# Dynamic Programming

## Backgrounds

- **Explores the space of all *possible solutions***
  - Decompose a problem into a series of subproblems
  - Build up correct solutions to larger and larger subproblems

## Backgrounds

- The term *Dynamic Programming* comes from Control Theory
- *Programming* refers to the use of tables (arrays) to construct a solution.
- Used extensively in "Operation Research" taught in the Math dept.

## Key Idea

- Dynamic programming usually *reduces time* by using *more space*
- Solve the problem by solving subproblems of increasing size, while saving each solution for a subproblem in a table
- Use the table to find the solution to larger problems.
- Time is saved since each subproblem is solved only once

# Steps to Design a Dynamic Programming Algorithm

- Establish a recursive property giving the solution to an instance of the problem
- Solve an instance of the problem in a bottom-up fashion by solving smaller instances first
  - Recall that divide & conquer is top-down
  - Divide and conquer blindly divides a problem into smaller instances
    - A divide & conquer algorithm, e.g., the recursive nth Fibonacci number algorithm, may solve the same instance more than once → Slow

## Fibonacci's Series

# Fibonacci's Series:

- **Definition:** $S_0 = 0$,

  $S_1 = 1$,

  $S_n = S_{n-1} + S_{n-2}$ $n>1$

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

- Applying the recursive definition we get:

```
fib (n)
1. if n < 2
2.    return n
3. else return (fib (n -1) + fib(n -2))
```

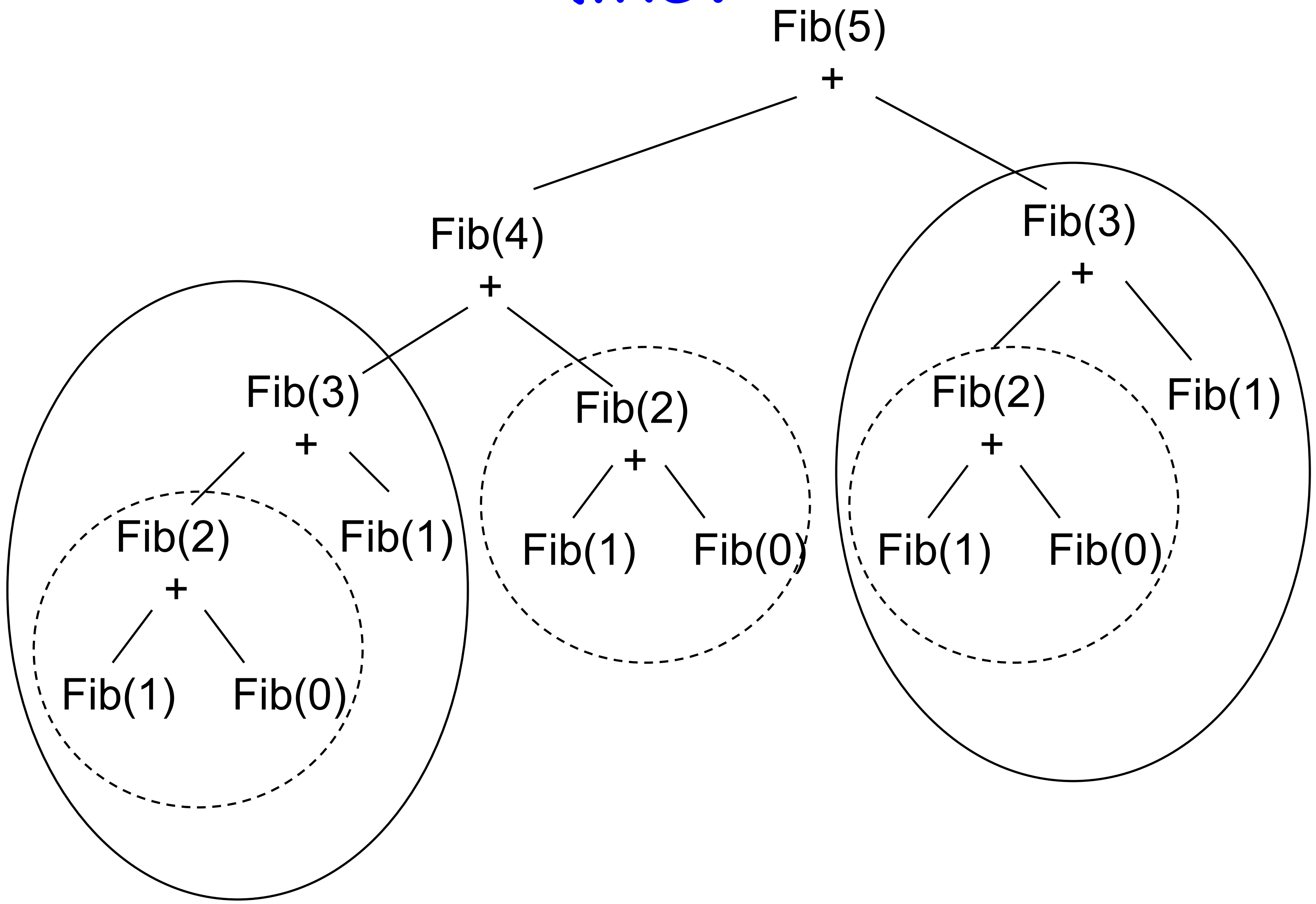
# Analysis using Substitution of Recursive Fibonacci

- Let T(n) be the number of additions done by fib(n)

- $T(n)=T(n-1)+T(n-2)+1$ for $n>=2$
  $T(2)=1$

- $T(n) \in O(2^n)$, $T(n) \in \Omega(2^{n/2})$

# What does the Recursion Tree look like?



# Dynamic Programming Solution for Fibonacci

- **Build a table with the first *n* Fibonacci numbers.**

```
fib(n)
  1. A[0] =0
  2. A[1] = 1
  3. for i ← 2 to n
  4.        A[ i ] = A [i -1] + A[i -2 ]
  5. return A
```

**Is there a recurrence equation?**

**How many additions are needed?**

**What is the space requirements?**

## Binomial Coefficient

## Binomial Coefficient

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

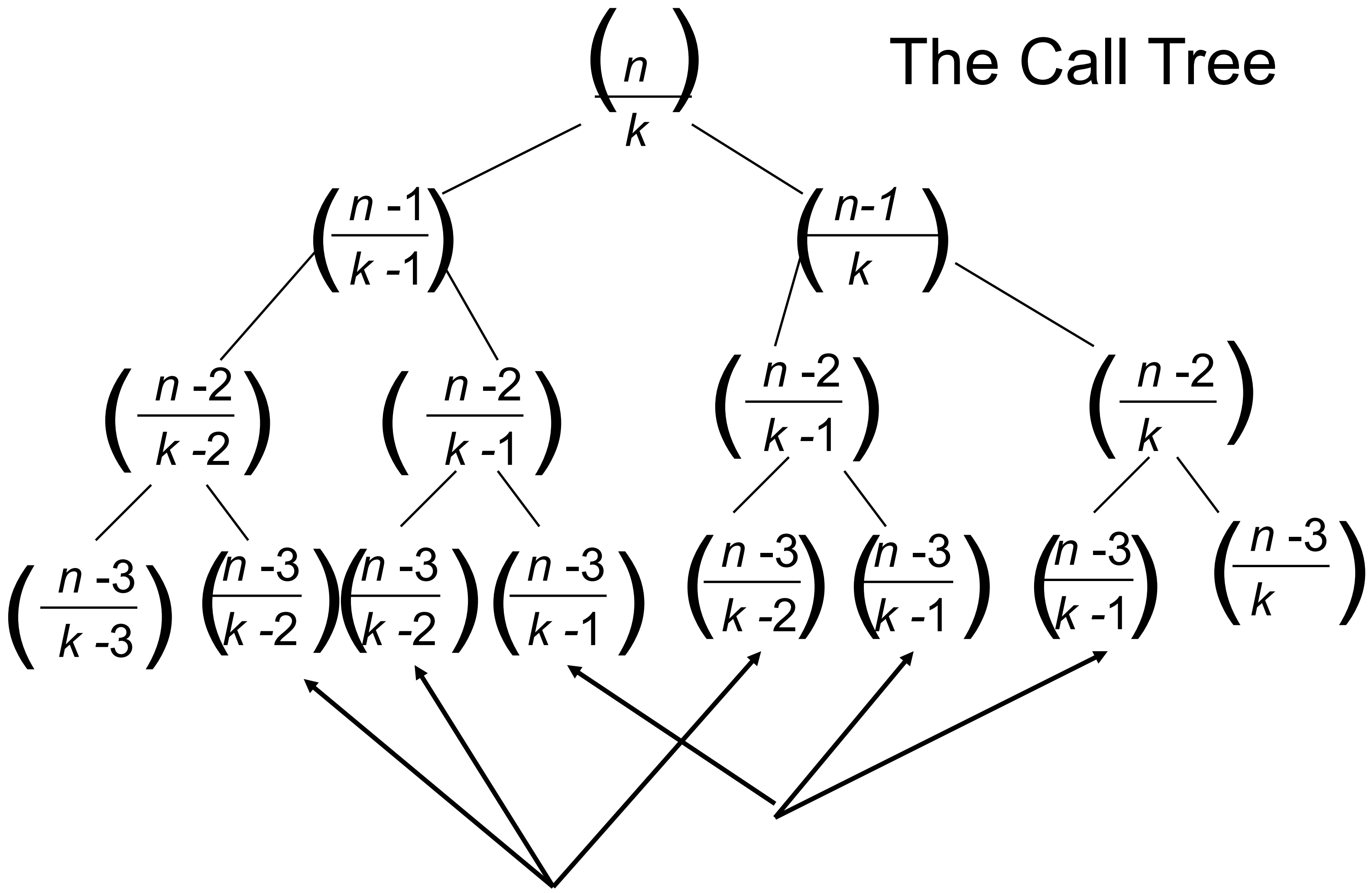
$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < \mathrm{k} < \mathrm{n} \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

