MF20006: Introduction to Computer Science

Lecture 8: Algorithm III

Hui Xu

xuh@fudan.edu.cn

Outline

1. Dynamic Programming

2. Coin Problem

3. Longest Common Subsequence

4. Shortest Path Problem

5. Travels Salesman Problem

Warm Up

- **Problem: Given a number n, compute the n-th Fibonacci number defined by**

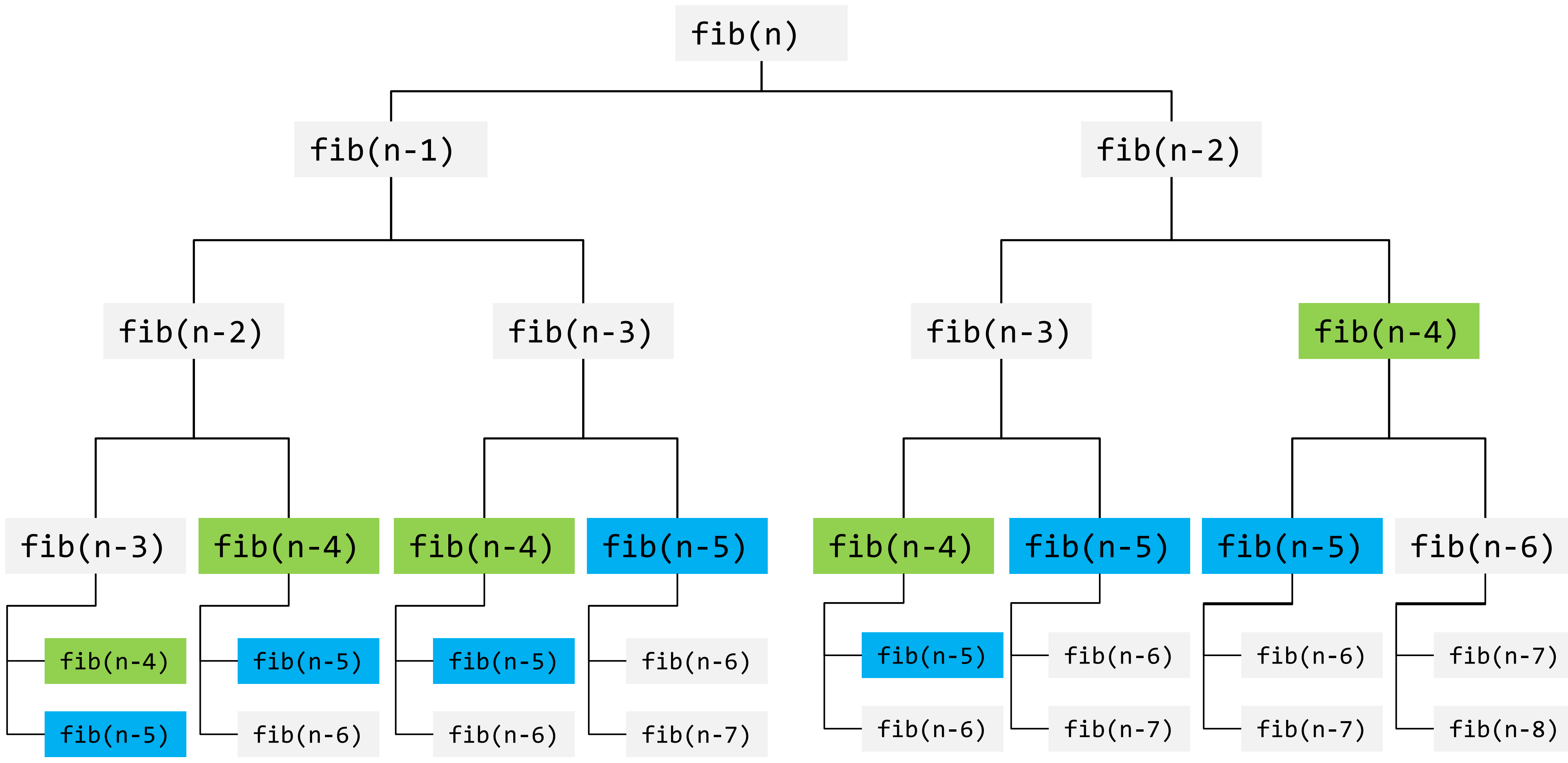
$$\forall n \geq 2, f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0, f(1) = 1$$

