Chapter 3
Dynamic Programming

- The term “dynamic programming” comes from control theory, and in this sense “programming” means the use of an array (table) in which a solution is constructed.

- $D_{ivide}A_{nd}C_{onquer}$ algorithm : the number of terms for determining fib(n) term is exponential in *n.*
- The reason is that the DAC approach top-down approach, where the smaller instances are unrelated
- To compute fib(5) we need to compute fib(4), and fib(3) theses terms are related in that they both require the fib(2)., it ends up computing the fib(2) more than once.
- The result is a very inefficient algorithm.

- Bottom-up approach to problem solving
- Instance of problem divided into smaller instances
- Smaller instances solved first and stored for later use.
- Look it up instead of re-compute
- Algorithm 1.7 for computing the *nth Fibonacci term* is an example of dynamic programming

Steps to develop a dynamic programming algorithm

1. Establish a recursive property that gives the solution to an instance of the problem
2. Compute the value of an optimal solution in a bottom-up fashion by solving smaller instances first

3.1 The Binomial Coefficient

For 0<=k<=n,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- For values of *n* and *k* that are not small, we cannot compute the binomial coefficient directly from this definition because *n* ! is very large.
- In the exercises we establish that

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

We can eliminate the need to compute *n! or k!*

Algorithm 3.1 Binomial Coefficient

- Divide-and-Conquer
- Recursive
- Very inefficient – like recursive Fibonocci
- To compute C(n,k), the algorithm computes

  2 C(n, k) -1 terms (**EXERCISE 2)**

  2 C(50, 10) -1 = 10,272,278,169

Algorithm 3.1

- **Binomial Coefficient Using Divide-and-Conquer**
- Problem: Compute the binomial coefficient.
- Inputs: nonnegative integers *n and k, where k ≤ n.*
- Outputs: *bin, the binomial coefficient*

Algorithm 3.1

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

For example, *bin (n − 1, k − 1)* and *bin (n − 1, k)* both need the result of *bin (n − 2, k − 1),* and this instance is solved separately in each recursive call.

Number of terms computed by recursive bin

- Algo. 3.1 uses Divide-and-Conquer
- Recursive
- Very inefficient
- In the exercises you will establish that the algorithm computes

$$2\binom{n}{k} - 1$$

terms

Dynamic Programming Solution to the Binomial Coefficient Problem

- Using the recursive property, construct an array B containing solutions to smaller instances

Construct Array B such that:

$$B[i,j] = \binom{i}{j}$$

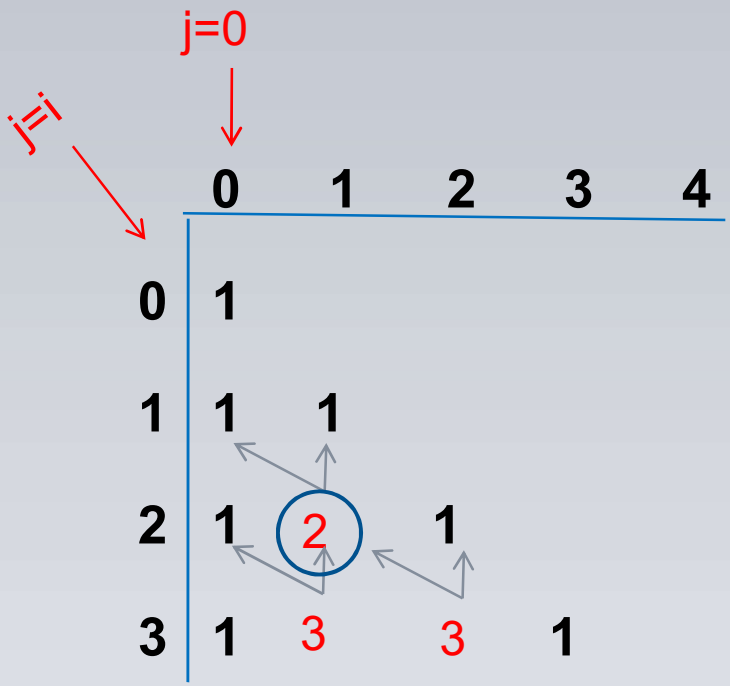
The steps for constructing a dynamic programming algorithm for this problem are as follows:

1. Establish a recursive property. This has already been done

```
B [i][j] =B [i-1][j-1] + B[i-1][j]    0<j<i
B [i][j] =1                           j=0 or j=i
```

