CIIC 4025 Analysis and Design of Algorithms

WILFREDO LUGO, PHD

Dynamic Programming

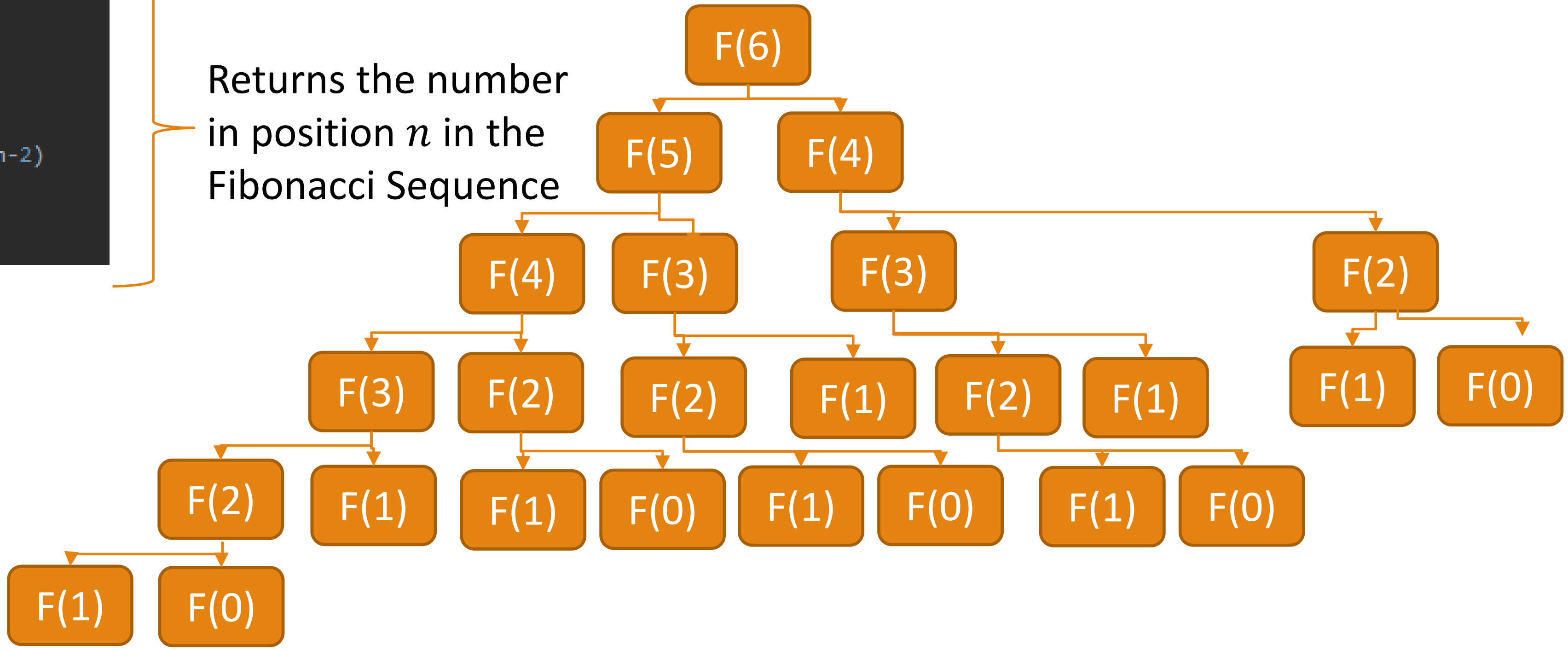
Fibonacci Sequence

- $F = \{0,1,1,2,3,5,8,13,21,\ldots\}$

```python
def Fibonacci(n):
    if n <= 1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

print(Fibonacci(6))
```

Returns the number in position n in the Fibonacci Sequence

Returns 8, since 8 is the element in the 6th position.