- **A problem is solved by breaking it into overlapping subproblems.**

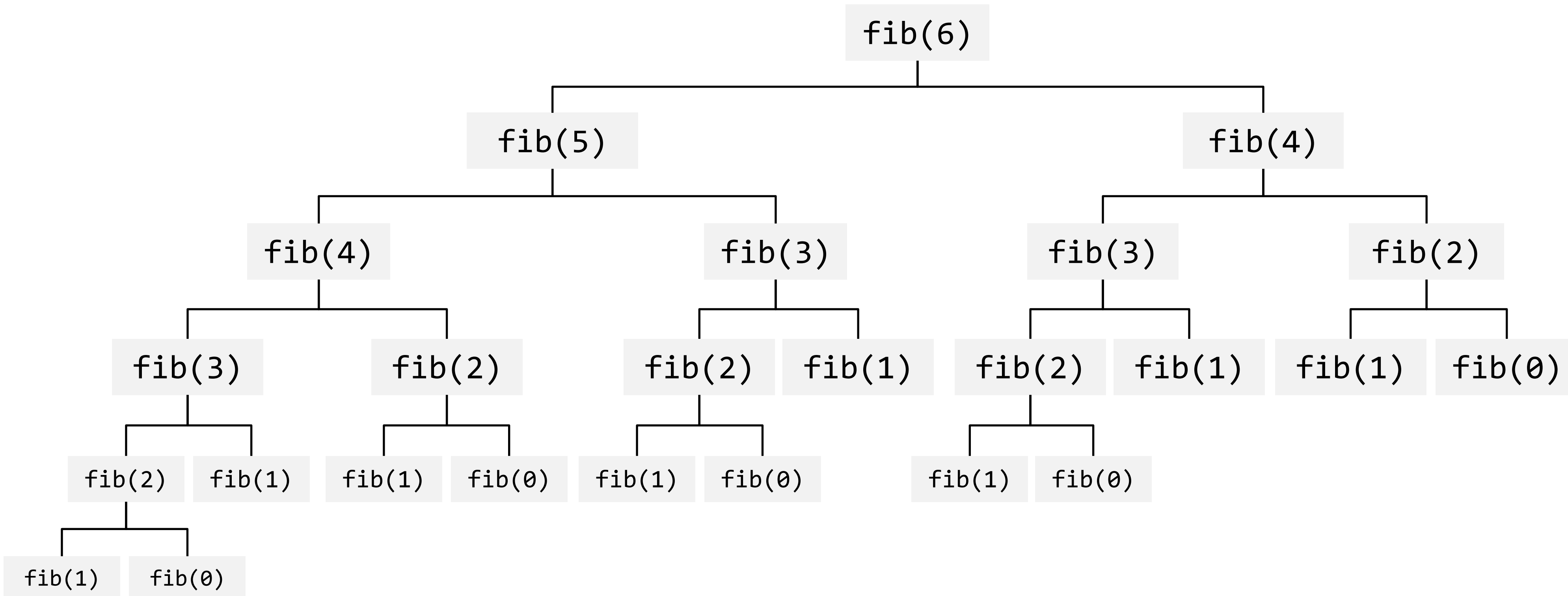
```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

How many fib values will be calculated?

Redundant Calculations



Suppose n=6



The number of fib values calculated:

$$T(2) = 3$$
$$T(3) = 5$$
$$T(4) = T(2) + T(3) + 1 = 9$$
$$T(5) = T(3) + T(4) + 1 = 15$$
$$T(6) = T(4) + T(5) + 1 = 25$$

Memo-based Optimization

```python
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n == 0 or n == 1:
        return n
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]
```

```
#: python fib.py
n=20  |  naïve recursion :  0.00214s  |  memoized:  0.00001s
n=30  |  naïve recursion :  0.24866s  |  memoized:  0.00002s
n=35  |  naïve recursion :  2.75685s  |  memoized:  0.00002s
n=40  |  naïve recursion : 30.53811s  |  memoized:  0.00002s
```

Iterative Implementation

```python
def fib(n):
    if n == 0:
        return 0
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Dynamic Programming

- **Dynamic Programming is a method for solving complex problems by breaking them down into overlapping subproblems that exhibit optimal substructure.**
- **Two main techniques for dynamic programming:**
  - Memoization (Top-down) – add caching to recursion.
  - Tabulation (Bottom-up) – fill a table iteratively.

Steps of Dynamic Programming

- ❑Characterize the structure of the optimal solution.
- ❑Define the DP state dp[i]: what does it mean?
- ❑Write a recurrence relation.
- ❑Initialize base cases.
- ❑Compute answers iteratively or recursively with memorization.

2. Coin Problem

Coin Problem

- Given coins of denominations [a, b, c], and an amount n, find the minimum number of coins to make up the amount. If not possible, return -1.
- For example, if the denominations are [1,2,5], and the amount is 11, the optimal solution is 5+5+1.

Canonical Coin System

- Using a greedy algorithm can find the optimal solution.
- However, not all coin system is canonical, e.g., [1,3,4].
  - 6=4+1+1, 6=3+3

```python
def greedy_coin(amount, coins=[5, 2, 1]):
    count_dict = {}
    for coin in coins:
        count = amount // coin
        count_dict[coin] = count
        amount %= coin
    return count_dict
```

Recurrence Relation

$$dp(x) = \begin{cases} \min\limits_{c \in coins} \left(dp(x-c)+1\right), & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ \infty, & \text{otherwise} \end{cases}$$

Implementation: Recursive Approach

```
def dp_coin_rec(coins, amount):
    if amount == 0:
        return 0
    if amount < 0:
        return float('inf')

    min = float('inf')
    for coin in coins:
        res = dp_coin_rec(coins, amount - coin)
        if res != float('inf'):
            min = min(min, res + 1)

    return min if min != float('inf') else -1
```