# Recursive algorithm

```
binomialCoef($n$, $k$)
    1.  if $k = 0$ or $k = n$
    2.      return 1
    3. else
            return (binomialCoef($n$-1, $k$-1) + binomialCoef($n$-1, $k$))
```

```
binomialCoef($n$, $k$)
1.  if $k$ = 0 or $k$ = $n$
2.      then return 1
3.  else return (binomialCoef($n$ –1, $k$ –1) + binomialCoef($n$–1, $k$))
```

## The Call Tree



## Dynamic Programming Solution

- Use a matrix B of n+1 rows, k+1 columns where

$$B[n,k]=\binom{n}{k}$$

- *Establish* a recursive property. Rewrite in terms of matrix B:

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

- Solve all “smaller instances of the problem” in a *bottom-up* fashion by computing the rows in *B* in sequence starting with the first row.

# The B Matrix

$$\begin{array}{c|ccccccc|}
 & 0 & 1 & 2 & 3 & 4 \ldots & j & k \\
\hline
0 & \mathbf{1} & & & & & & \\
1 & 1 & 1 & & & & & \\
2 & \mathbf{1} & \mathbf{2} & \mathbf{1} & & & & \\
3 & 1 & 3 & 3 & 1 & & & \\
4 & \mathbf{1} & \mathbf{4} & \mathbf{6} & \mathbf{4} & \mathbf{1} & & \\
 & & & & & & & \\
i & & & & & & & \\
 & & & & & & & \\
n & & & & & & & \\
\hline
\end{array}$$

$$\begin{tikzcd}
\mathbf{B}[i\ \mathbf{-1}, j\ \mathbf{-1}] \arrow[dr] & \mathbf{B}[i\ \mathbf{-1}, j\,] \arrow[d] \\
 & \mathbf{B}[\,i,\ j\,]
\end{tikzcd}$$

# Compute B[4,2]= $\binom{4}{2}$

- Row 0: **B[0,0] =1**
- Row 1: **B[1,0] = 1**
  **B[1,1] = 1**
- Row 2: **B[2,0] = 1**
  **B[2,1] = B[1,0] + B[1,1] = 2**
  **B[2,2] = 1**
- Row 3: **B[3,0] = 1**
  **B[3,1] = B[2,0] + B[2,1] = 3**
  **B[3,2] = B[2,1] + B[2,2] = 3**
- Row 4: **B[4,0] = 1**
  **B[4,1] = B[3, 0] + B[3, 1] = 4**
  **B[4,2] = B[3, 1] + B[3, 2] = 6**

## Dynamic Program

- bin($n$,$k$ )

```
1. for i = 0 to n  // every row
2.    for j = 0 to minimum( i, k )
3.       if j == 0 or j == i   // column 0 or diagonal
4.            B[ i , j ] = 1
5.       else
6.            B[ i , j ] =B[i -1, j -1]  +  B[i -1, j ]
7. return B[ n, k ]
```

- What is the run time?
- How much space does it take?
- If we only need the last value, can we save space?

# Dynamic programming

- All values in column 0 are 1
- All values in the $k$+1 diagonal cells are 1
- $j \neq i$ and $0 < j <= \min\{i,k\}$ ensures we only compute B[ $i, j$ ] for $j < i$ and only first $k$+1 columns

- **Elements above diagonal (B[i,j] for j > i) are not computed since $\binom{i}{j}$ is undefined for j > i**

## Number of iterations

**i:** **0, 1, 2, …, k, k+1, …, n**
**#iterations: 1, 2, 3, …k+1, k+1, …, k+1**

$$\sum_{i=0}^{n} \sum_{j=0}^{\min(i,k)} 1 = \sum_{i=0}^{k} \sum_{j=0}^{i} 1 + \sum_{i=k+1}^{n} \sum_{j=0}^{k} 1 =$$

$$\sum_{i=0}^{k} (i+1) + \sum_{i=k+1}^{n} (k+1) =$$

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

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

# Dynamic Programming to solve an Optimization Problem

- Used for problems in which an optimal solution for the original problem can be found from optimal solutions to subproblems of the original problem

- A recursive algorithm based on the divide and conquer approach can solve the problem. But the recursive algorithm may compute the solution to the same subproblem more than once and, therefore, it is slow.
  - Fibonacci term and binomial coefficient have such inefficient recursive algorithms
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - More details on Fibonacci term and binomial coefficient later

- To reduce time, compute the optimal solution of a subproblem only once and save its value. Reuse the saved value whenever the same subproblem needs to be solved.

## Principle of Optimality – Optimal Substructure

- Principle of optimality
  - Given an optimal sequence of decisions or choices, each subsequence must also be optimal.

- The principle of optimality applies to a *problem* (not an algorithm)
  - A large number of optimization problems satisfy this principle. Examples follow.

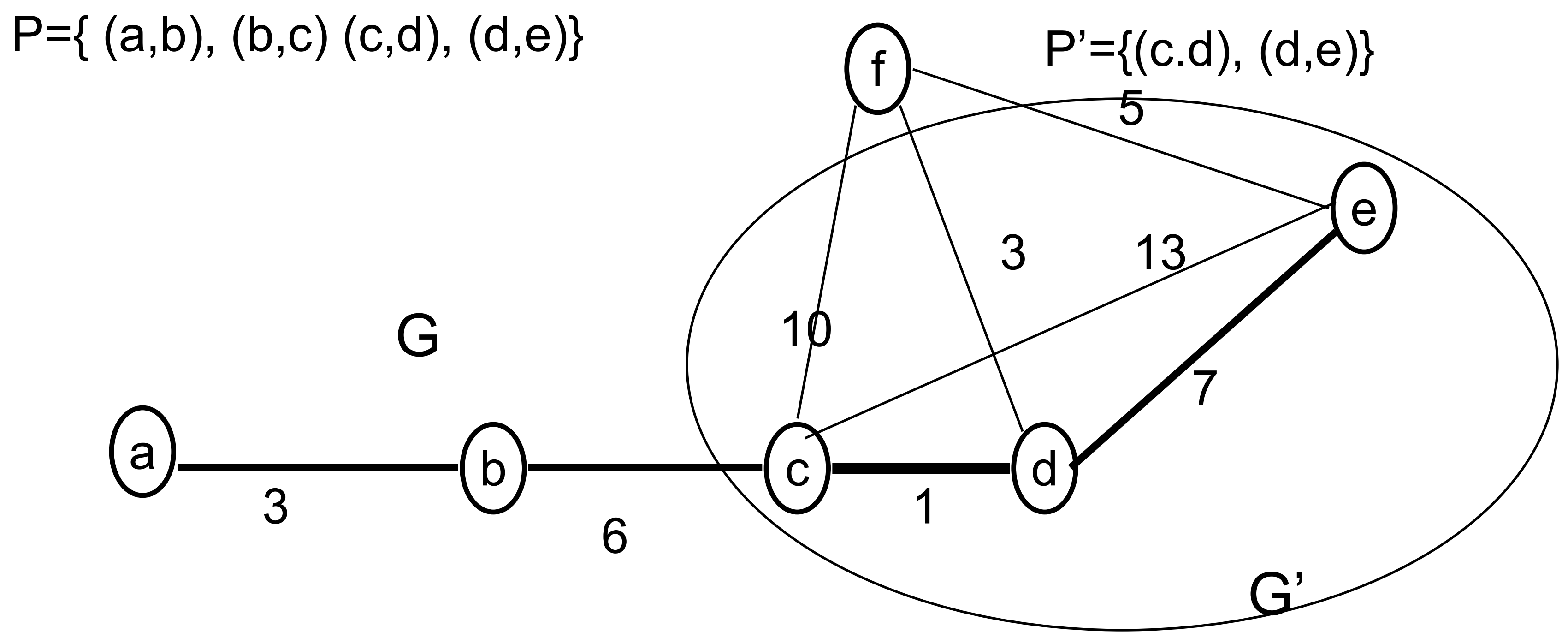
# Principle of optimality – Shortest path problem

**Problem:** Given a graph *G* and vertices *s* and *t*, find a shortest path in *G* from *s* to *t*

**Theorem:** A subpath P' (from s' to t') of a shortest path P is a shortest path from s' to t' of the subgraph G' induced by P'. *Subpaths* are paths that start or end at an intermediate vertex of P.