Dynamic Programming

```python
def Fibonacci(n):
    if n <= 1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

print(Fibonacci(6))
```

$$T(n) = \begin{cases} 1, if\ n \leq 1 \\ T(n-1) + T(n-2) + c \end{cases}$$

Assumption A:

$$T(n-1) \approx T(n-2), thus\ T(n) = 2T(n-2) + c$$

Using Master Theorem we then have: $a = 2, b = 2, k = 0$

$$T(n) = O\left(2^{\frac{n}{2}}\right)$$

Assumption B:

$$T(n-2) \approx T(n-1), thus\ T(n) = 2T(n-1) + c$$

Using Master Theorem we then have: $a = 2, b = 1, k = 0$

$$T(n) = O(2^n)$$

$$T(n) = aT(n-b) + f(n)$$

$$a > 1, b > 0\ and\ f(n) = \Theta\left(n^k\right)\ where\ k \geq 0$$

$$Case\ 1: if\ a < 1, then\ O\left(n^k\right)\ or\ O(f(n))$$

$$Case\ 2: if\ a = 1, then\ O\left(n^{k+1}\right)\ or\ O(n * f(n))$$

$$Case\ 3: if\ a > 1, then\ O(n^k a^{\frac{n}{b}})\ \text{or}\ O(f(n) a^{\frac{n}{b}})$$