Memo-based Optimization

```
def dp_coin_memo(coins, amount):
    memo = {}
    def dp(n):
        if n == 0:
            return 0
        if n < 0:
            return float('inf')
        if n in memo:
            return memo[n]
        min = float('inf')
        for coin in coins:
            res = dp(n - coin)
            if res != float('inf'):
                min = min(min, res + 1)
        memo[n] = min
        return memo[n]
    result = dp(amount)
    return result if result != float('inf') else -1
```

When amount = 6, coins = [1,3,4]

```
dp(6) = min(inf, dp(5)+1, dp(3)+1, dp(2)+1)

dp(5) = min(inf, dp(4)+1, dp(2)+1, dp(1)+1)

dp(4) = min(inf, dp(3)+1, dp(1)+1, dp(0)+1)

dp(3) = min(inf, dp(2)+1, dp(0)+1, inf)

dp(2) = min(inf, dp(1)+1, inf, inf)

dp(1) = min(inf, dp(0)+1, inf, inf)

dp(0) = 0
```

When amount = 6, coins = [1,3,4]

```
dp(6) = min(inf, 3, 2, 3)

dp(5) = min(inf, 2, 3, 2)

dp(4) = min(inf, 2, 2, 1)

dp(3) = min(inf, 3, 1, inf)

dp(2) = min(inf, 2, inf, inf)

dp(1) = min(inf, 1, inf, inf)

dp(0) = 0
```

Iterative Update: When amount = 6, coins = [1,3,4]

coin = 1:

dp[0]	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]
0	1	2	3	4	5	6

coin = 3:

dp[0]	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]
0	1	2	1	2	3	2

dp[3] = min(dp[3], dp[3-3]+1) = 1
dp[4] = min(dp[4], dp[4-3]+1) = 2
dp[5] = min(dp[5], dp[5-3]+1) = 3
dp[6] = min(dp[6], dp[6-3]+1) = 2

Iterative Update: When amount = 6, coins = [1,3,4]

coin = 3:

dp[0]	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]
0	1	2	1	2	3	2

coin = 4:

dp[0]	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]
0	1	2	1	1	2	2

dp[4] = min(dp[4], dp[4-4]+1) = 1
dp[5] = min(dp[5], dp[5-4]+1) = 2
dp[6] = min(dp[6], dp[6-4]+1) = 2

Implementation: Iterative Approach

```python
def dp_coin_iter(amount, coins):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)
    if dp[amount] != float('inf'):
        return dp[amount]
    else:
        return -1
```

Performance Benchmark

```
#: python coin.py
  Amount |  Iterative (s) |    Memo (s) |   Recursive (s)
------------------------------------------------------------
      10 |       0.000031 |    0.000045 |        0.000306
      20 |       0.000031 |    0.000074 |        0.021924
      50 |       0.000035 |    0.000085 |             nan
     100 |       0.000068 |    0.000168 |             nan
     500 |       0.000359 |    0.000926 |             nan
```

3. Longest Common Subsequence

Longest Common Sequence

- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguously.
- Given two strings text1 and text2, find the length of their longest common subsequence which appears in both strings.

text1: fundamental

text2: fudan

f	u	n	d	a	m	e	n	t	a	l

f	u		d	a			n			

Recurrence Relation

$$dp(x) = \begin{cases} 0, & \text{if} \quad i < 0 \; or \; j < 0 \\ dp(i-1, j-1) + 1, & \text{if} \; text1[i] = text2[j] \\ max\big(dp(i-1, j), dp(i, j-1)\big), & otherwise \end{cases}$$

Algorithm: Recursive 1

```
def lcs(text1, text2, i=None, j=None):
    if i is None: i = len(text1) - 1
    if j is None: j = len(text2) – 1
    if i < 0 or j < 0:
        return 0

    if text1[i] == text2[j]:
        return lcs(text1, text2, i - 1, j - 1) + 1
    else:
        return max(
            lcs(text1, text2, i - 1, j),
            lcs(text1, text2, i, j - 1)
        )
```

Memo-based Optimization

```python
def lcs(text1, text2, i=None, j=None, memo=None):
    if memo is None:
        memo = {}
    if i is None:
        i = len(text1) - 1
    if j is None:
        j = len(text2) - 1
    if (i, j) in memo:
        return memo[(i, j)]
    if i < 0 or j < 0:
        return 0
    if text1[i] == text2[j]:
        memo[(i, j)] = lcs(text1, text2, i - 1, j - 1, memo) + 1
    else:
        memo[(i, j)] = max(
            lcs(text1, text2, i - 1, j, memo),
            lcs(text1, text2, i, j - 1, memo)
        )
    return memo[(i, j)]
```

Iterative Analysis

	f	u	n	d	a	m	e	n	t	a	l
f	1										
u		2	2								
d				3							
a					4	4	4	4			
n								5			

```
if text1[i] == text2[j]:
    dp[i][j] = dp[i - 1][j - 1] + 1
```

Iterative Analysis

	f	u	n	d	a	m	e	n	t	a	l
f	1	1	1	1	1	1	1	1	1	1	1
u	1	2	2	2	2	2	2	2	2	2	2
d	1	2	2	3	3	3	3	3	3	3	3
a	1	2	2	3	4	4	4	4	4	4	4
n	1	2	3	3	4	4	4	5	5	5	5

```
if text1[i] != text2[j]:
    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

Algorithm: Iterative