**Proof**: If P' was not a shortest path from s' to t' in G',
we can substitute the subpath from s' to t' in P
by the shortest path in G' from s' to t'.
The result is a shorter path from s to t than P.
This contradicts our assumption that P is a shortest path
from s to t.

# Principle of optimality: Example



P' must be a shortest path from c to e in G'. Otherwise, P cannot be a shortest path from a to e in G.

## Principle of optimality
### MST (minimum spanning tree) problem

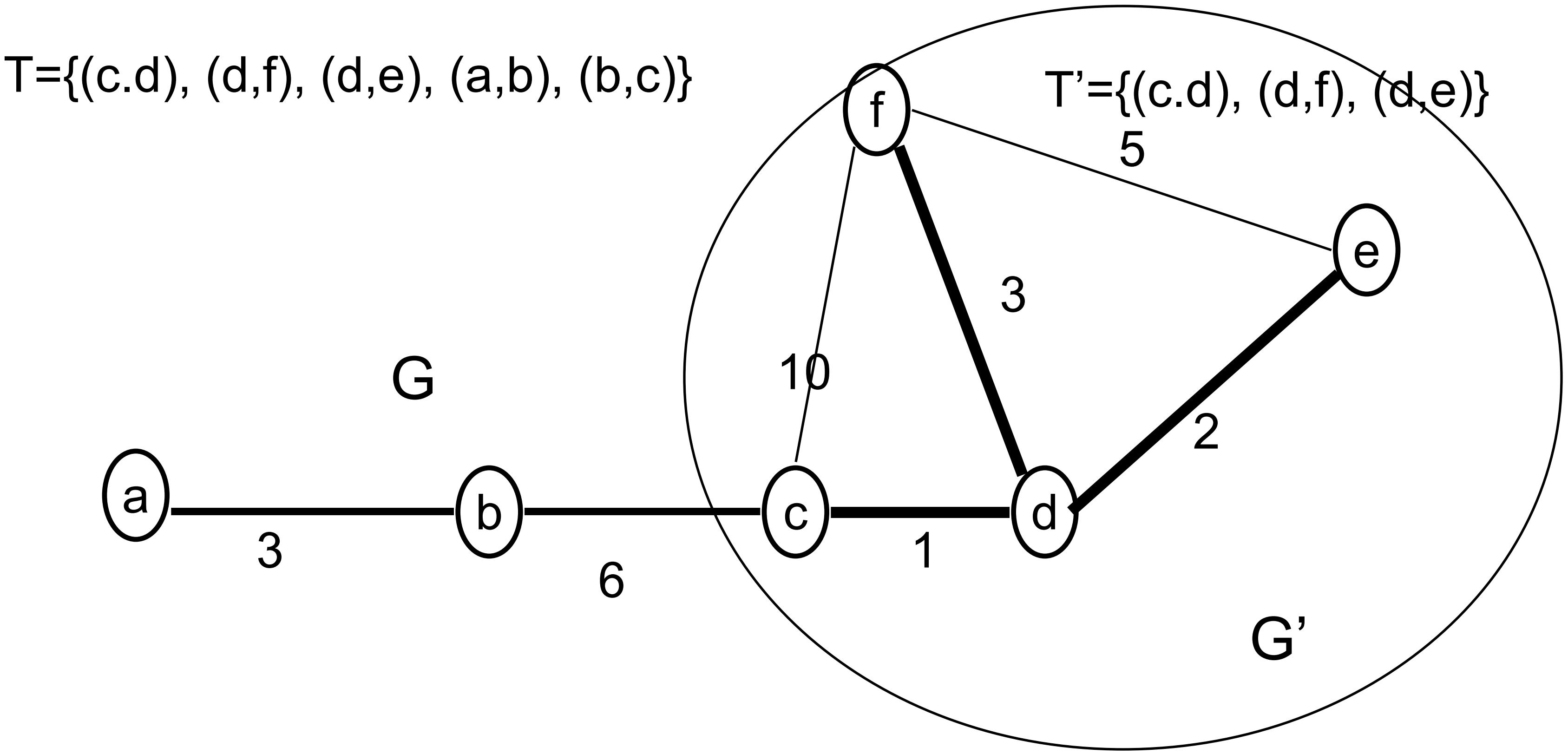
- Spanning tree: A tree connecting all the vertices in an undirected connected graph
- Minimum spanning tree: A spanning tree with the minimum weight

# Principle of optimality for MST problem

- **Problem**: Given an undirected connected graph *G*, find a minimum spanning tree
- **Theorem**: Any subtree *T'* of an *MST T* of *G* is an *MST* of the subgraph *G'* of *G* induced by the vertices of the subtree.

**Proof**: If *T'* is not an *MST* of *G'*, we can substitute the edges of *T'* in *T*, with the edges of an MST of G'. This would result in a lower cost spanning tree, contradicting our assumption that *T* is an *MST* of *G*.

# Principle of optimality



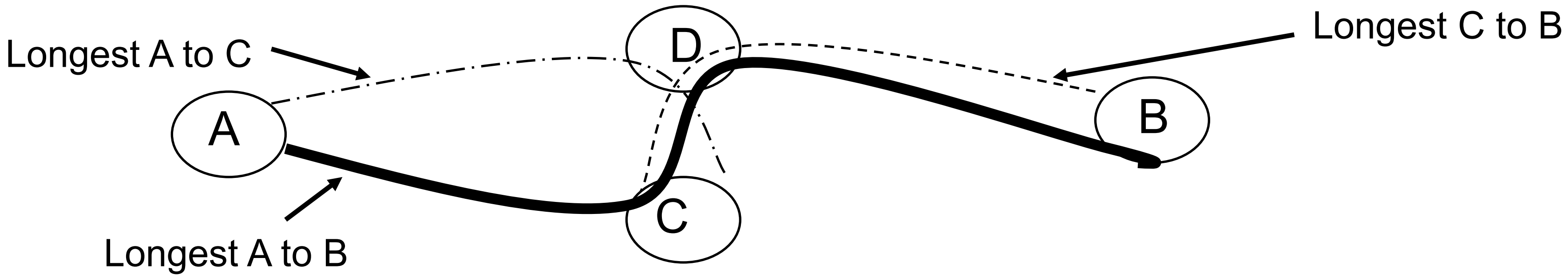
T′ must be an MST of G′, otherwise T cannot be an MST

# A problem that does not satisfy Principle of Optimality

**Problem**: What is the longest simple route between City A and B?

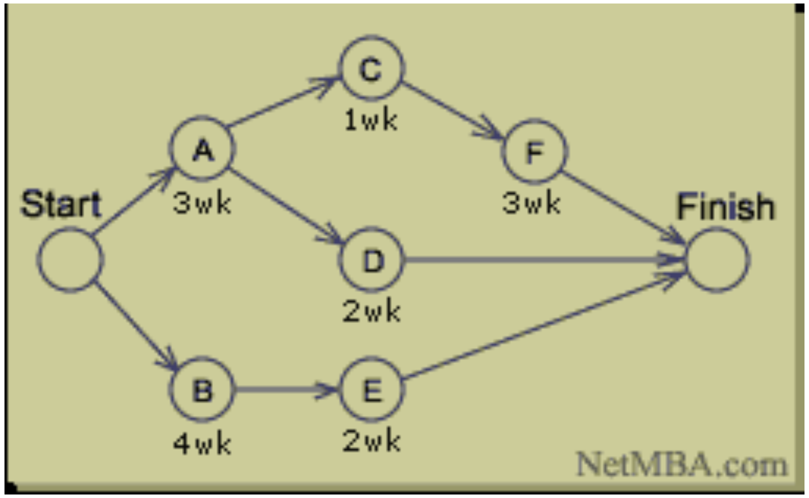
- Simple route → Never visit the same city twice.

- The longest simple route (solid line) has city C as an intermediate city.
- It does not consist of the longest simple route from A to C plus the longest simple route from C to B.