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

- At each iteration, the values needed for that iteration have already been computed



$$B[i][j] = B[i-1][j-1] + B[i-1][j]$$

$$B[i][j] = 1 \qquad j=0 \text{ or } j=i$$

Algorithm 3.2

- Binomial Coefficient Using Dynamic Programming
- Problem: Compute the binomial coefficient.
- Inputs: nonnegative integers *n and k, where k ≤ n.*
- Outputs: *bin2, the binomial coefficient*

```
int bin2 (int n,  int k)
{
   index i,  j;
   int B[0..n][0..k];

   for (i = 0; i<= n ;  i++)
      for (j = 0;  j <= 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] ;
}
```

- The *parameters n* and *k* are not the size of the 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,* let's compute the number of passes through the ***for-j*** loop.
- The following table shows the number of passes for each value of i:

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

The total number of passes is therefore given by

$1 + 2 + 3 + 4 \ldots + k + (k+1 + k+1 + \ldots + k+1)$

$= k(k+1)/2 + (n-k+1)(k+1)$

$(k+1)/2[k+2n-2k+2] = (k+1)[2n-k+2]/2 \in \theta(nk)$

n-k+1 times

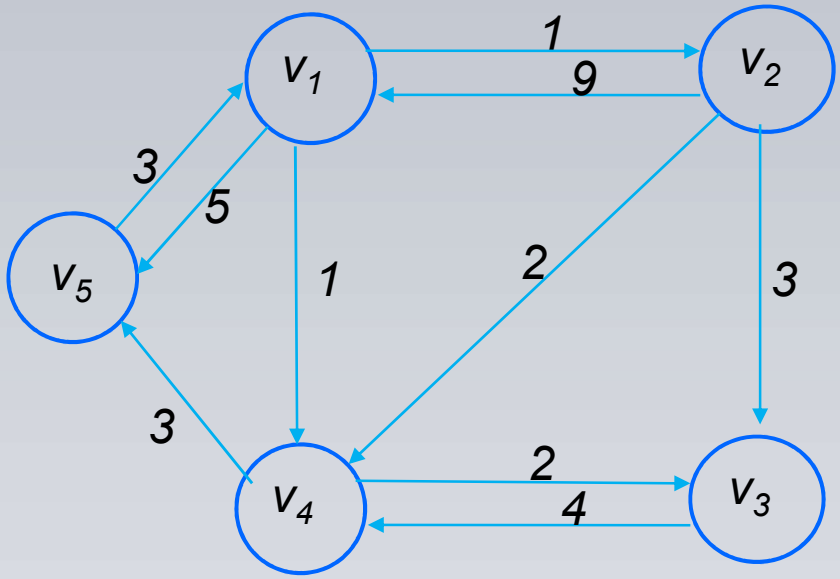
improvement

- once a row is computed, we no longer need the values in the row that precedes it. Therefore, the algorithm can be written using only a one-dimensional array indexed from 0 to *k*.
- *bin (n , k)* = *bin (n , n-k)*

3.2 Floyd's Algorithm for Shortest Paths

- A common problem encountered by air travelers is the determination of the shortest way to fly from one city to another when a direct flight does not exist.

3.2 Floyd's Algorithm for Shortest Paths



weighted, directed graph (***digraph***)

- circles :***vertices,***
- line : ***edge (arc)***
- **path :** a sequence of vertices such that there is an edge from each vertex to its successor. [*v1, v4, v3*]

- ***cycle :*** A path from a vertex to itself ***.***
- ***The*** path [*v1, v4, v5, v1*] is a cycle.
- If a graph contains a cycle, it is ***cyclic;*** otherwise, it is ***acyclic.***
- A path is called ***simple*** if it never passes through the same vertex twice.
- The ***length of a path*** in a weighted graph is the sum of the weights on the path; in an unweighted graph it is simply the number of edges in the path.

Floyd's Algorithm for Shortest Paths

- A problem that has many applications is finding the shortest paths from each vertex to all other vertices. Clearly, a shortest path must be a simple path. In Figure 3.2 there are three simple paths from *v*1 to *v*3—namely [*v*1, *v*2, *v*3], [*v*1, *v*4, *v*3], and [*v*1, *v*2, *v*4, *v*3]. Because [*v*1, *v*4, *v*3] is the shortest path from *v*1 to *v*3. As mentioned previously, one common application of shortest paths is determining the shortest routes between cities.

Optimization Problem

- The Shortest Paths problem is an ***optimization problem***
- Multiple candidate solutions
- Solution to the instance is a candidate solution with an optimal value
- Because there can be more than one shortest path from one vertex to another, our problem is to find **any** one of the shortest paths.

- Suppose there is an edge from every vertex to every other vertex.