New Video

Dynamic Programming

Fibonacci Sequence

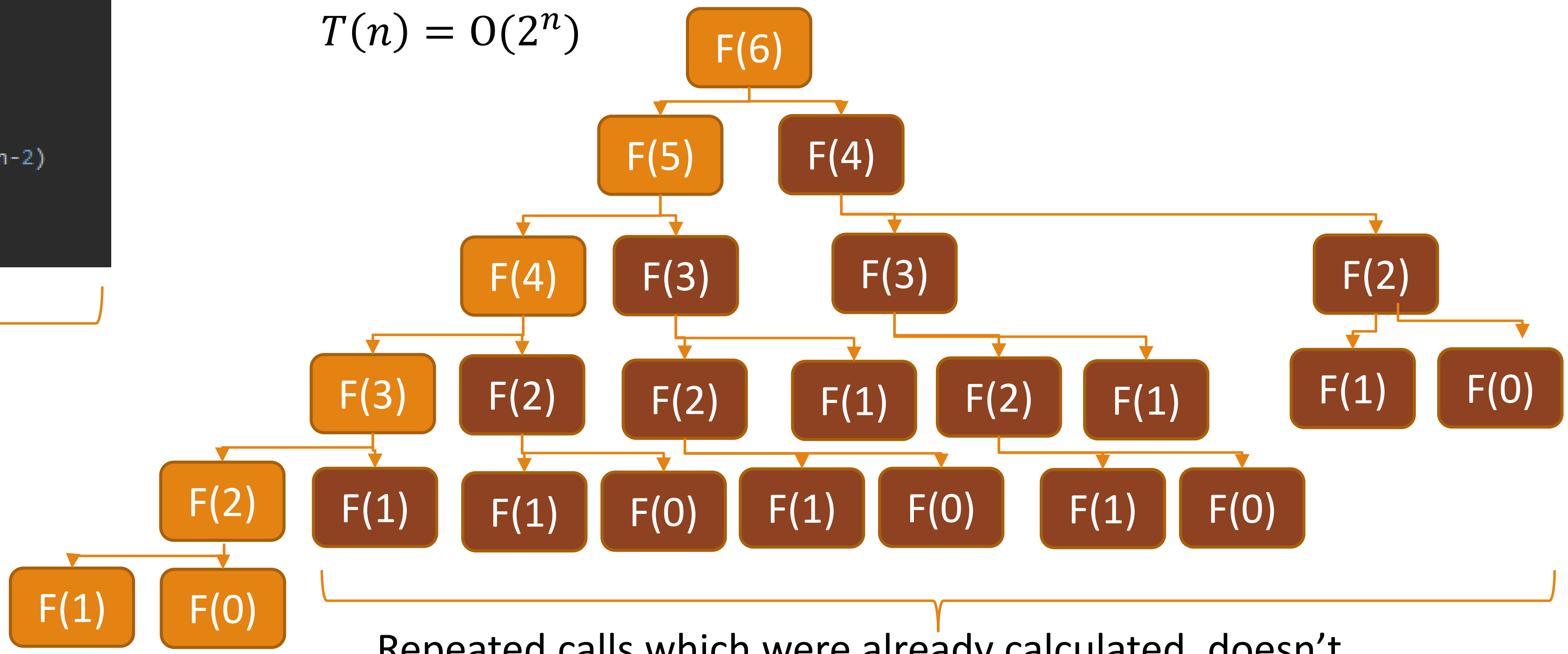
- $F = \{0,1,1,2,3,5,8,13,21,\ldots\}$

$$T(n) = \begin{cases} 1, if\ n \leq 1 \\ T(n-1) + T(n-2) + c \end{cases}$$

```python
def Fibonacci(n):
    if n <= 1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

print(Fibonacci(6))
```

Returns the number in position n in the Fibonacci Sequence

$$T(n) = O(2^n)$$

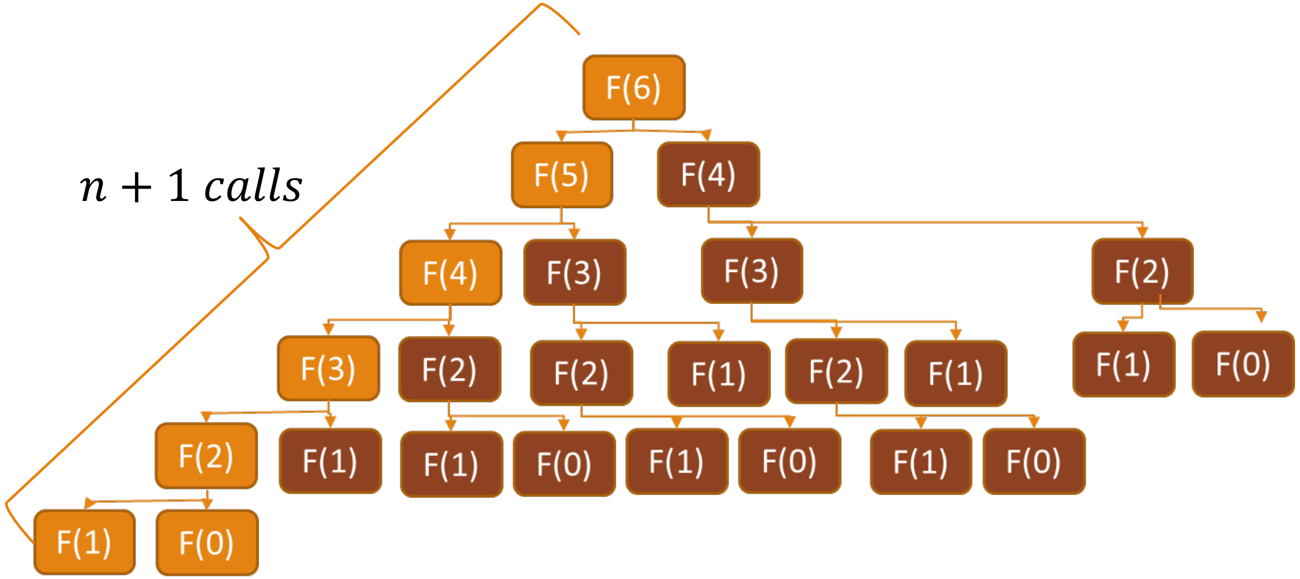


Repeated calls which were already calculated, doesn't makes sense to re-calculate them again.

Dynamic Programing

- Memoization
  - Optimization technique use primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

```python
def Fibonacci(indexPos,n):
    if indexPos[n] != -1:
        return indexPos[n]
    elif n <= 1:
        indexPos[n] = n
        #print(indexPos)
        return n
    else:
        n_1 = indexPos[n-1]
        n_2 = indexPos[n-2]
        if n_1 == -1:
            n_1 = Fibonacci(indexPos,n-1)
            indexPos[n-1] = n_1
        if n_2 == -1:
            n_2 = Fibonacci(indexPos,n-2)
            indexPos[n-2] = n_2
        return n_1 + n_2
n=6
indexPos = np.zeros([n+1,1])
indexPos=indexPos-1
value1 = Fibonacci(indexPos,6)
```