# Longest path problem

- Longest path problem
  - Critical Path Management: How long does it take to shut down and restart inter-connected chemical plants? (DuPont, 1957)
  - CPM is widely applied to project management too



# Floyd's Algorithm

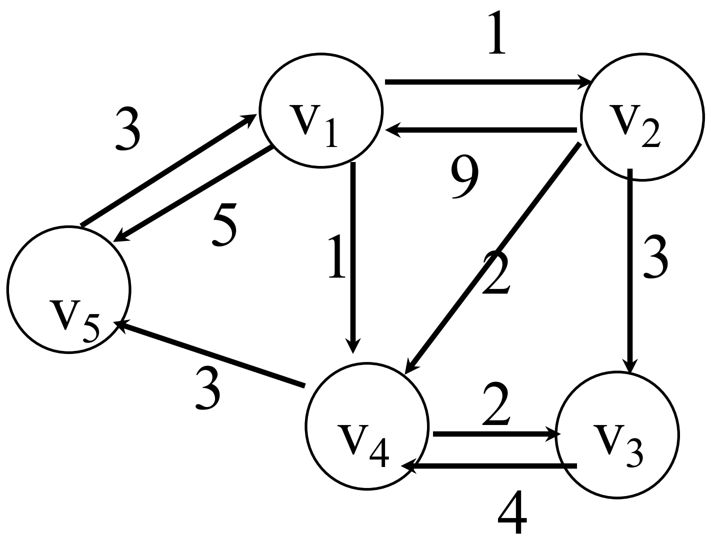
All pairs shortest path

## All pairs shortest path

- *The problem:* find the shortest path between every pair of vertices of a graph
  - Expensive using a brute-force approach
- *The graph* may contain no negative cycles
- *Representation*: a weight matrix where  
  W(i,j)=0 if i=j  
  W(i,j)=∞ if there is no edge between i and j  
  W(i,j)="weight of edge"
- Note: we have shown principle of optimality applies to shortest path problems

## The weight matrix and the graph

| | 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 |



Adjacency Matrix

## The subproblems

- How can we define the shortest distance $d_{i,j}$ in terms of “smaller” problems?
- One way is to restrict the paths to only include vertices from a restricted subset.
- Initially, the subset is empty.
- Then, it is incrementally increased until it includes all the vertices.

## The subproblems

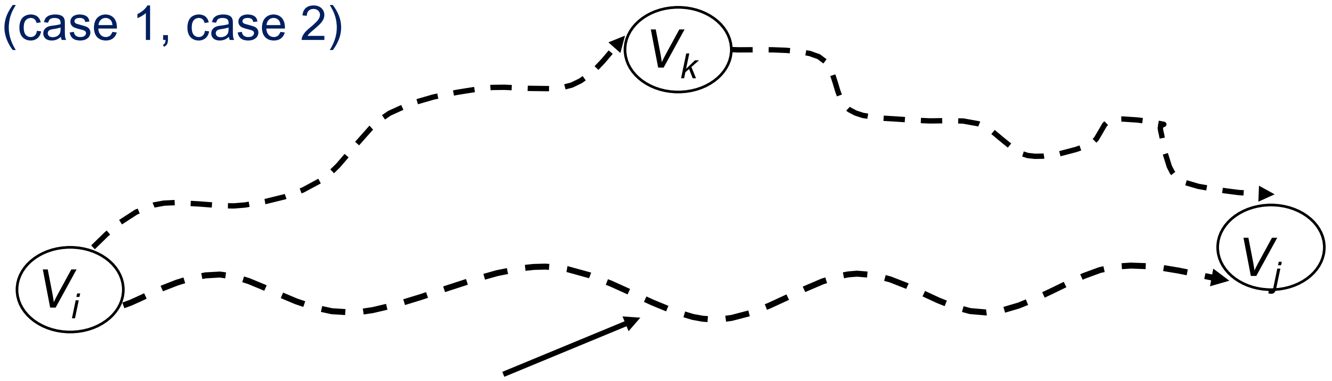
- Let $D^{(k)}[i,j]$ = weight of a shortest path from $v_i$ to $v_j$ using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - $D^{(0)}=W$
  - $D^{(n)}=D$ which is the goal matrix
- How do we compute $D^{(k)}$ from $D^{(k-1)}$ ?

## The Recursive Definition:

Case 1: A shortest path from $v_i$ to $v_j$, which only uses vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, does not use $v_k$ $\Rightarrow$ $D^{(k)}[i,j] = D^{(k-1)}[i,j]$

Case 2: A shortest path from $v_i$ to $v_j$, which only uses vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices, does use $v_k$ $\Rightarrow$ $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$

Shortest path from $v_i$ to $v_j$ using intermediate vertices $\{V_1, \ldots V_k\}$ = min(case 1, case 2)

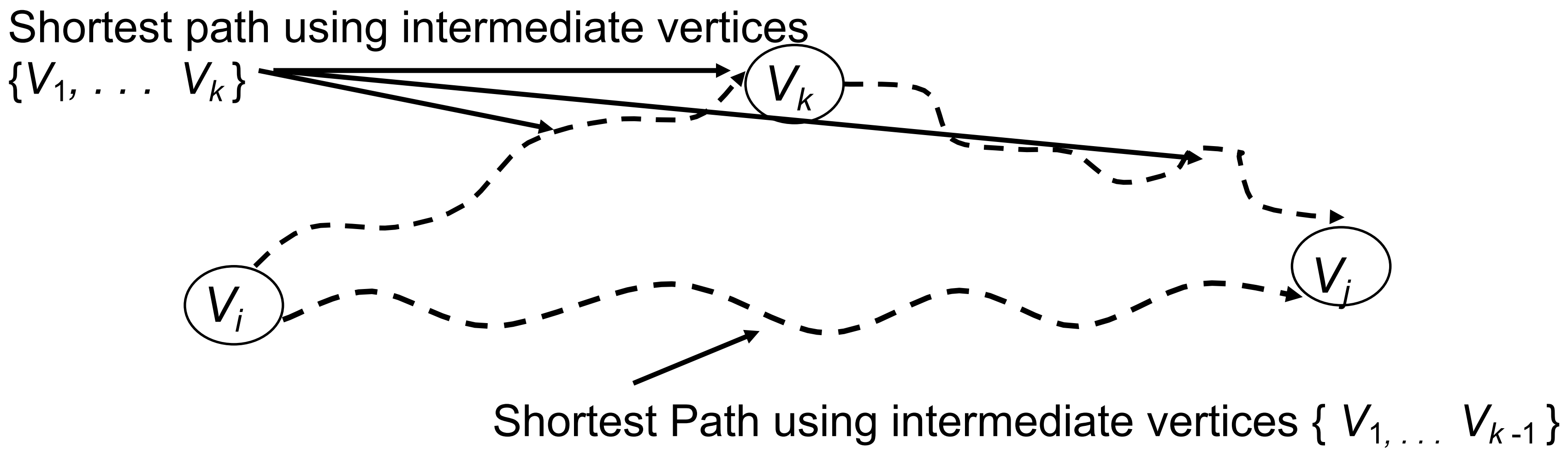


Shortest Path using intermediate vertices $\{ V_{1, \ldots} V_{k-1} \}$

# The recursive definition

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

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


Shortest path using intermediate vertices $\{V_1, . . . \ V_k\}$
$V_k$
$V_i$
$V_j$
Shortest Path using intermediate vertices $\{ V_{1,} . . . \ V_{k-1} \}$

## The pointer array P

- Used to enable finding a shortest path
- Initially the array contains 0

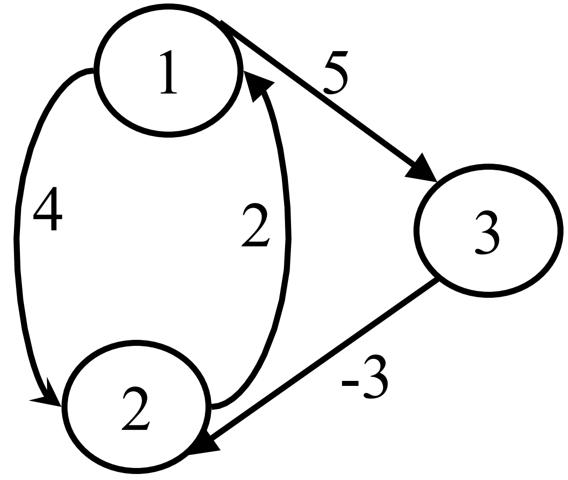
- Each time a shorter path from *i* to *j* is found, save the *k* value that provided the minimum

- To print the intermediate nodes on the shortest path, execute a recursive procedure that prints the shortest paths from *i* and *k*, and from *k* to *j*

# Floyd's Algorithm Using n+1 *D* matrices

```
Floyd (input graph G)
   // Compute shortest distance between all pairs of nodes in G;
   // Save P to find shortest paths
    1. $D^0 \leftarrow W$    // initialize D array to W [ ]
    2. $P \leftarrow 0$      // initialize P array to [0]
    3. for $k \leftarrow 1$ to n
    4.     for $i \leftarrow 1$ to n
    5.         for $j \leftarrow 1$ to n
    6.             if ($D^{k-1}[i, j] > D^{k-1}[i, k] + D^{k-1}[k, j]$)
    7.             then  $D^k[i, j] \leftarrow D^{k-1}[i, k] + D^{k-1}[k, j]$
    8.                   $P[i, j] \leftarrow k$;
    9.             else $D^k[i, j] \leftarrow D^{k-1}[i, j]$
```

# Example



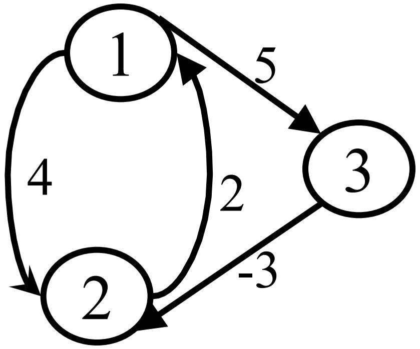
$$W = D^0 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & \infty \\ \hline 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 0 \\ \hline 3 & 0 & 0 & 0 \\ \hline \end{array}$$