```
def lcs(text1, text2):
    m, n = len(text1), len(text2)
    # init an extra row and column with value 0.
    dp = [[0] * (n+1) for _ in range(m+1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:  # take max from left or top
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

With Python lru_cache Decorator

- ❑lru_cache remembers the results of function calls.
- ❑If the function is called again with the same arguments, Python returns the stored result instantly.

```
from functools import lru_cache

def lcs(text1, text2):
    @lru_cache(None)
    def helper(i, j):
        if i < 0 or j < 0:
            return 0
        if text1[i] == text2[j]:
            return helper(i - 1, j - 1) + 1
        return max(helper(i - 1, j), helper(i, j - 1))
    return helper(len(text1) - 1, len(text2) - 1)
```

Performance Benchmark

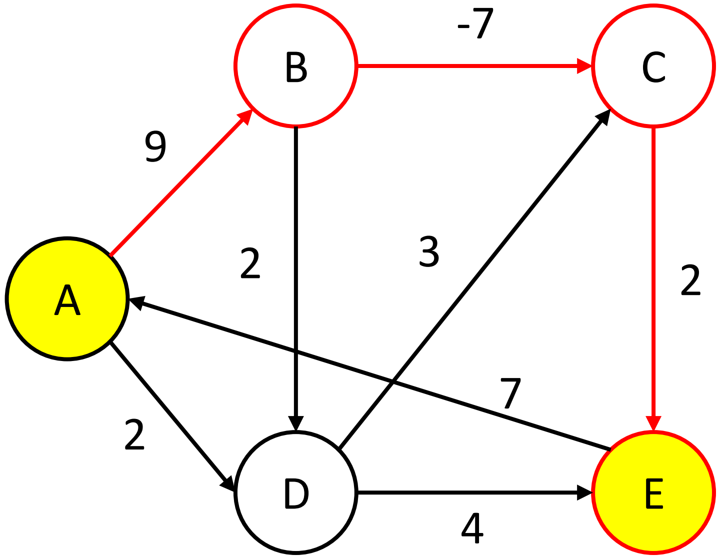
```
python lcs.py
  Length |  Iterative |       Memo |      Cache |    Recursive
------------------------------------------------------------
       5 |   0.000026 |   0.000017 |   0.000033 |     0.000012
       7 |   0.000031 |   0.000021 |   0.000022 |     0.000017
      12 |   0.000069 |   0.000068 |   0.000048 |          nan
      15 |   0.000081 |   0.000087 |   0.000064 |          nan
```

Be Careful with lru_cache

```python
from functools import lru_cache

x = 10
@lru_cache()
def f(a):
    return a + x
print(f(1))  # 11

x = 20
print(f(1))  # still 11, not 21
```

4. Revisit the Shortest Path Problem

The Shortest Path Problem

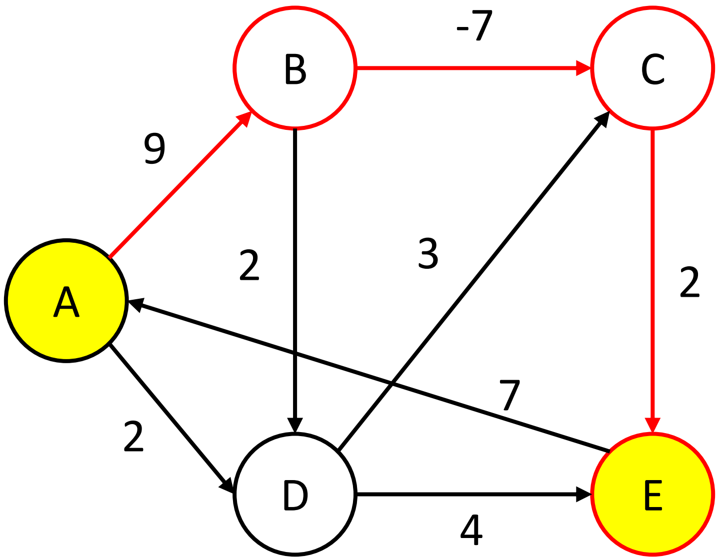


$$d_k(v) = \min\left(d_{\mathrm{k-1}}(\mathrm{v}), \min_{(u,v)\in E} d_{k-1}(u) + w(u,v)\right)$$

Bellman-Ford Algorithm: Recursive

```
def bellman_ford_recur(V, edges, src):
    def shortest(v, k):
        if v == src:
            return 0
        if k == 0:
            return float('inf')

        best = shortest(v, k - 1)
        for u, vv, w in edges:
            if vv == v:
                best = min(best, shortest(u, k - 1) + w)
        return best

    dist = [shortest(v, V - 1) for v in range(V)]
    return dist
```

Iterative Analysis



$$d_k(v) = \min_{(u,v)\in E}(d_{k-1}(u) + w(u,v))$$

$$d_1(B) = 9$$
$$d_1(D) = 2$$
$$d_2(C) = 2$$
$$d_2(E) = 6$$
$$d_3(E) = 4$$

Bellman-Ford Algorithm: Iterative

```python
def bellman_ford_iter(V, edges, src):
    INF = float('inf')
    dist = [INF] * V
    dist[src] = 0

    for _ in range(V - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            print("Illegal: negative-weight cycles.")
            return None

    return dist
```