Dynamic Programing

```python
def Fibonacci(n):
    if n <= 1:
        return n
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

print(Fibonacci(6))
```

Original Fibonacci: $O(2^n)$

```
4.622299999998414e-05
0.00030893900000000807
value 8 vs 8
```

```python
def Fibonacci(indexPos,n):
    if indexPos[n] != -1:
        return indexPos[n]
    elif n <= 1:
        indexPos[n] = n
        #print(indexPos)
        return n
    else:
        n_1 = indexPos[n-1]
        n_2 = indexPos[n-2]
        if n_1 == -1:
            n_1 = Fibonacci(indexPos,n-1)
            indexPos[n-1] = n_1
        if n_2 == -1:
            n_2 = Fibonacci(indexPos,n-2)
            indexPos[n-2] = n_2
        return n_1 + n_2
n=6
indexPos = np.zeros([n+1,1])
indexPos=indexPos-1
value1 = Fibonacci(indexPos,6)
```

Fibonacci with Memoization: $O(n)$

New Video

Dynamic Programming

- Tabulation
  - Approach where you solve a dynamic programming problem by first filling up a table, and then compute the solution to the original problem based on the results in this table.

```python
def FibonacciTabulation(n):
    indexPos=np.zeros([n+1,1])
    if n <= 1:
        return n;
    indexPos[0] = 0
    indexPos[1] = 1
    for index in range(2,n+1):
        indexPos[index] = indexPos[index-2]+indexPos[index-1]
    return indexPos
n=6
print(FibonacciTabulation(n)[n])
```

Fills table with values (bottom up)