# k = 1

## Vertex 1 can be intermediate node

$$D^1 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & \infty & -3 & 0 \\ \hline \end{array}$$



$D^1[2,3] = \min( D^0[2,3], D^0[2,1]+D^0[1,3] )$

$= \min(\infty, 7)$

$= 7$

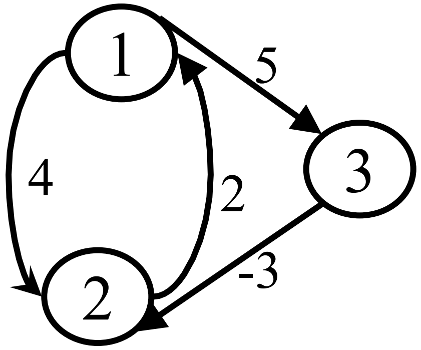
$D^1[3,2] = \min( D^0[3,2], D^0[3,1]+D^0[1,2] )$

$= \min(-3, \infty) = -3$

$$D^0 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & \infty \\ \hline 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 0 & 0 & 0 \\ \hline \end{array}$$

## k = 2: Vertices 1, 2 can be intermediate



$$D^1 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & \infty & -3 & 0 \\ \hline \end{array}$$

$$D^2 = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ \hline 2 & 2 & 0 & 7 \\ \hline 3 & -1 & -3 & 0 \\ \hline \end{array}$$

$D^2[1,3] = \min( D^1[1,3], D^1[1,2]+D^1[2,3] )$

$= \min (5, 4+7)$

$= 5$

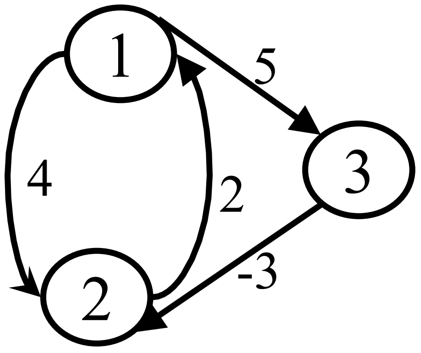
$$P = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 1 \\ \hline 3 & 2 & 0 & 0 \\ \hline \end{array}$$

$D^2[3,1] = \min( D^1[3,1], D^1[3,2]+D^1[2,1] )$

$= \min (\infty, -3+2)$

$= -1$

## k = 3: Vertices 1, 2, 3 can be intermediate



$$D^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \end{array}$$

$$D^3 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 2 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \end{array}$$

$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$

$= \min(4, 5+(-3))$

$= 2$

$$P = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 3 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 2 & 0 & 0 \end{array}$$

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$

$= \min(2, 7+(-1))$

$= 2$

## Floyd's Algorithm: Using two D matrices

Floyd

```
1. D ← W   // initialize D array to W [ ]
2. P ← 0     // initialize P array to [0]
3. for k ← 1 to n
 // Computing D' from D
4.       for i ← 1 to n
5.            for j ← 1 to n
6.                 if (D[i, j] > D[i, k] +D[k,j ] )
7.                 then D'[i, j] ← D[ i, k ] + D[ k, j ]
8.                         P[ i, j ] ← k;
9.                 else  D'[ i, j ] ← D[ i, j ]
10.    Move D' to D
```

## Can we use only one D matrix?

- $D[i,j]$ depends only on elements in the $k^{th}$ column and row of the distance matrix

- We will show that the $k^{th}$ row and the $k^{th}$ column of the distance matrix are unchanged when $D^k$ is computed

- This means $D$ can be calculated *in-place*

## The main diagonal values

- Before we show that the *k*th row and column of *D* remain unchanged, we show that the main diagonal remains 0

- $D^{(k)}[\, j,j \,] = \min\{ D^{(k-1)}[\, j,j \,] ,\ \ D^{(k-1)}[\, j,k \,] + D^{(k-1)}[\, k,j \,] \}$
  $= \min\{ 0,\ \ D^{(k-1)}[\, j,k \,] + D^{(k-1)}[\, k,j \,] \}$
  $= 0$
- Based on which assumption?
  - There is no negative cycle

# The $k^{th}$ column

- $k^{th}$ column of $D^k$ is equal to the $k^{th}$ column of $D^{k\text{-}1}$
- *Intuitively true* - a path from i to k will not become shorter by adding k to the allowed subset of intermediate vertices
- For all i,

$$\begin{aligned} D^{(k)}[i,k] &= \min\{ D^{(k-1)}[i,k],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,k] \} \\ &= \min\{ D^{(k-1)}[i,k], D^{(k-1)}[i,k] + 0 \} \\ &= D^{(k-1)}[i,k] \end{aligned}$$

## The $k^{th}$ row

- k^th row of $D^k$ is equal to the $k^{th}$ row of $D^{k-1}$

- *Intuitively true* - A path from k to j will not become shorter by adding k to the allowed subset of intermediate vertices

- For all *j*,

$$\begin{aligned} D^{(k)}[k,j] &= \min\{ D^{(k-1)}[k,j], \mathrm{D}^{(k-1)}[k,k] + D^{(k-1)}[k,j] \} \\ &= \min\{ D^{(k-1)}[ k,j ], 0 + D^{(k-1)}[k,j ] \} \\ &= D^{(k-1)}[ k,j ] \end{aligned}$$

## Floyd's Algorithm using a single *D*

Floyd

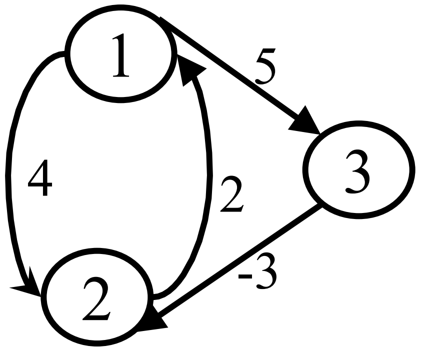
```
1. $D \leftarrow W$   // initialize $D$ array to $W$[ ]
2. $P \leftarrow 0$    // initialize P array to [0]
3. for $k \leftarrow 1$ to $n$
4.     do for $i \leftarrow 1$ to $n$
5.         do for $j \leftarrow 1$ to $n$
6.             if ($D[i, j] > D[i, k] + D[k, j]$)
7.              then {$D[i, j] \leftarrow D[i, k] + D[k, j]$
8.                    $P[i, j] \leftarrow k$; }
```

Time-complexity: $T(n)=n*n*n = n^3$

# Printing intermediate nodes on shortest path from q to r

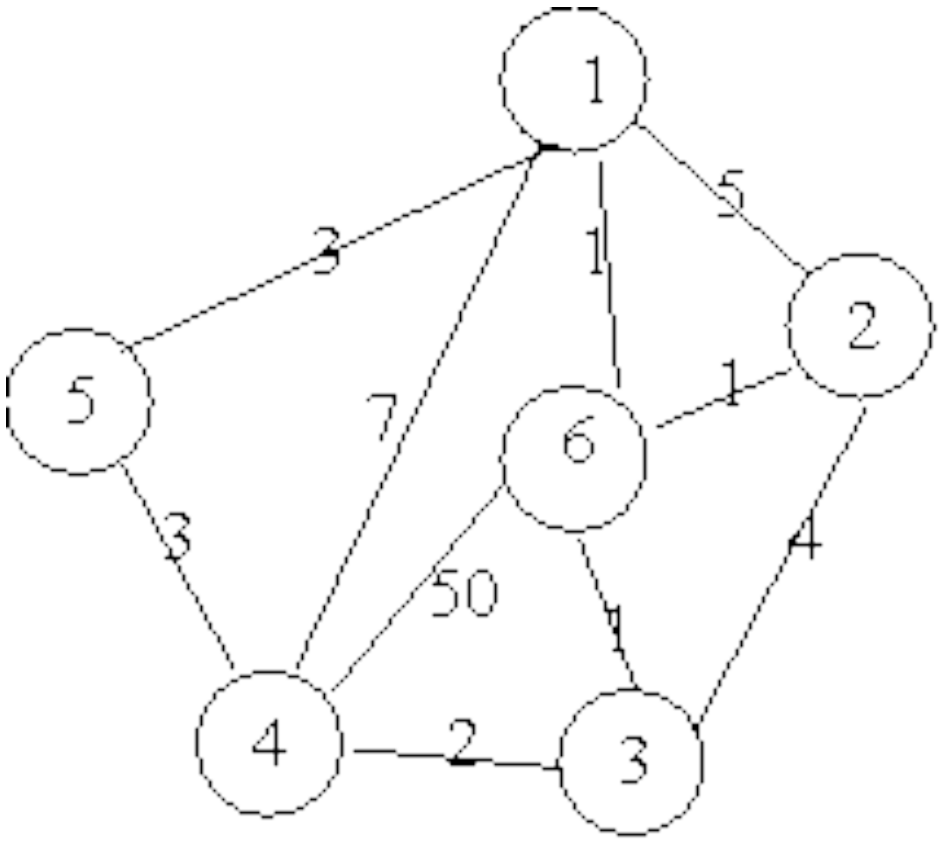
```
path(index q, r)
    if (P[ q, r ]!=0)
            path(q, P[q, r])
            println( "v"+ P[q, r])
            path(P[q, r], r)
    return;

  //no intermediate nodes
    else return
```

$$P = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 3 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 2 & 0 & 0 \end{array}$$



