A == 1)
        cout << P[v][A] << " -> 1 " << endl;
    else {
        cout << P[v][A] << " -> ";
        tour(P[v][A], diff(A, P[v][A]), P);
                       cout << "optimal tour: 1 -> ";
                       tour(1, pow(2, n - 1) - 1, P);
```





- Time Complexity of Algorithm 3.11:
  - Basic Operation: the instructions executed for each value of  $v_i$ .
  - Input Size: *n*, the *number of vertices* in the graph.
  - The total number of times the basic operation is done is given by

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)k {n-1 \choose k}$$

$$= (n-1) \sum_{k=1}^{n-2} k {n-2 \choose k}$$

$$= (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$



- What we have gained:
  - Our algorithm is still  $\Theta(n^2 2^n)$  which is *exponential*.
  - When there are 20 cities and it takes 1  $\mu$ s to process 1 city,
    - Brute-force approach:  $(n-1)! = 19! \mu S = 3857 years$
    - Dyn. Prog.:  $(n-1)(n-2)2^{n-3} = 19 \times 18 \times 2^{17} \mu s = 45$  seconds
  - Can we solve the TSP in *polynomial* time?
    - Nobody solved it in polynomial time.
    - Nobody proved that it is impossible to solve it in polynomial time.

# Any Questions?