Get solution from table in constant time

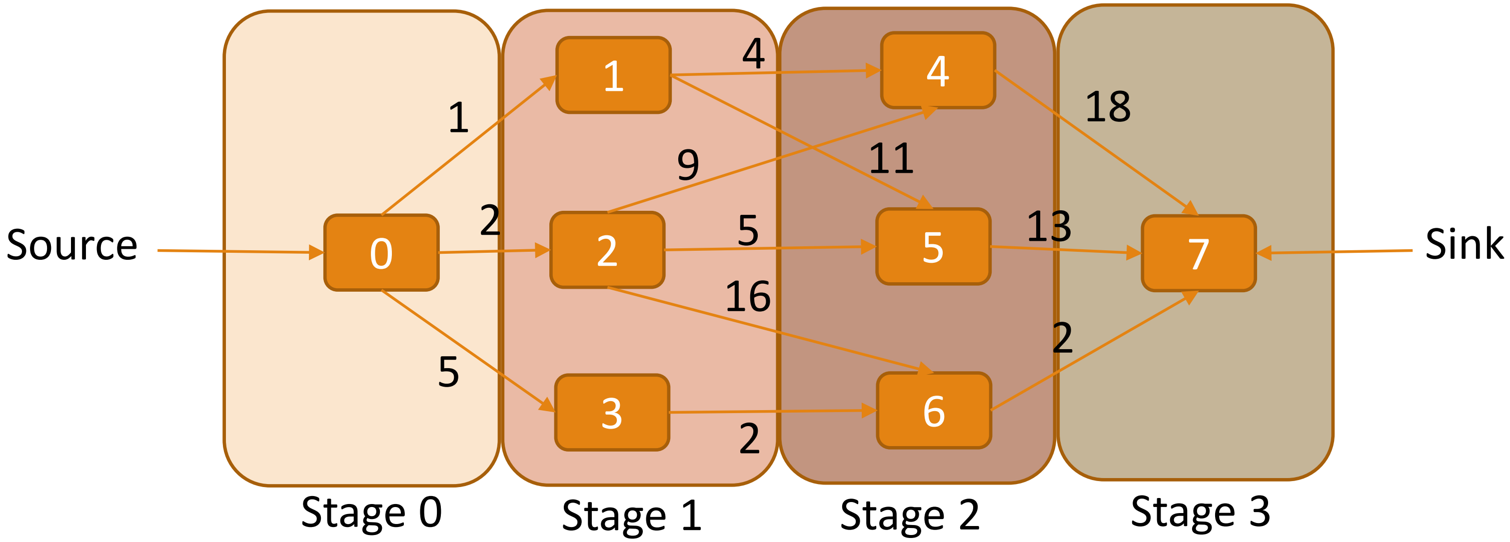
No recursion, $O(n)$

New Video

Dynamic Programming: Multistage Graph

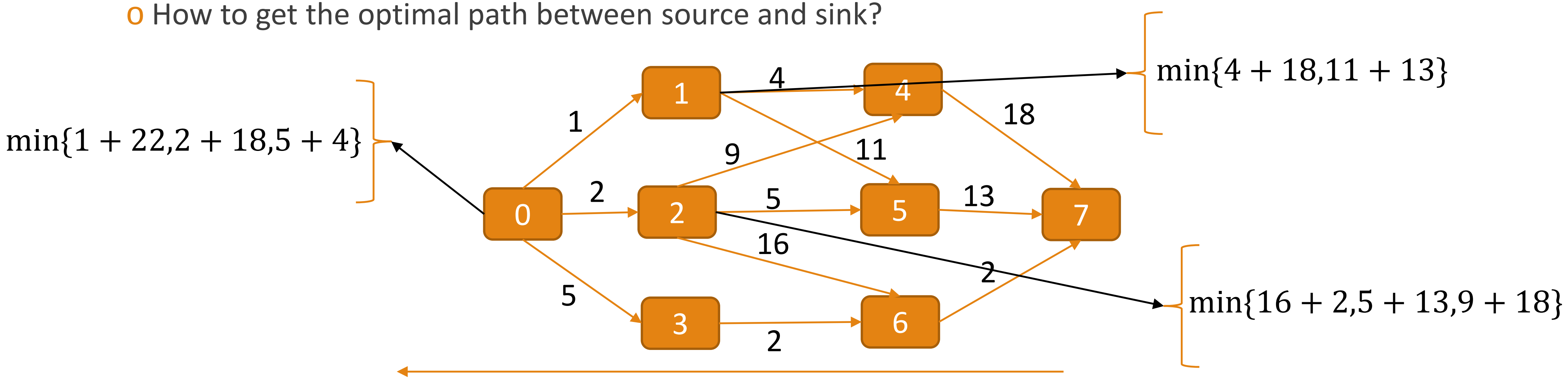
Multistage graph

- Directed graph in which the nodes can be divided into a set of stages such that all edges in one node are pointing to nodes of the next stage.