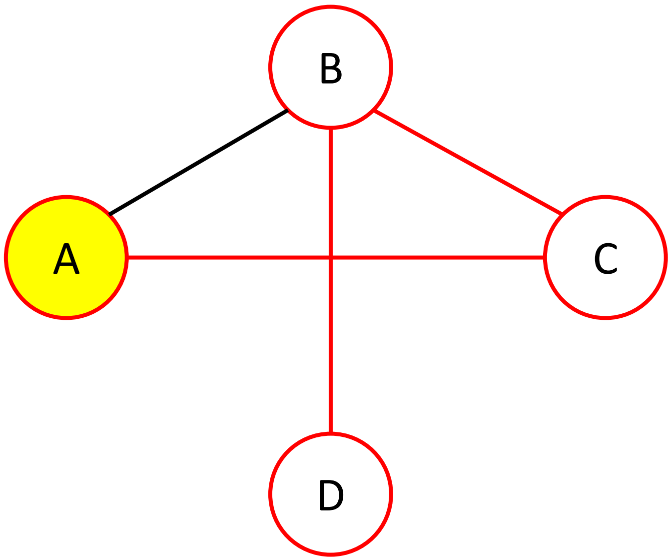
Complexity Comparison

- **Bellman Ford:** $O(V \times E)$
- **Dijkstra:** $O(E)$
  - If further consider the cost of sorting the shortest path with min heap
  - $(E + V) \times logV$
  - E for push elements, V for pop.

5. Travels Salesman Problem

Hamiltonian Path

- ❑A path that visits each vertex exactly once on an unweighted graph.
- ❑Hamiltonian cycle further requires returning to the original node.



Solve the Problem via Dynamic Programming

- If there is a path visiting all vertices in set S and ending at vertex v, $\boldsymbol{dp(S,v) = true}$
- **Base:** $\forall \boldsymbol{v} \in \boldsymbol{V},\ \boldsymbol{dp(\{v\},v) = true}$
- **Answer:** $\bigvee_{v} \boldsymbol{dp(V,v)}$
- **Recurrence relation:**

$$dp(S,v) = \bigvee_{u \in S,(u,v) \in E} dp(S\{v\},u)$$

Represent Vertex Sets with Bitmasks

mask	D	C	B	A
0001				T
0010			T	
0011			T	T
0100		T		
0101		T		T
0110		T	T	
0111		T	T	T
1000	T			
1001	T			T
1010	T		T	
1011	T		T	T
1100	T	T		
1101	T	T		T
1110	T	T	T	
1111	T	T	T	T

- 1: the vertex is in the set
- 0: the vertex is not in the set

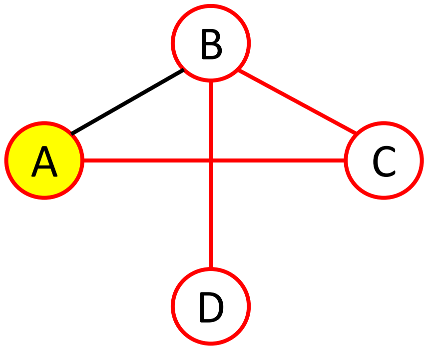
Implementation with Python: Recursive

```python
def hamiltonian_path(graph):
    n = len(graph)
    ALL_VISITED = (1 << n) - 1  # bitmask with all nodes visited
    @lru_cache(None)
    def dfs(mask, v):
        if mask == (1 << v):  # Base case: only u is visited
            return True
        # Try to reach u from any previous vertex v
        for u in range(n):
            if mask & (1 << u) and u != v and graph[u][v]:
                prev_mask = mask ^ (1 << v)
                if dfs(prev_mask, u):
                    return True
        return False

    for end in range(n):  # Try all possible end vertices
        if dfs(ALL_VISITED, end):
            return True
    return False
```

Implementation with Python

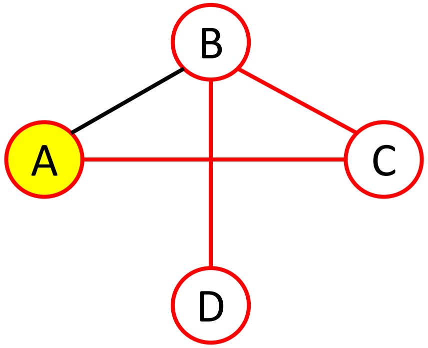
```python
def hamiltonian_path(graph):
    n = len(graph)

    dp = [[False] * n for _ in range(1 << n)] # 2ⁿ * n
    parent = [[-1] * n for _ in range(1 << n)]

    for i in range(n): # Initialize
        dp[1 << i][i] = True

    for mask in range(1, 1 << n):
        for u in range(n):
            if dp[mask][u]:  # If there's a path to 'u'
                for v in range(n):
                    if graph[u][v] and not (mask & (1 << v)):
                        dp[mask | (1 << v)][v] = True
                        parent[mask | (1 << v)][v] = u
```