- Because the second vertex on such a path can be any of *n − 2 vertices,* the third vertex on such a path can be any of *n − 3 vertices, … , and the second-to-last vertex on such* a path can be only one vertex, the total number of paths from one vertex to another vertex that pass through all the other vertices is

(n-2) (n-3) (n-4) 2 1=(n-2) !

which is worse than exponential

This algorithm is called Brute Force

Adjacency Matrix M

- W[i, j] = weight of the edge from $v_i \rightarrow v_j$
- W[i, j] = ∞ if there is no edge from $v_i \rightarrow v_j$
- W[i, j] = 0 if i = j
- v_i *is* ***adjacent to*** v_j if there is an edge from v_i ***to*** v_j ***,***

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge from } v_i \text{ to } v_j \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \\ 0 & \text{if } i = j. \end{cases}$$

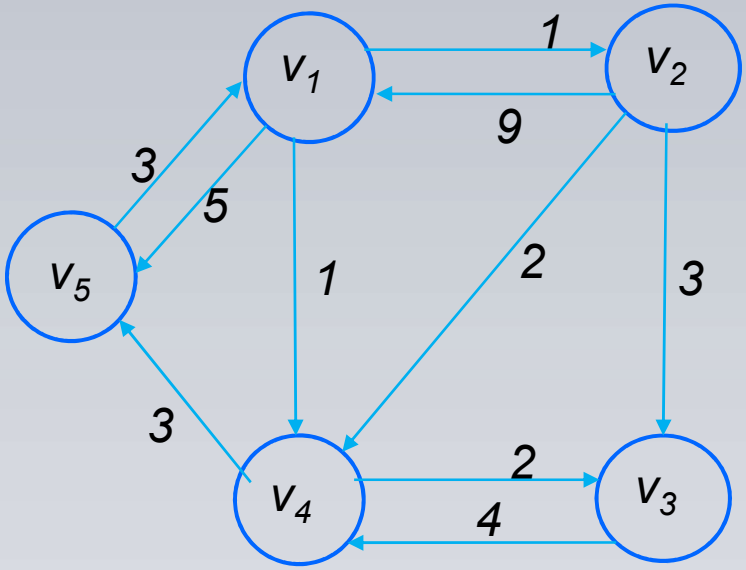
Figure 3.3

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

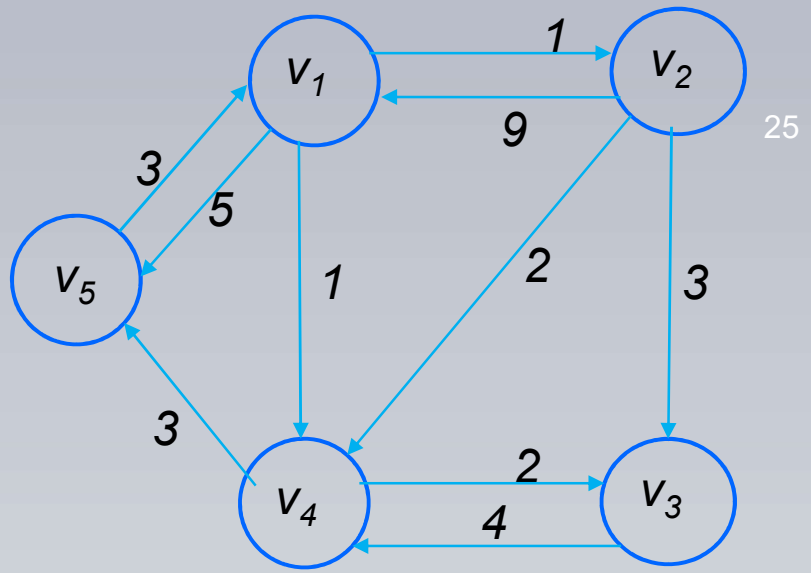
D



- The array D contains the lengths of the shortest paths in the graph.

- G: n vertices
- Create D^k where 0 <= k <=n
- D^k [i, j] = length of a shortest path from v_i to v_j using only vertices in the set {v_1 , v_2 , . . . v_k } as intermediate vertices
- D^n [i, j]= length of the shortest path from v_i to v_j that is allowed to pass through any of the other vertices.
- Because $D^{(0)}$[i, j] is the length of *a* shortest path that is not allowed to pass through any other vertices,
  - $D^0 = W$
  - $D^n = D$