- *If* D[q, r] < ∞, print node q and call *path(…)*
- After returning from *path(…),* print node r

## Graph for a Floyd example

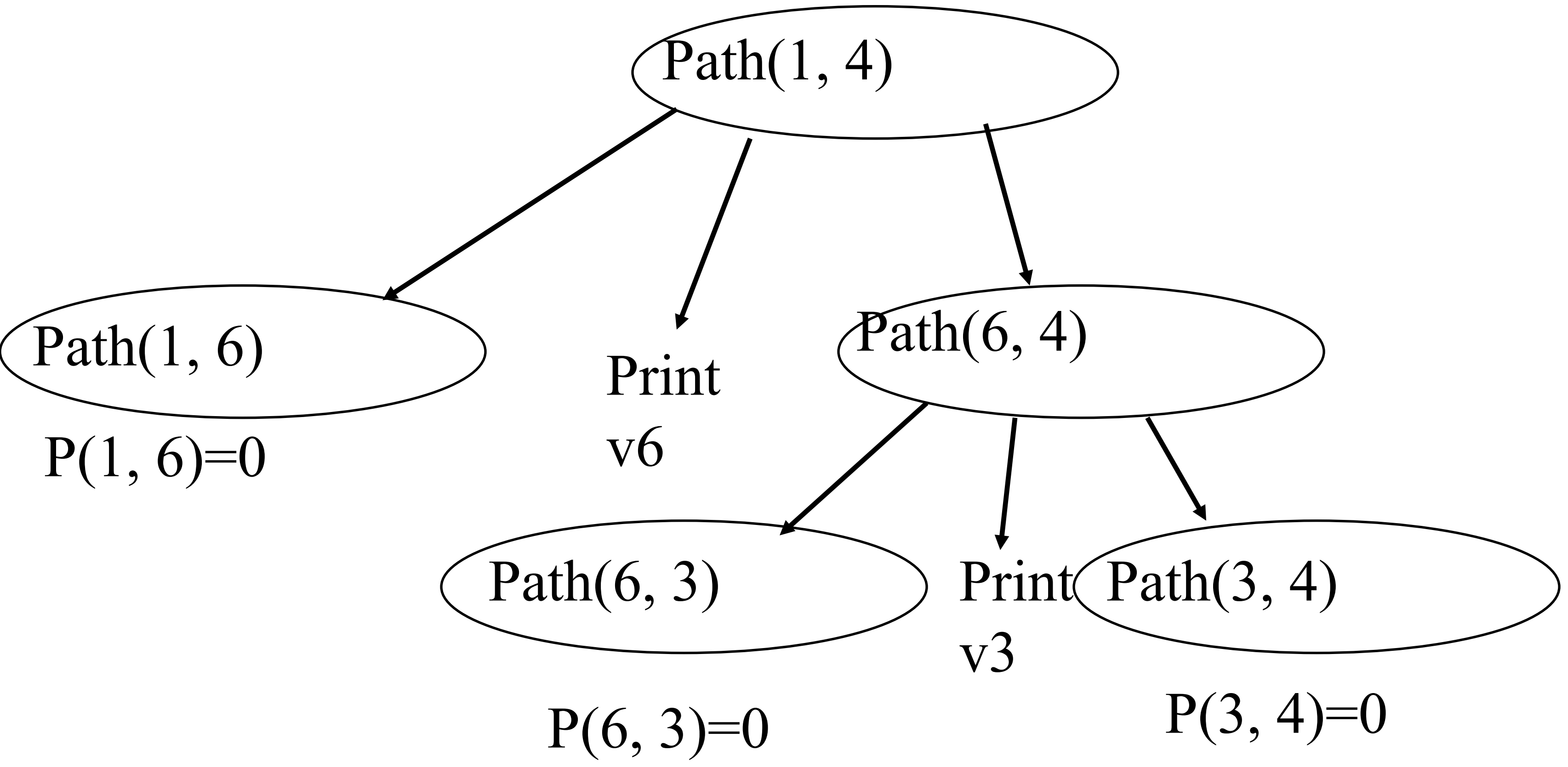


# The final distance matrix D and P

$$D^6 = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 2(6) & 2(6) & 4(6) & 3 & 1 \\ 2 & 2(6) & 0 & 2(6) & 4(6) & 5(6) & 1 \\ 3 & 2(6) & 2(6) & 0 & 2 & 5(4) & 1 \\ 4 & 4(6) & 4(6) & 2 & 0 & 3 & 3(3) \\ 5 & 3 & 5(6) & 5(4) & 3 & 0 & 4(1) \\ 6 & 1 & 1 & 1 & 3(3) & 4(1) & 0 \end{array}$$

The values in parenthesis are the nonzero P values – intermediate nodes that minimize the distance.

# The call tree for Path(1, 4)



The intermediate nodes on the shortest path from 1 to 4 are v6, v3.
The shortest path is v1, v6, v3, v4.

## Optimal Binary Search Tree

## Binary Search Trees

- A binary tree of keys such that
  - Each node has one key
  - The keys in the left subtree of a given node are smaller than or equal to the item in the node
  - The keys in the right subtree of a given node are greater than the item in the node

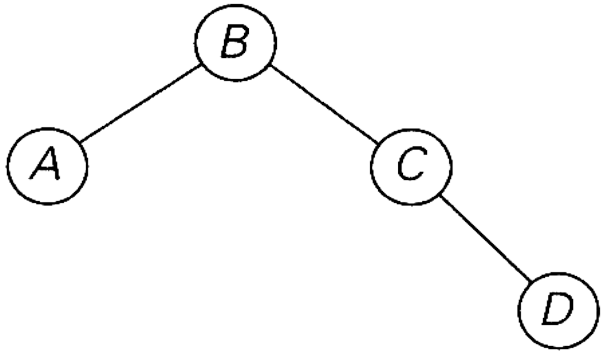
# Optimal Binary Search Tree

- Problem
  - Given sequence $K = k_1 < k_2 <\cdots < k_n$ of $n$ sorted keys, with a search probability $p_i$ for each key $k_i$.
  - Build a binary search tree (BST) with minimum expected search cost.
  - Actual cost = # of items examined.
  - For key $k_i$, cost = depth$_T(k_i)$+1, where depth$_T(k_i)$ = depth of $k_i$ in BST $T$ .

- Minimize the expected search time for a given probability distribution

# Binary Search Trees

| Index i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Key | A | B | C | D |
| P(i) | 0.4 | 0.3 | 0.2 | 0.1 |



Average number of comparisons

$$\sum_{i=1}^{n} P(i) \cdot (depth_T(K_i) + 1)$$

# Optimal Binary Search Tree

Let $T$ be a binary search tree that contains $K_i$ , $K_{i+1}$ , . . . , $K_j$ for some $1 \le i \le j \le n$.
We define the *cost* of *average search time* as

$$c(T) = \sum_{l=i}^{j} P(l) \cdot (depth_T(K_l) + 1)$$

**Recurrence relation**

Let $T_L$ and $T_R$ be the left and right sub-trees of $T$. Then

$$c(T) = c(T_L) + c(T_R) + \underbrace{\sum_{l=i}^{j} P(l)}_{\text{Tree depth is increased by 1}}$$

Tree depth is increased by 1

# Optimal Binary Search Tree

- *Principle of optimality*

- Let *T* be a binary search tree that has the minimum cost among all trees containing keys $K_i$ , $K_{i+1}$ , . . . , $K_j$

- Let $K_m$ be the key at the root of *T*. So, $i \leq m \leq j$.

- $T_L$, the left subtree of *T*, is a binary search tree that has the minimum cost among all trees containing keys $K_i , . . . , K_{m-1}$

- $T_R$, the right subtree of *T*, is a binary search tree that has minimum cost among all trees containing keys $K_{m+1} , . . . . , K_j$

# Expected Search Cost

$$E[\text{search cost in } T]$$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

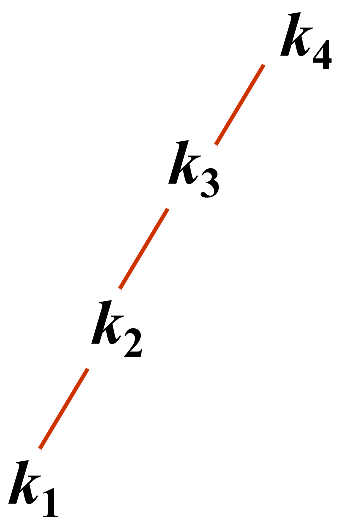
$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

Sum of probabilities is 1.