Dynamic Programming: Multistage Graph

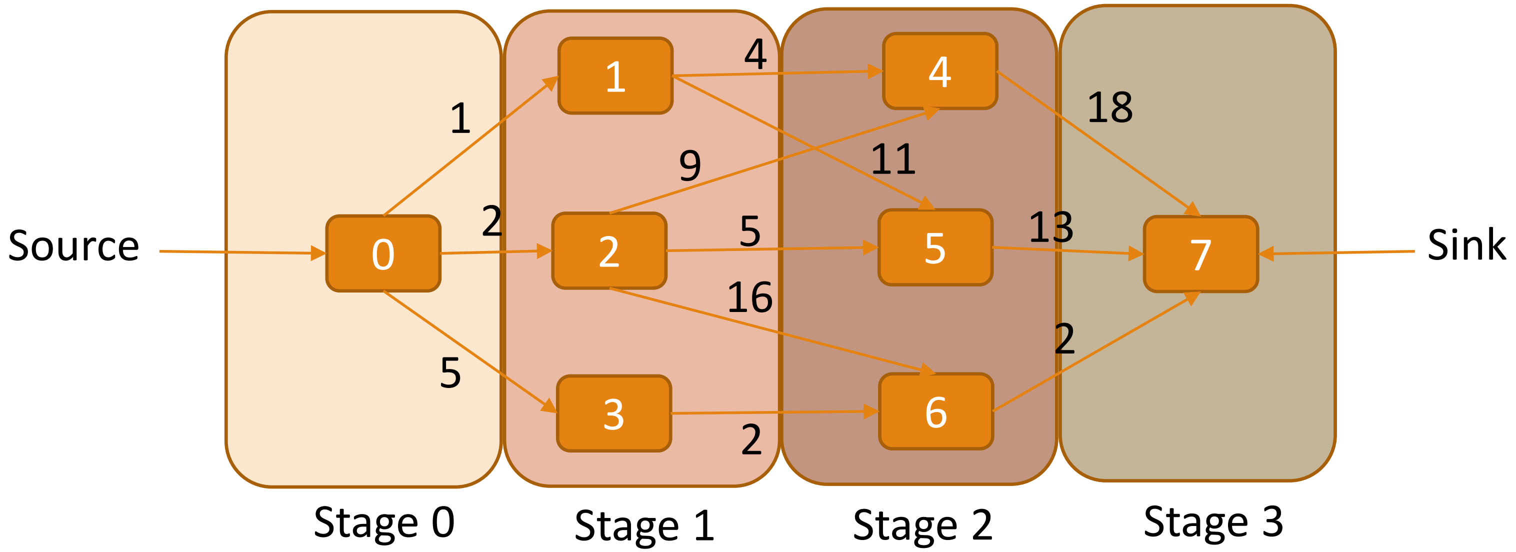
- Basic Problem:
  - How to get the optimal path between source and sink?



Nodes	0	1	2	3	4	5	6	7
Cost	9	22	18	4	18	13	2	0
Destination	3	4	5/6	6	7	7	7	7

The table is filled from right to left

Dynamic Programming: Multistage Graph



Node (Vertex)

$$Cost(i,j) = min\{c(j,l) + Cost(i+1,l)\}\ where\ j,l\ \in V\ and\ i \in S$$

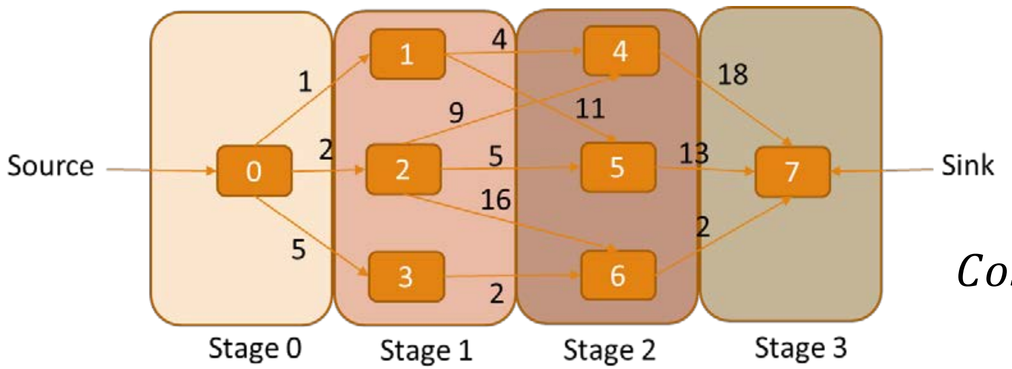
Stage

Weight between vertex j and vertex l

Ex. $Cost(1,2) = \min\{9 + Cost(2,4),\ 5 + Cost(2,5), 16 + Cost(2,6)\}$

New Video

Dynamic Programming: Multistage Graph



$$Cost(i,j) = min\{c(j,l) + Cost(i+1,l)\}\ where\ j,l\ \in V\ and\ i \in S$$

$$Cost\ Adjacency\ Matrix = \begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & & 1 & 2 & 5 & & & & \\ 1 & & & & & 4 & 11 & & \\ 2 & & & & & 9 & 5 & 16 & \\ 3 & & & & & & & 2 & \\ 4 & & & & & & & & 18 \\ 5 & & & & & & & & 13 \\ 6 & & & & & & & & 2 \\ 7 & & & & & & & & \end{array}$$