Example: Init



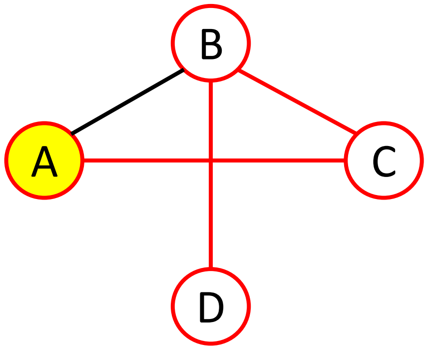
mask	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	F	F
0100	F	T	F	F
0101	F	F	F	F
0110	F	F	F	F
0111	F	F	F	F
1000	T	F	F	F
1001	F	F	F	F
1010	F	F	F	F
1011	F	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



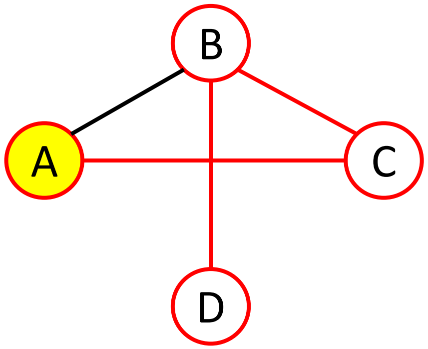
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	F
0100	F	T	F	F
0101	F	T(A)	F	F
0110	F	F	F	F
0111	F	F	F	F
1000	T	F	F	F
1001	F	F	F	F
1010	F	F	F	F
1011	F	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



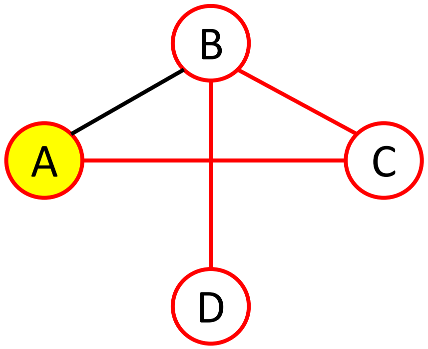
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	F
0110	F	T(B)	F	F
0111	F	F	F	F
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	F	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



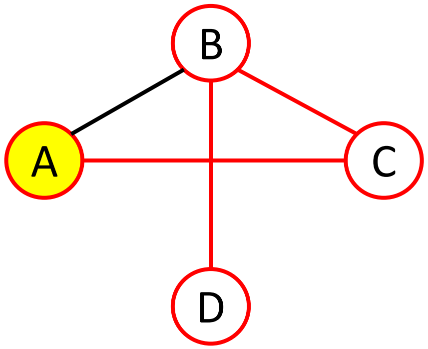
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	F
0110	F	T(B)	F	F
0111	F	T(A/B)	F	F
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



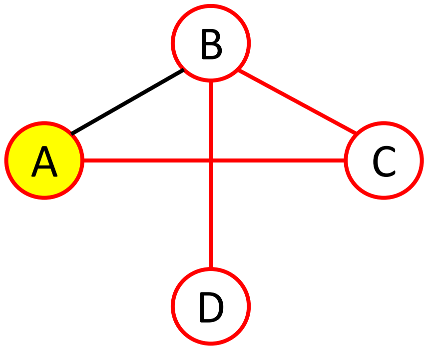
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	F	F
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



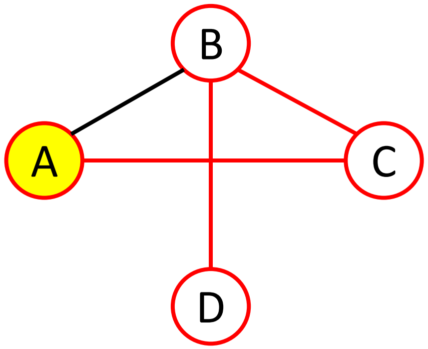
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	F
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	F	F	F	F
1111	F	F	F	F

Example



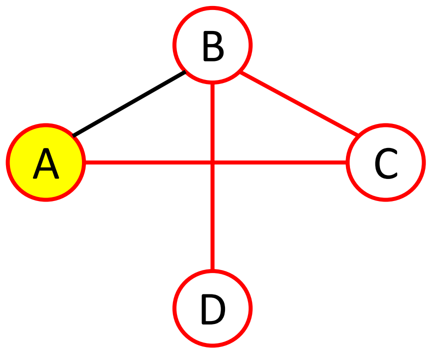
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	F	F	F
1111	F	F	F	F

Example



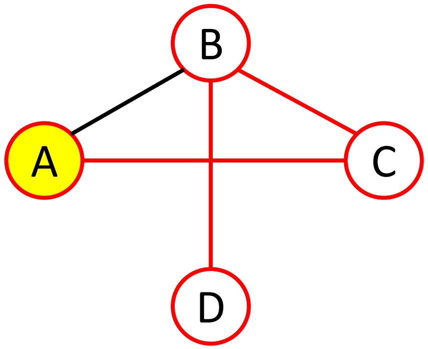
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	F	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	F	F	F
1111	T(B)	F	F	F