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i$$

## Example

- Consider 5 keys with these search probabilities: $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, $p_4 = 0.4$

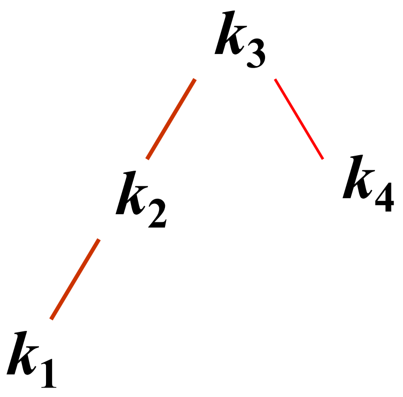


| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i) \cdot p_i$ |
|---|---|---|
| 1 | 3 | 0.3 |
| 2 | 2 | 0.4 |
| 3 | 1 | 0.3 |
| 4 | 0 | 0 |
| Sum | | 1 |

Therefore, E[search cost] = 2

# Example

- $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, $p_4 = 0.4$



| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i)\cdot p_i$ |
|---|---|---|
| 1 | 2 | 0.2 |
| 2 | 1 | 0.2 |
| 3 | 0 | 0 |
| 4 | 1 | 0.4 |
|  |  | 0.8 |

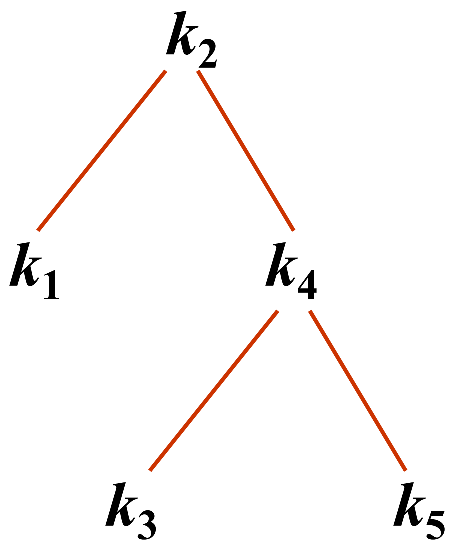
Therefore, E[search cost] = 1.8

This tree turns out to be optimal for this set of keys.

*Optimal BST may not have the highest-probability key at root.*

# Example

- Consider 5 keys with these search probabilities:
  $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.

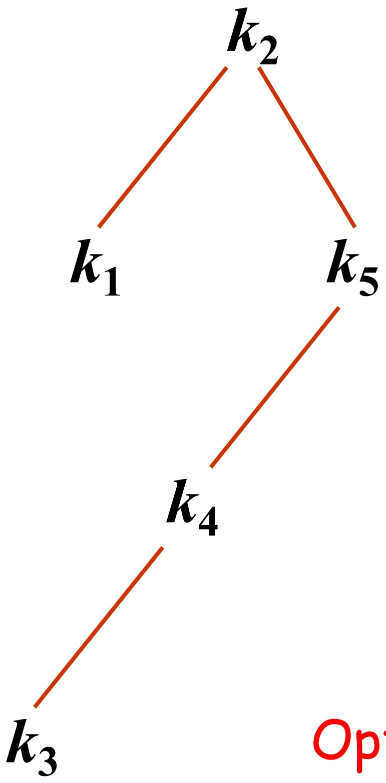


| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i) \cdot p_i$ |
|---|---|---|
| 1 | 1 | 0.25 |
| 2 | 0 | 0 |
| 3 | 2 | 0.1 |
| 4 | 1 | 0.2 |
| 5 | 2 | 0.6 |
| | | 1.15 |

Therefore, E[search cost] = 2.15.

# Example

- $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.



| $i$ | $\text{depth}_T(k_i)$ | $\text{depth}_T(k_i) \cdot p_i$ |
|---|---|---|
| 1 | 1 | 0.25 |
| 2 | 0 | 0 |
| 3 | 3 | 0.15 |
| 4 | 2 | 0.4 |
| 5 | 1 | 0.3 |
| | | 1.10 |

Therefore, E[search cost] = 2.10.

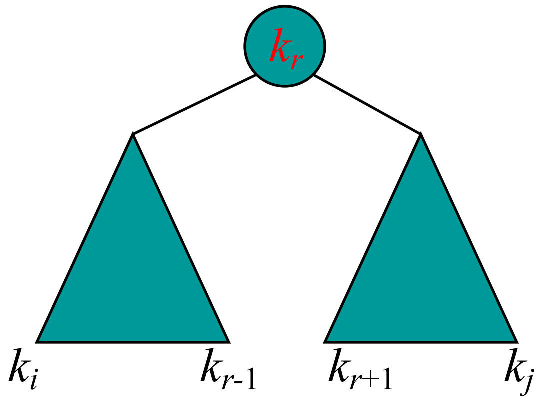
Optimal BST may not have the smallest height.

## Example

- **Observations:**
  - Optimal BST may not have the smallest height.
  - Optimal BST may not have the highest-probability key at root.
- Build by exhaustive checking? (brute force)
  - Construct every possible *n*-node BST
  - For each BST, assign keys and compute expected search cost
  - Not efficient

# Optimal Substructure

- One of the keys in $k_i, \ldots, k_j$, say $k_r$, where $i \leq r \leq j$, must be the root of an optimal subtree for these keys.
- Left subtree of $k_r$ contains $k_i, \ldots, k_{r-1}$
- Right subtree of $k_r$ contains $k_r+1, \ldots, k_j$



- To find an optimal BST:
  - Examine all candidate roots $k_r$ , for $i \leq r \leq j$
  - Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$

- *Reuse optimal BSTs to find a bigger optimal BST with more nodes ➔ Dynamic programming!*

# Computing an Optimal Solution: Initialization for pseudo code in the next slide

For each subproblem $(i,j)$, store:

- expected search cost in a table $A[1..n+1, 0..n]$
  - Will use only entries $A[i, j]$, where $j \geq i-1$.
- Root$[i, j]$ = root of subtree with keys $k_i,..,k_j$, for $1 \leq i \leq j \leq n$.
- $P[1..n+1, 0..n]$ = sum of probabilities
  - $p[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $p[i, j] = p[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$.

- *The Optimal Solution is:*
  - *$A[1, n]$ (minimum search time)*
  - *$Root[1, n] = R$ $(1<=r<=n)$*
  - Make an optimal binary search tree based on the root at R with keys $k_1,\ldots, k_n$.

## Optimal binary search tree – Algorithm

```
Void optsearchtree(int n, const float p[], flot& minavg, index R[][])
{
integer i, j, k, diagonal;
float A[1..n+1][0..n];
for (i = 1; i <= n; i++) //initialization
{
        A[i][i-1] = 0;
        R[i][i-1] = 0;
        A[i][i] = p[i];
        R[i][i] = i;
 }

A[n+1] [n] = 0;
R[n+1][n] = 0;
```

## Optimal binary search tree – Algorithm (Cont.)

```
for (diagonal = 1; diagonal <= n-1; diagonal++)
    for (i = 1; i <= n - diagonal; i++)
    {
        j = i + diagonal;
        $A[i][j] = \min_{i \le k \le j}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m$
        R[i][j] = a value of k that gave the minimum
    }
minavg = A[1][n];
}
```

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

- The nested loop iterates $\Theta(n^2)$ and each iteration takes $\Theta(n)$ to find the minimum

## Longest Common Subsequence (LCS)

# Longest Common Subsequence (LCS)

- Problem: Given sequences x[1..m] and y[1..n], find a longest common subsequence of both.

- Example: **x=ABCBDAB and y=BDCABA**
  - BCA is a common subsequence and
  - BCBA and BDAB are two LCSs
  - Note: A substring is contiguous, but a subsequence doesn't have to be

- Applications
  - Pattern Analysis
  - Bioinformatics

## LCS

- Come up with a brute force solution and evaluate it
- Write a recurrence equation
- Dynamic programming solution

# Brute force solution

- Solution: **For every subsequence of x, check if it is a subsequence of y.**
- **Analysis :**
  - Each subsequence of X corresponds to a subset of the indices {1, 2, …., m} of X. There are $2^m$ subsequences of x.

$$C_m^1 + C_m^2 + C_m^3 + \ldots + C_m^m = 2^m$$

  - Each check takes O(n) time, since we scan y for the first element, and then scan for the second element, etc.
  - The worst case running time is $O(n2^m)$.