Dynamic Programming: Multistage Graph

```python
def shortestDist(graph):
    n = len(graph);
    dist = np.zeros([n,1])
    positive_infinity = float('inf')

    for i in range(n-2,-1,-1):
        dist[i] = positive_infinity

        for j in range(n):
            if graph[i][j] == 0:
                continue
            dist[i] = min(dist[i],graph[i][j] + dist[j])
    return dist[0]


graph = [[0,1,2,5,0,0,0,0],
         [0,0,0,0,4,11,0,0],
         [0,0,0,0,9,5,16,0],
         [0,0,0,0,0,0,2,0],
         [0,0,0,0,0,0,0,18],
         [0,0,0,0,0,0,0,13],
         [0,0,0,0,0,0,0,2],
         [0,0,0,0,0,0,0,0]]

print(shortestDist(graph))
```

One more array can be added to store the Path

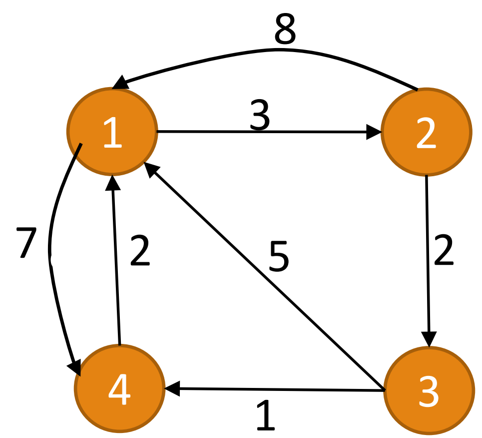
$(n-2)\ loop$

$n\ loop$

These needs to be multiplied since one is inside the other, thus $O(n^2)$

New Video

Dynamic Programming: All Pairs Shortest Path (Floyd-Warshall)



$$
\begin{array}{c c c c c}
\text{Vertex:} & 1 & 2 & 3 & 4
\end{array}
$$

$$Cost\ Adjacency\ Matrix = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} = A_0$$

$$A_1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix}$$

Pre-populate with the values for vertex 1

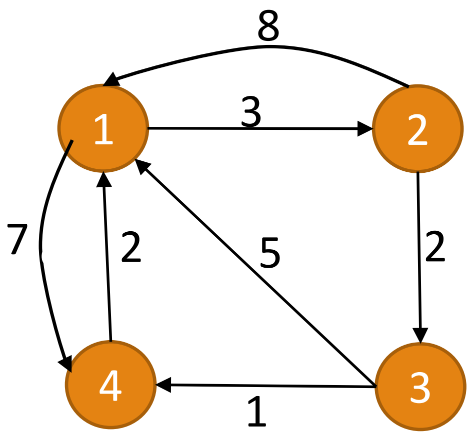
$$A_1[2,3] = \min(A_0[2,3], A_0[2,1] + A_0[1,3])$$
$$A_1[2,3] = \min(2, 8 + \infty) = 2$$

$$A_1[2,4] = \min(A_0[2,4], A_0[2,1] + A_0[1,4])$$
$$A_1[2,4] = \min(\infty, 8 + 7) = 15$$

$$A_1[3,2] = \min(A_0[3,2], A_0[3,1] + A_0[1,2])$$
$$A_1[3,2] = \min(\infty, 5 + 3) = 8$$

Dynamic Programming: All Pairs Shortest Path (Floyd-Warshall)

$$A_1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$



$$A_2 = \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & 0 & \\ & 5 & & 0 \end{bmatrix}$$

Pre-populate with the values for vertex 2 based on previous matrix

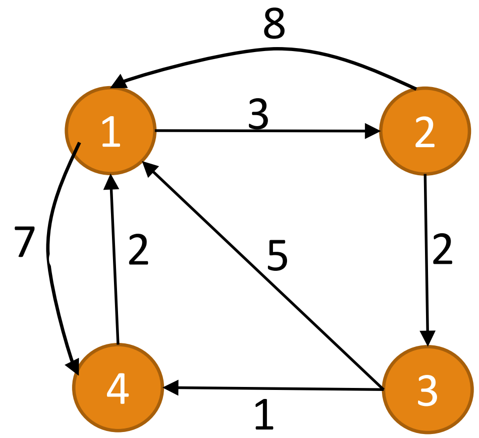
$$A_2[1,3] = \min(A_1[1,3], A_1[1,2] + A_1[2,3])$$
$$A_2[1,3] = \min(\infty, 3 + 2\ ) = 5$$

$$A_2[1,4] = \min(A_1[1,4], A_1[1,2] + A_1[2,4])$$
$$A_2[1,3] = \min(7,3 + 15\ ) = 7$$