Example



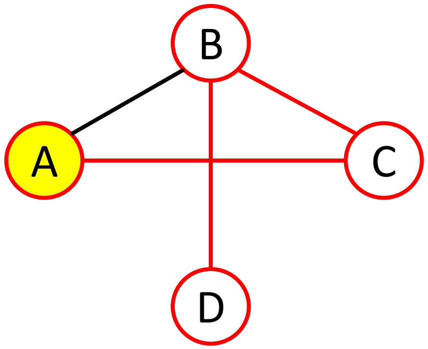
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	T(D)	F
1011	T(B)	F	F	F
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	F	F	F
1111	T(B)	F	F	F

Example



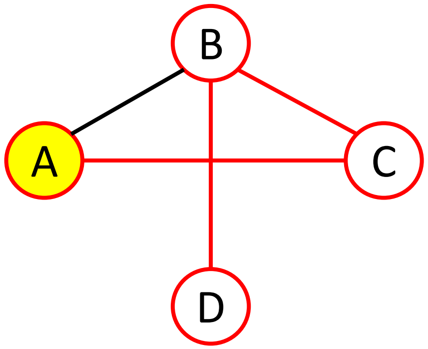
	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	T(D)	F
1011	T(B)	F	F	T(B)
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	T(B)	F	F
1111	T(B)	F	F	F

Example



	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	T(D)	F
1011	T(B)	F	F	T(B)
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	T(B)	F	F
1111	T(B)	T(A)	F	F

Example



	D	C	B	A
0001	F	F	F	T
0010	F	F	T	F
0011	F	F	T(A)	T(B)
0100	F	T	F	F
0101	F	T(A)	F	T(C)
0110	F	T(B)	T(C)	F
0111	F	T(A/B)	T(A/C)	T(B/C)
1000	T	F	F	F
1001	F	F	F	F
1010	T(B)	F	T(D)	F
1011	T(B)	F	F	T(B)
1100	F	F	F	F
1101	F	F	F	F
1110	T(B)	T(B)	F	F
1111	T(B)	T(A)	F	T(C)

Path Reconstruction

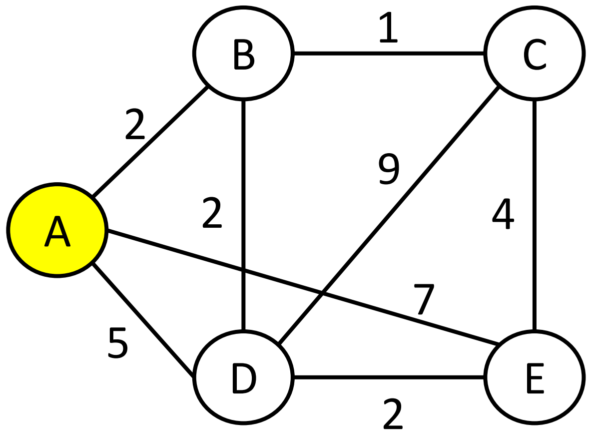
```
full_mask = (1 << n) - 1
for end in range(n):
    if dp[full_mask][end]:  # found Hamiltonian Path
        # reconstruct path
        path = []
        mask = full_mask
        cur = end
        while cur != -1:
            path.append(cur)
            prev = parent[mask][cur]
            mask ^= (1 << cur) # set the bit mast of cur to 0
            cur = prev
        path.reverse()
```

Exercise

- Hamiltonian cycle requires returning to the original node.
- Modify the program to detect Hamiltonian cycle.

Travelling Salesman Problem (TSP)

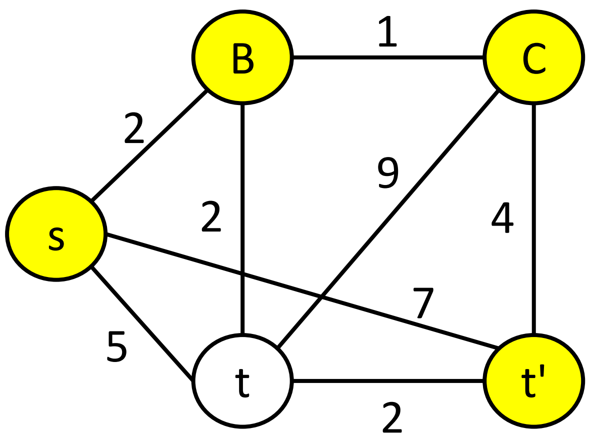
- Similar to Hamiltonian cycle but concerns weighted graph
- Find the shortest path that visit each node exactly once (and returns to the original node)



Solve TSP with Dynamic Programming

- Supposing the final solution is {s, …, u, v}

$$Cost(S,t) = \begin{cases} \min\limits_{u \in S}\left(Cost(S\backslash \mathrm{v}, u) + w(u,v)\right) & if\,|S| > 1 \\ 0 & otherwise \end{cases}$$



Python: Recursive

```python
def tsp_recursive(graph):
    n = len(graph)
    ALL_VISITED = (1 << n) - 1  # bitmask with all cities visited
    @lru_cache(None)
    def dfs(mask, u):
        if mask == 1:
            return graph[0][u] if u != 0 else 0
        best = math.inf
        prev_mask = mask & ~(1 << u)
        m = prev_mask
        while m:
            v = (m & -m).bit_length() - 1 # find the last 1
            cost = dfs(prev_mask, v) + graph[v][u]
            best = min(best, cost)
            m &= m – 1 # change the last 1 to 0
        return best
    full_mask = (1 << n) - 1
    res = math.inf
    for u in range(1, n):
        res = min(res, dfs(full_mask, u) + graph[u][0])
    return res
```