Example 3.2

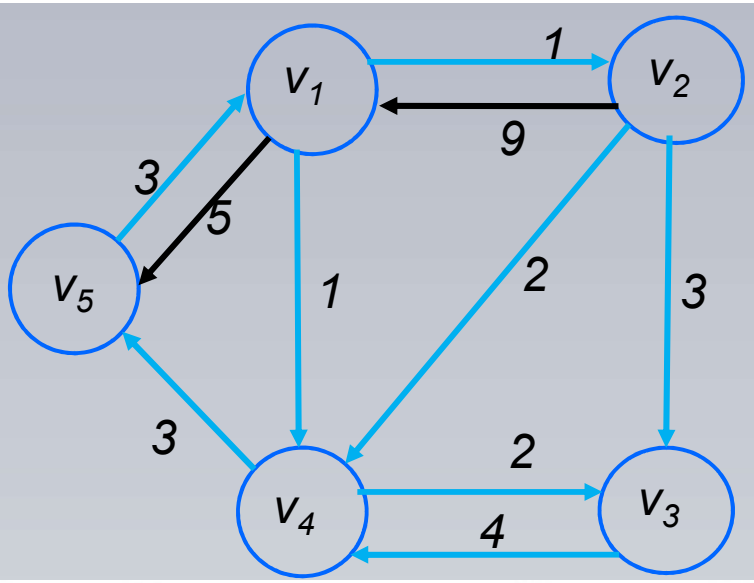


$$D^{(0)}[2][5] = length[v_2, v_5] = \infty.$$

$$D^{(1)}[2][5] = minimum(length[v_2, v_5], length[v_2, v_1, v_5])$$

$$= minimum(\infty, 14) = 14.$$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14.$ {For any graph these are equal because a}
{shortest path starting at v_2 cannot pass }
{through v_2.}



$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$. {For this graph these are equal because}
{including v_3 yields no new paths}
{from v_2 to v_5.}

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

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$. {For any graph these are equal because a}
{shortest path ending at v_5 cannot}
{pass through v_5.}

Dynamic Programming Steps

- To determine *D* from *W* we need only find a way to obtain D^n from D^0 . The steps

1. Compute D^k from $D^{(k-1)}$
2. Solve an instance of the problem in bottom-up fashion by repeating the process for k=1 to n This creates the sequence

$$D^0, D^1, D^2, \ldots D^n . \qquad (3.2)$$

D^0 = *W*, D^n = *D*

- Two Cases to consider
- D^k [i,j] = minimum (case 1, case 2)

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

Case 1

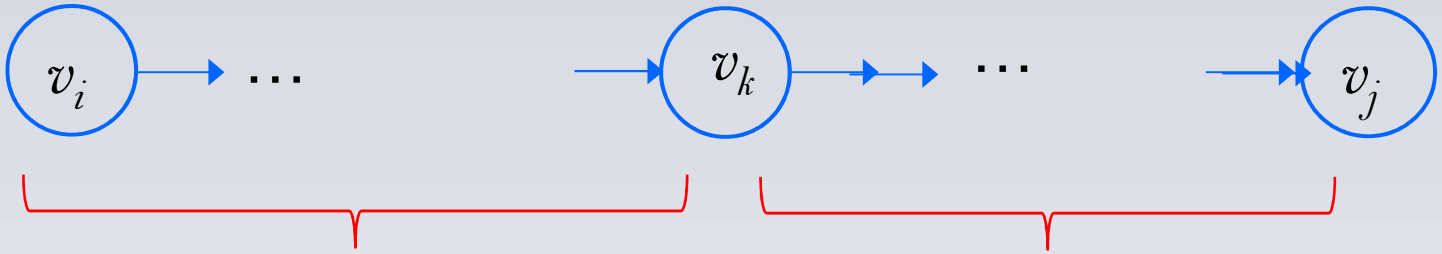
- At least one shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \ldots v_k\}$ as intermediate vertex does not use v_k

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

Case 2

- All shortest paths from v_i to v_j use v_k
- Path = **v_i , . . . , v_k , . . . , v_j**
  - where v_i , . . . , v_k consists only of vertices in {v_1 , v_2 , . . . v_{k-1} } as intermediates: Cost of path = $D^{(k-1)}$ [i, k]

  and where v_k , . . . , v_j consists only of vertices in {v_1 , v_2 , . . . v_{k-1} } as intermediates: Cost of path = $D^{(k-1)}$ [k, j]



shortest path from v_i to v_k using only vertices in {v_1 , v_2 , . . . v_{k-1} }

shortest path from v_k to v_j using only vertices in {v_1 , v_2 , . . . v_{k-1} }

Floyd's Algorithm for Shortest Paths – Algorithm 3.3

- Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.
- Inputs: A weighted, directed graph G with *n* vertices
- The graph is represented by a two-dimensional array W, its rows and columns indexed from 1 to n, where W [i] [j] is the weight on the edge from the ith vertex to the jth vertex.
- Outputs: A two-dimensional array D, its rows and columns indexed from 1 to n, where D [i] [j] is the length of a shortest path from the ith vertex to the jth vertex.

```
void floyed (int  n, const number W [ ] [ ], number D [ ] [ ] )
{
   index  i,   j ,  k ;
   D = W  ;
   for  (k = 1;  k <= 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  ] )
```

```
If( D[ i ][k ] +  D[k][j] < D[ i ][j]  ) D[ i ][j] = D[ i ][k] + D[k][j] ;
```

Analysis of Algorithm 3.3

- **Every-Case Time Complexity (Floyd's Algorithm for Shortest Paths)**
- Basic operation: The instruction in the **for-*j* loop.**
- Input size: *n, the number of vertices in the graph.*
- We have a loop within a loop within a loop, with *n* passes through each loop. So

  $T(n) = n^3 \in \theta(n^3)$

Does Dynamic Programming Apply to all Optimization Problems?