Dynamic Programming: All Pairs Shortest Path (Floyd-Warshall)

$$A_1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$



$$A_2 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

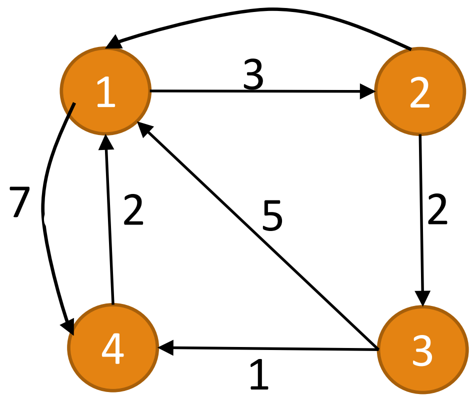
$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])\,, k > 0$$

New Video

Dynamic Programming: All Pairs Shortest Path (Floyd-Warshall)

```python
def FloydWarshal(graph):
    n = len(graph)
    A = graph
    for k in range(0,n):
        for i in range(0,n):
            for j in range(0,n):
                A[i][j] = min(A[i][j],A[i][k] + A[k][j])
    return A

pos_inf = float('inf')
graph2 = [[0,3,pos_inf,7],
        [8,0,2,pos_inf],
        [5,pos_inf,0,1],
        [2,pos_inf,pos_inf,0]]
print(FloydWarshal(graph2))
```

$O(n^3)$



$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])\,, k > 0$$

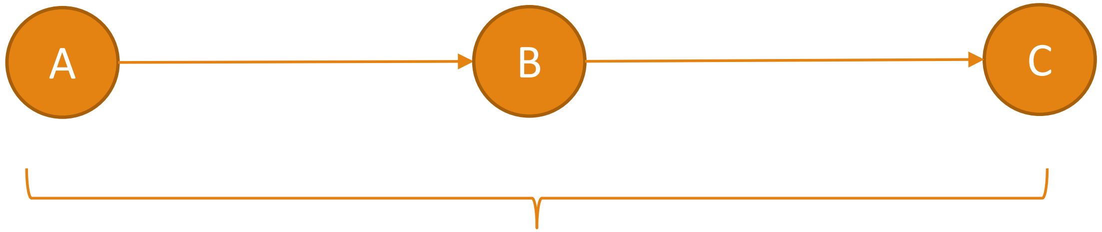
```
C:\Users\lugow\AppData\Local\Programs\Python\Python37\python.exe "D:/lu
[[0, 3, 5, 6], [5, 0, 2, 3], [3, 6, 0, 1], [2, 5, 7, 0]]

Process finished with exit code 0
```

$$A_4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

New Video

Dynamic Programming: Common Techniques

- Principle of Optimality
  - A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its sub-problems.



The optimal path from A to C is the same as the optimal path from A to B combined with the optimal path from B to C

Dynamic Programming vs Divide and Conquer

	Divide and Conquer	Dynamic Programming
Sub-problems	Problem is divided into sub-problems, sub-problems are then solved **independently** of each other and them combined to get to an overall solution.	Problem is divided into sub-problems, but the sub-problem is then used to get the solution of a bigger problem. Bigger problems are **dependent** of the solution of sub-problems to calculate their own solution.
Solution	The problem is solved one-time across all sub-problems, combination step joins all solutions.	The solution is being recalculated in an iterative way across sub-problems and on each iteration, a new sub-problem is solved.
Tabulation	No need for storing intermediate solutions since all sub-solutions are calculated at the same time.	All sub-problems solutions are being stored to be used on next steps.