# Writing the recurrence equation

- Let $X_i$ denote the *$i^{th}$ prefix*, i.e., x[1..i], of x[1..*m*], and $X_0$ denotes an empty prefix

- We will first compute the *length of an LCS of $X_m$ and $Y_n$, LenLCS*(*m*, *n*), and then use the length information to find the actual subsequence

- We need a recursive formula for computing *LenLCS*(*i*, *j*)

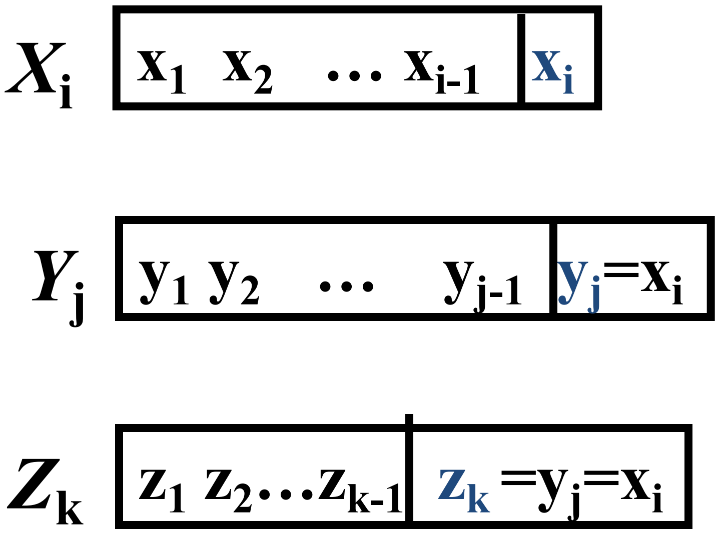
## Writing the recurrence equation

- If $X_i$ and $Y_j$ end with the same character $x_i = y_j$, the LCS must include the character. If they did not, we might get a longer LCS by adding a common character.

- If $X_i$ and $Y_j$ do not end with the same character there are two possibilities:
  - Either the LCS does not end with $x_i$,
  - or it does not end with $y_j$

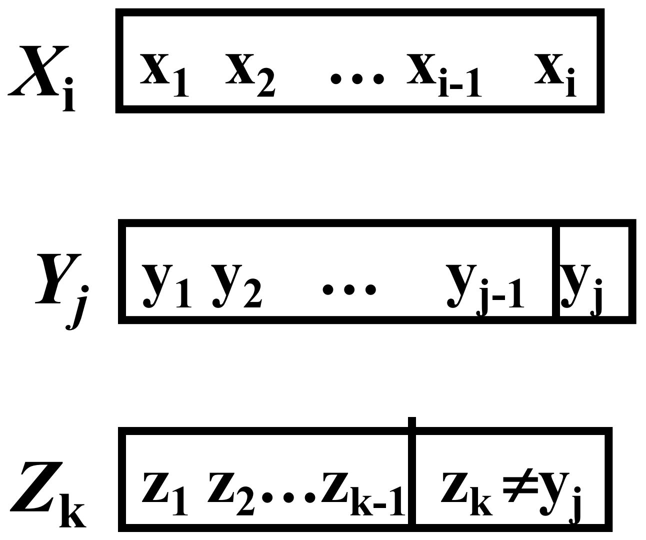
- Let $Z_k$ denote an LCS of $X_i$ and $Y_j$

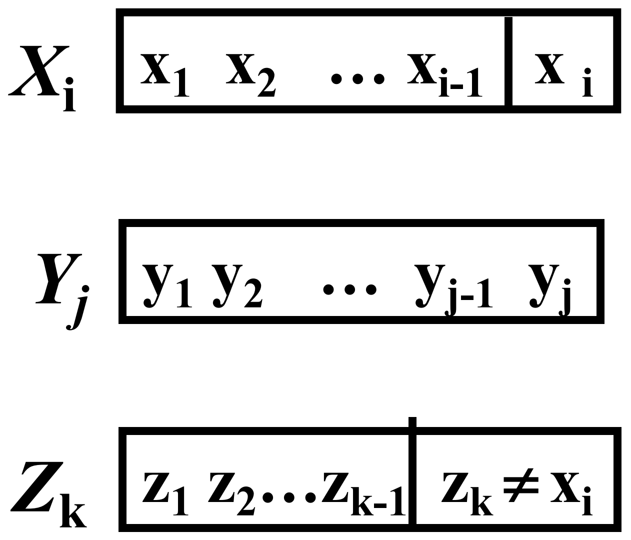
## $X_i$ and $Y_j$ end with $x_i=y_j$



$Z_k$ is $Z_{k-1}$ followed by $z_k = y_j = x_i$ where $Z_{k-1}$ is an LCS of $X_{i-1}$ and $Y_{j-1}$ and $LenLCS(i, j)=LenLCS(i-1, j-1)+1$

# $X_i$ and $Y_j$ end with $x_i \neq y_j$


$X_i$
$x_1$ $x_2$ … $x_{i-1}$ $x_i$
$Y_j$
$y_1$ $y_2$ … $y_{j-1}$ $y_j$
$Z_k$
$z_1$ $z_2$…$z_{k-1}$ $z_k \neq y_j$


$X_i$
$x_1$ $x_2$ … $x_{i-1}$ $x_i$
$Y_j$
$y_1$ $y_2$ … $y_{j-1}$ $y_j$
$Z_k$
$z_1$ $z_2$…$z_{k-1}$ $z_k \neq x_i$

**$Z_k$ is an LCS of $X_i$ and $Y_{j-1}$** **$Z_k$ is an LCS of $X_{i-1}$ and $Y_j$**

$$LenLCS(i, j) = \max\{LenLCS(i, j-1), LenLCS(i-1, j)\}$$

# The recurrence equation

$$lenLCS(i,j)=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ lenLCS(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } x_\mathrm{i}=y_\mathrm{j} \\ \max\{lenLCS(i-1,j), lenLCS(i,j-1)\} & \text{otherwise} \end{cases}$$

# Dynamic programming solution

- Initialize the first row and the first column of the matrix *LenLCS* to 0
- Calculate *LenLCS* (1, *j*) for *j* = 1,…, *n*
- Then the *LenLCS* (2, *j*) for *j* = 1,…, *n*, etc.
- Store also in a table an *arrow* pointing to the array element that was used in the computation.

# LCS-Length(X, Y)

```
m ← length[X}
n ← length[Y]
for i ← 1 to m do
    c[i, 0] ← 0
for j ← 1 to n do
    c[0, j] ← 0
```

# LCS-Length(X, Y) cont.

```
for i ← 1 to m do
    for j ← 1 to n do
        if $x_i = y_j$
            c[i, j] ← c[i-1, j-1]+1
            b[i, j] ← "D"
        else
            if c[i-1, j] ≥ c[i, j-1]
                c[i, j] ← c[i-1, j]
                b[i, j] ← "U"
            else
                c[i, j] ← c[i, j-1]
                b[i, j] ← "L"
return c and b
```

Time Complexity: $\Theta(mn)$

# Example

| | $y_j$ | B | D | C | A |
|---|---|---|---|---|---|
| $x_i$ | 0 | 0 | 0 | 0 | 0 |
| A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 |
| B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 |
| C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 |
| B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 |

To find an LCS follow the arrows, for each diagonal arrow there is a member of the LCS

## Memoization

## Idea

- Memoization is a top-down approach for dynamic programming. It is used for recursive code that solves subproblems more than once.

- The idea is to modify the recursive code, and  use a table to store the solution for every problem that was already solved.

- Logically equivalent to recursive calls, but recursive calls are replaced with table look-ups for more efficiency

## Initialization

- Before applying the recursive code, you need to initialize the table to some "impossible value".
- For example, if the function can have only positive values you can initialize the table to negative values.

## Method

- The recursive code is changed to first check the table to determine whether you have already solved the current problem or not.
- If the value in the table is still the initial value, you know that you have not yet solved the current problem. So, you solve the subproblem and store the solution in the table.
- Otherwise, you know that the value is the required solution and use or return it directly as appropriate.

## Example 1

The following code is for a memoized version of recursive Fibonacci.

```
MemoizedFib(n)
    for (i=0; i<=n; i++)
      A[i]=-1 //initialize the array
    return LookupFib(n)


LookupFib(n) //compute fn if not yet done
if A[n]== -1      //initial value
    if n<=1  //base case
       A[n] = n //store solution

    //solve and store a general case
    else A[n] = (LookupFib(n-1)+LookupFib(n-2))

return A[n] //return f_n = A[n]
```

If A[n] <> -1, simply return A[n] making no more recursive calls!!!

# Example 2

The following code is for a memoized version of the recursive code for binomial coefficients

```
•   memoizedBC(n, k)
•    for i=0 to n
•      for j=0 to min(i, k)
•         B[i, j] = -1 //initialize array
•   return binomialCoef(n, k)

•   binomialCoef(n, k)
      if B[n, k] == -1 //B[n, k] not yet computed
•       if k = 0 or k = n //Base case
•         B[n,k] = 1 //solve base case and store

•       //solve and store a general case
•       else B[n,k]=(binomialCoef(n-1,k -1)+binomialCoef(n-1, k))

•     return B[n, k]
```

If B[n,k] <> -1, simply return B[n,k] with no recursive calls!!!

## Summary

- Dynamic programming is a powerful tool
- Bottom-up approach
- Store results for smaller instances and reuse them to solve a bigger instance

- Now let’s move on to graphs and greedy algorithms