- No
- 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 substances.
- The principle of optimality must apply in the shortest path problem.
- Shortest Paths Problem
  - If v_k is a node on an optimal path from v_i to v_j then the sub-paths v_i to v_k and v_k to v_j are also optimal paths

- **Algorithm 3.4**
- **Floyd's Algorithm for Shortest Paths 2**
- Problem: Same as in Algorithm 3.3, except shortest paths are also created.

vertex P[*i*][*j*] = highest index of an intermediate vertex on the shortest path from vi to vj, if at least one intermediate vertex exists.
0, if no intermediate vertex exists.

```
void floyed2 (int  n, const number W [ ] [ ], number D [ ] [ ], index P [ ] [ ] )
{
   index  i,  , j ,  k ;
   for  ( i  = 1;  i  <= n;  i++)
      for  ( j  = 1;  j <= n;  j++)
            P [ i ] [ j ] =0;
    D = W ;

   for  (k = 1;  k <=  n;  k++)
      for  ( i  = 1;  i  <= n;  i++)
         for  ( j  = 1;  j <= n;  j++)
            if  ( D[ i ][ k] +  D[ k ][ j ] < D[ i ][ j ]  ) {
                  P[ i ][ j ] = k ;
                  D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ] ;

            }
```

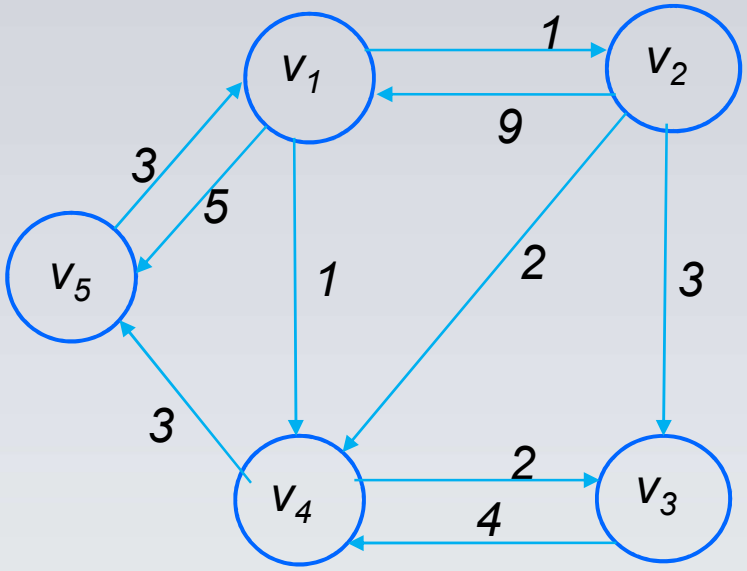
	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

To print Path(5, 3)

path(5, P[5][3])			vertex P[5][3]	path(P[5][3] ,3)
path(5, 4)			v_4	path(4 ,3)
path(5, P[5][4])	vertex P[5][4]	path(P[5][4] ,4)	v_4	path(4 ,3)
path(5, 1)	v_1	path(1 ,4)	v_4	path(4 ,3)
	v_1		v_4	

Add start and end

v_5	v_1	v_4	v_3



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

Algorithm 3.5

Print Shortest Path

Problem: Print the intermediate vertices on a shortest path from one vertex to another vertex in a

weighted graph.

Inputs: the array *P produced by Algorithm 3.4, and two indices, q and r, of vertices in the graph that*

Outputs: the intermediate vertices on a shortest path from *vq to vr.*

```
void path (index  q,  r )
{
   if (P [q ] [r ] != 0) {
         path (q, P [q ] [r ] ) ;
         cout << "v " << P [q ] [r ] ;
         path ( P [q ] [r ] ,  r ) ;
    }
}
```

Dynamic Programming and Optimization Problems

- 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.