Python: Iterative

```
def tsp(graph):
    n = len(graph)
    INF = math.inf
    dp = [[INF] * n for _ in range(1 << n)]
    parent = [[-1] * n for _ in range(1 << n)]

    dp[1][0] = 0 # Start at node 0

    for mask in range(1 << n):
        for u in range(n):
            if not (mask & (1 << u)):
                continue  # skip if u not in mask
            for v in range(n):
                if mask & (1 << v):
                    continue  # v already visited
                new_mask = mask | (1 << v)
                new_cost = dp[mask][u] + graph[u][v]
                if new_cost < dp[new_mask][v]:
                    dp[new_mask][v] = new_cost
                    parent[new_mask][v] = u
```

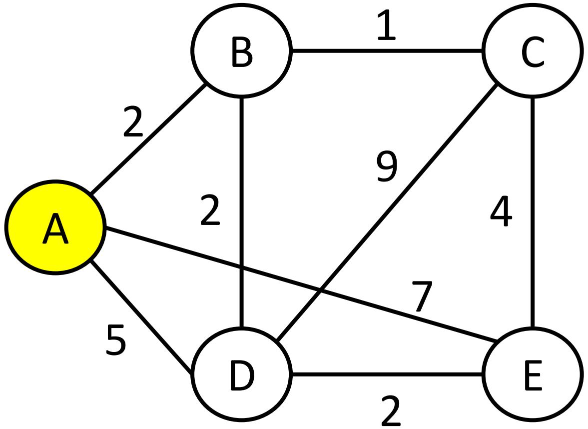
Complexity

- ❑The number of bitmask is exponential to the number of vertex.
- ❑Complexity: $O(2^n \times n^2) = O(2^n)$
- ❑The problem cannot be solved in polynomial time.

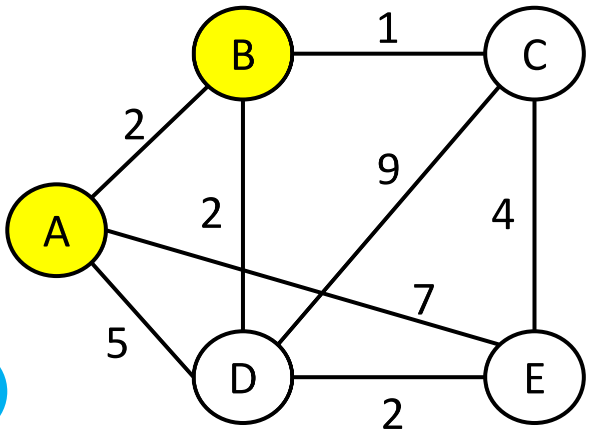
Solve TSP with Greedy Algorithm

❑Tradeoff between the precision and efficiency.

- ➢Each time select an edge to a new node with the smallest weight.
- ➢Fallback if: 1) the node forms a loop; 2) unable to reach more nodes.

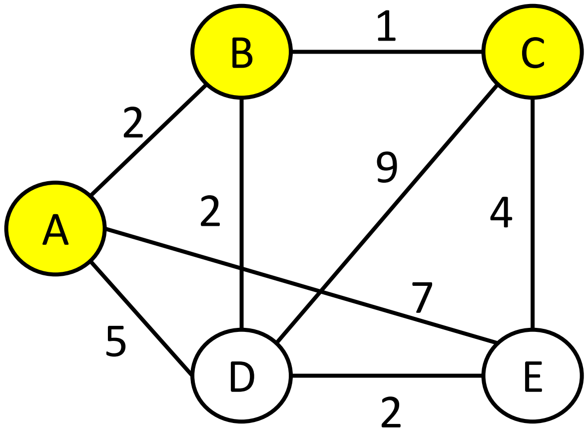


	A	B	C	D	E
Visited?	Y	N	N	N	N
Distance	0	2	∞	5	7

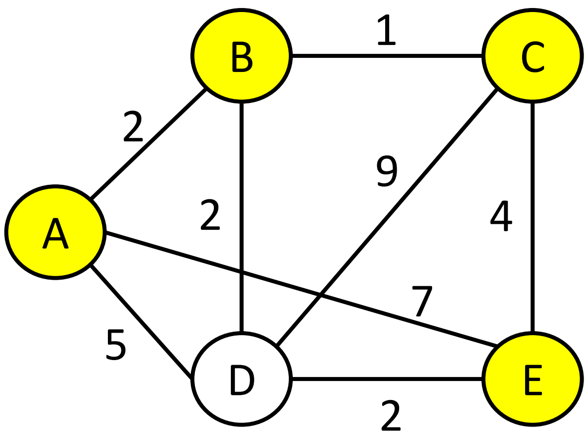


	A	B	C	D	E
Visited?	Y	Y	N	N	N
Distance	0	0	1	2	∞

Solve TSP with Greedy Algorithm



	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	0	0	0	9	4



	A	B	C	D	E
Visited?	Y	Y	Y	N	N
Distance	0	0	0	2	0

Cost: 2+1+4+2+5 = 14