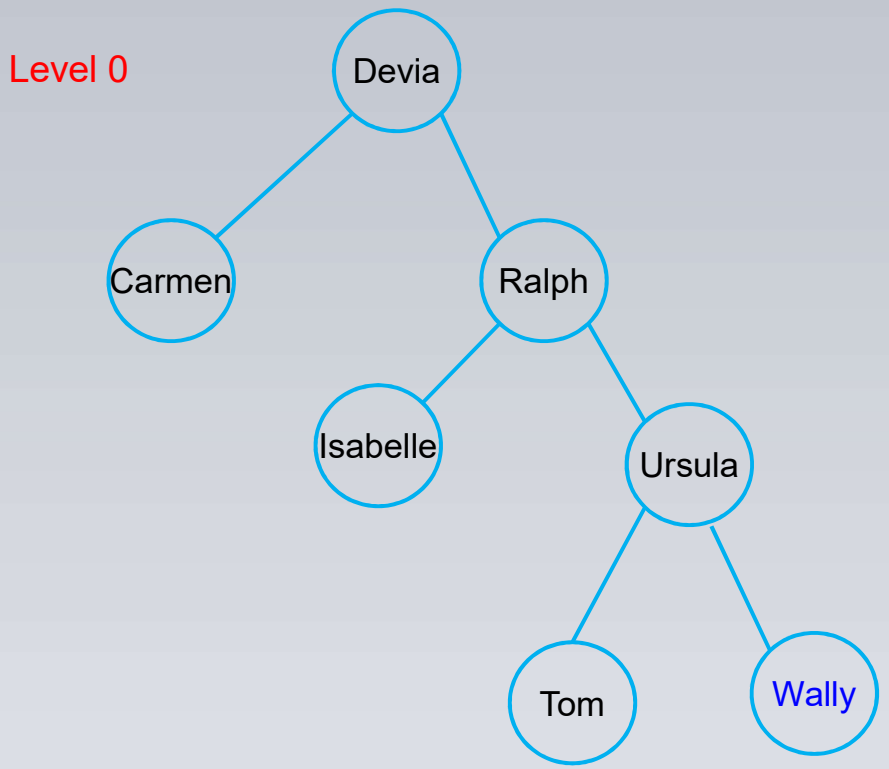
3.5 Optimal Binary Search Trees

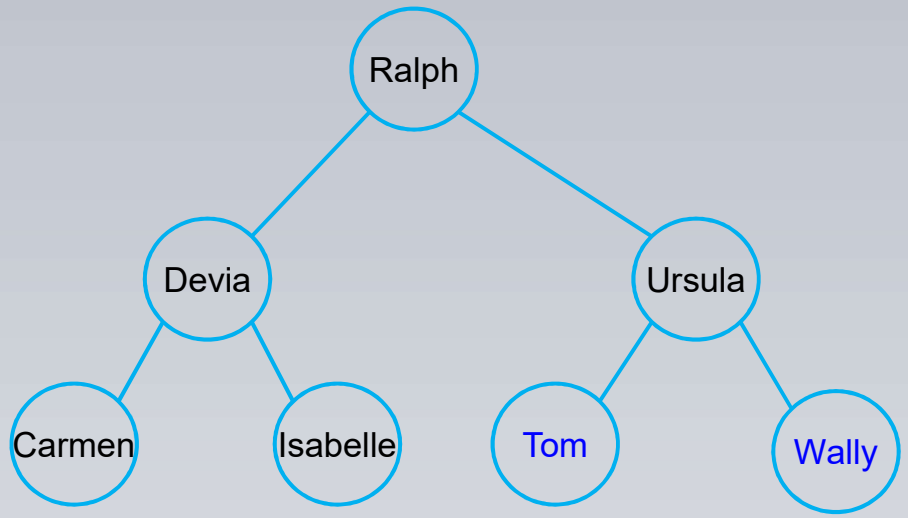
Definition

- A ***binary search tree*** is a binary tree of items (ordinarily called 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.

depth-first search





The ***depth*** of a node in a tree is the number of edges in the unique path from the root to the node.
The *depth of a tree* is the maximum depth of all nodes in the tree.

- A binary tree is called ***balanced*** if the depth of the two subtrees of every node never differ by more than 1.

- Our goal is to organize the keys in a binary search tree so that the average time it takes to locate a key is minimized.
- A tree that is organized in this fashion is called ***optimal.***
- It is not hard to see that, if all keys have the same probability of being the search key, the tree on the right in Figure 3.10 is optimal.

- We are concerned with the case where the keys do not have the same probability.
- An example of this case would be a search of one of the trees in Figure 3.10 for a name picked at random from people in the United States. Because “Tom” is a more common name than “Ursula,” we would assign a greater probability to “Tom.”
- (See Section A.8.1 in Appendix A for a discussion of randomness.)

Chained-Matrix Multiplication

- Optimal order for chained-matrix multiplication dependent on array dimensions
- Consider all possible orders and take the minimum: $t_n > 2^{n-2}$
- Principle of Optimality applies
- Develop Dynamic Programming Solution
- M[i,j] = minimum number of multiplications needed to multiply A_i through A_j

3.6 Traveling Salesperson Problem:TSP

- **n cities** : each city connects to some of the other cities by a road
- Minimize travel time – determine a shortest route that starts at the salesperson's home city, visits each city once, and ends at home city
- Instance : a weighted digraph.
- A **tour** (called a Hamiltonian circuit) in a digraph is a path from a vertex to itself that passes through each of the other vertices exactly once.
- An optimal tour in a weighted, digraph is such a path of minimum length.

- The following are the 3 tours and lengths for the graph :
- 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

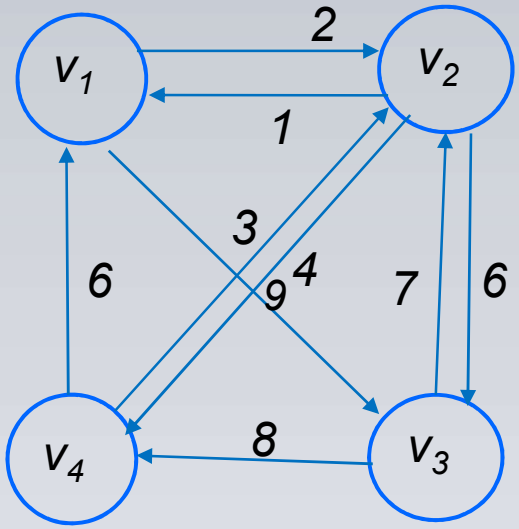
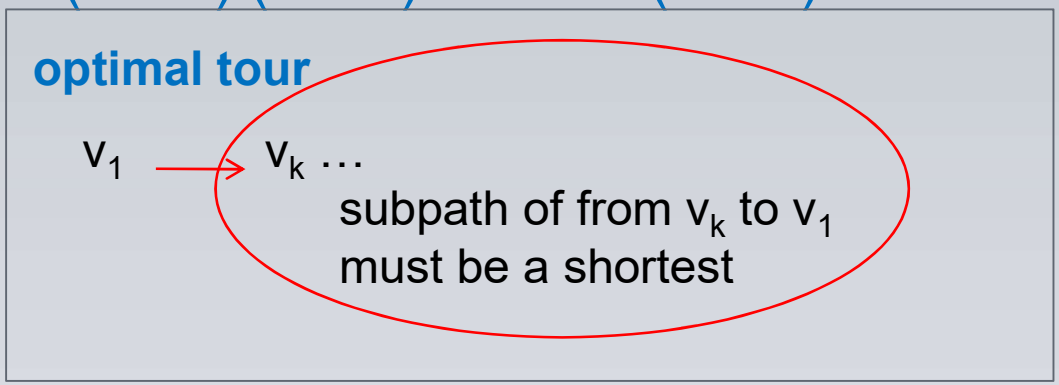


Figure 3.16

- Length [v_1 , v_2 , $\mathbf{v_4}$, $\mathbf{v_3}$, v_1]= ∞
- Length [$\mathbf{v_1}$, $\mathbf{v_4}$, v_2 , v_3 , v_1]= ∞
- Length [$\mathbf{v_1}$, $\mathbf{v_4}$, v_3 , v_2 , v_1]= ∞

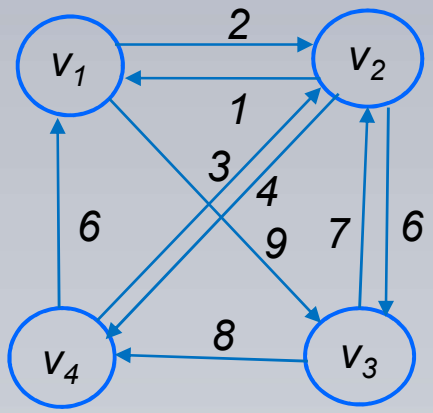
- If we consider all possible tours, the 2nd vertex on the tour can be any of $n - 1$ vertices, the 3rd vertex on the tour can be any of $n - 2$ vertices, … , the nth vertex on the tour can be only one vertex. Therefore, the total number of tours is

$$(n - 1)\,(n - 2)\ \ldots\ 1 = (n - 1)\,!$$



- This means that the principle of optimality applies, and we can use dynamic programming.

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0



Let $A \subseteq V$

D $[v_i]$ [A] length of shortest path from v_i to v_1 passing through each vertex of A exactly once.

D $[v_1]$ $[V - \{v_1\}]$ length of shortest path from v_1 to v_1 passing through each vertex of $V - \{v_1\}$ exactly once.

Example 3.10

If $A = \{ v_3 \}$, then D $[v_2]$ $[A]$ = length $[v_2 , v_3 , v_1] = \infty$

If $A = \{ v_3 , v_4 \}$, then D $[v_2]$ $[A]$ = min (length $[v_2, v_3 , v_4, v_1]$, length $[v_2, v_4 , v_3 , v_1]$)

= min (20, ∞) = **20**

- Because $V - \{ v_1 , v_j \}$ contains all the vertices except v_1 and v_j and the principle of optimality applies, we have

  Length of optimal tour = $\min_{2 \le j \le n} (W[1] [j] + D [v_j] [V - \{v_1, v_j \}])$

- In general, for $i \neq 1$ and v_i not in A,

$$D[v_i][A] = \min_{j: v_j \in A} (W[i][j] + D[v_j][A - \{v_j\}]) \quad \text{if } A \neq \phi$$

$$D [v_i] [\phi] = W[i] [1] \qquad (3.7)$$

- We can create a dynamic programming algorithm for the TSP problem using Equality 3.7. But first, let's illustrate how the algorithm would proceed.

Example 3.11

- Determine an optimal tour. First consider the empty set:

$$D[v_2][\phi] = 1$$
$$D[v_3][\phi] = \infty$$
$$D[v_4][\phi] = \mathbf{6}$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	**7**	0	8
4	**6**	**3**	∞	0

- Next consider all sets containing one element:

$$D[v_3][\{v_2\}] = \min_{j:\, v_j \in \{v_2\}} (W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$

$$= W[3][2] + D[v_2][\phi] = 7 + 1 = 8$$

$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$

$$D[v_2][\{v_4\}] = 4 + 6 = 10$$

$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$

- Next consider all sets containing **two** elements:

$$D[v_4][\{v_2, v_3\}] = \min_{j:\, v_j \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}])$$

$$= \min(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}])$$

$$= \min(3 + \infty, \infty + 8) = \infty$$

Similarly,

$$D[v_3][\{v_2, v_4\}] = \min(7 + 10, 8 + 4) = 12$$

$$D[v_2][\{v_3, v_4\}] = \min(6 + 14, 4 + \infty) = 20$$

Finally, compute the length of an optimal tour :

$$D[v_1][\{v_2, v_3, v_4\}] = \ldots$$

Algorithm 3.11

- **The Dynamic Programming Algorithm for the TSP Problem**
- Problem: Determine an optimal tour in a weighted, directed graph.
- Inputs: a weighted, digraph, represented by a 2-dimensional array *W* [*n*][*n*] and *n,* the number of vertices in the graph.
- Outputs: a variable ***minlength*** **(**optimal tour), and a 2-dim array *P*
  P [1..n][indexed by all subsets of $V - \{v_1\}$].
- *P* [*i*] [*A*] is 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 *travel* (int n, const number W [ ] [ ] , index P [ ] [ ], number& *minlength*)
{
index i, j, k ; number D [1..n] [subsets of $V - \{ v_1 \}$];
for ($i = 2$; $i <= n$; i ++)
D [i] [ϕ] = W [i] [1]
for (**k** = 1; $k <= n$ - 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] = $\min_{\text{j: vj} \in \text{A}}$ (W [i] [j] + D [j] [A-{ v_j }]) ;
P [i] [A] = value of j that gave the minimum;
}
D [1] [V $- \{ v_1 \}$] = $\min_{2 \leq j \leq n}$ (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 $- \{v_1\}$] ;
}

Thoerem 3.1

For all $n \geq 1$

$$\sum_{k=1}^{n} k \binom{n}{k} = n2^{n-1}$$

Analysis of Algorithm 3.11

- **Every-Case Time and Space Complexity (The Dynamic Programming Algorithm for the Traveling Salesperson Problem)**
- Basic operation: The time in both the first and last loops is insignificant compared to the time in the middle loop because the middle loop contains various levels of nesting. Therefore, we will consider the instructions executed for each value of v_j to be the basic operation. They include an addition instruction.
- Input size: *n,* the number of vertices in the graph.
- For each set *A* containing *k* vertices, we must consider *n − 1 − k* vertices, and for each of these vertices, the basic operation is done *k* times. Because the number of subsets A of $V - \{v_1\}$ containing *k* vertices is equal to c(n-1, k), the total number of times the basic operation is done is given by

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)k\binom{n-1}{k} \qquad (3.8)$$

It is not hard to see that

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

Substituting this equality into Equality 3.8, we have

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

Applying theorem 3.1,

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

Dynamic Programming Algorithm for Traveling Sales Person Problem

- $\Theta(n2^n)$
- Inefficient solution using dynamic programming
- Problem NP-Complete: no one has ever developed a polynomial-time algorithm, but no one has ever shown that such an algorithm is not possible.

Example 3.12

- 20-city.
- Assuming that the time to process the basic instruction in Brute-Force algorithm is 1 microsecond, and that the time to compute the length of each tour in Dynamic Prog. algorithm is 1 microsecond.

  Brute-Force alg 19! μs= 3857 years

  Dynamic Prog. Algorithm (20-1) (20-2) 2^{20-3} μs = 45 seconds