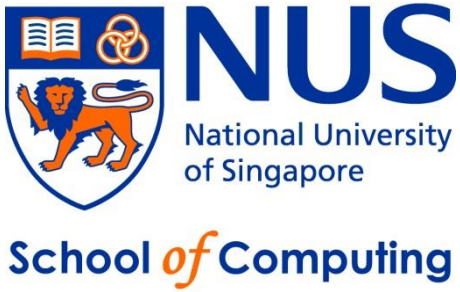
# CS2010 – Data Structures and Algorithms II

## Lecture 10 – Algorithms on DAG

chongket@comp.nus.edu.sg



# Outline

(Dynamic Programming) Algorithms on DAG

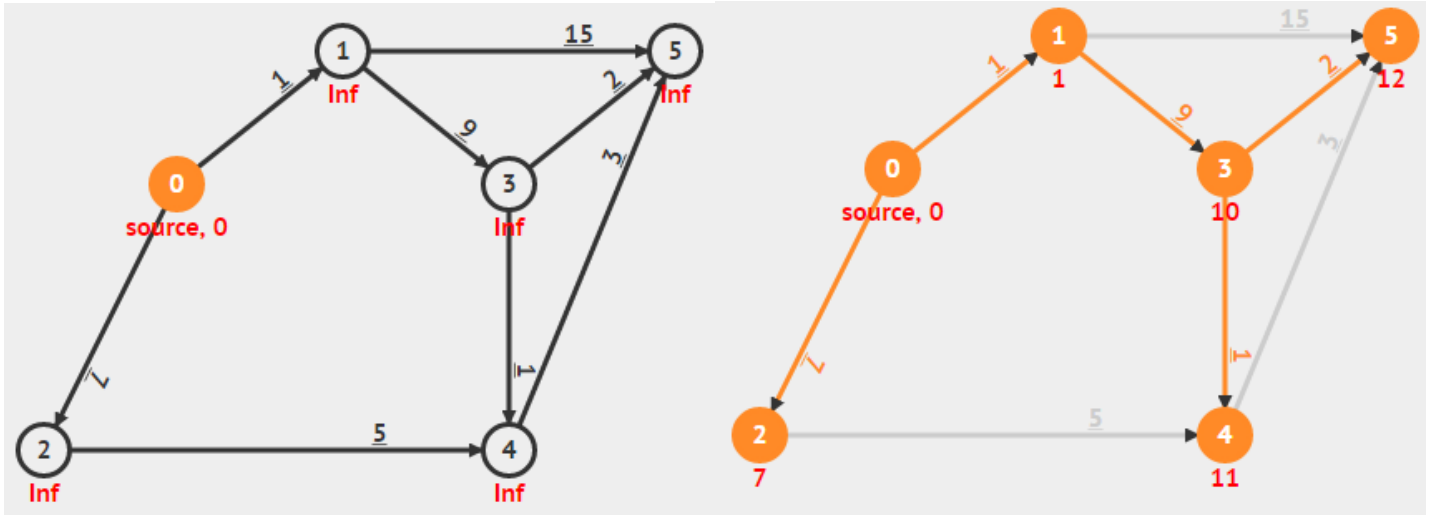
- SSSP on DAG *Revisited*
  - A gentle introduction to Dynamic Programming (DP) technique
    - Optimal sub structure
    - Overlapping sub problems
- SS Longest Paths (SSLP) on DAG
  - Reduced back to SSSP on DAG
  - SSLP on DAG → Longest Increasing Subsequence (LIS)
- Counting Paths on DAG

Reference: CP3 Section 3.5 (only parts of this big section) & Section 4.7.1

# SSSP in DAG

One topological order of this DAG is {0, 2, 1, 3, 4, 5}

- Try relaxing the outgoing edges of the vertices listed in the topological order above
  - With just one pass, each vertex v will have the correct D[v]
- Try graph below which is a slight modification of Example graph: DAG in https://visualgo.net/sssp
  - Run "Dynamic Programming" from source vertex 0



# Analysis of SSSP on DAG

1. Pre-processing step: Topological sort
   - This can be done with O(**V**+**E**) modified DFS as in Lecture 06 (try TopoSort animation @ https://visualgo.net/dfsbfs)
2. Then, following this topological order (**V** items), relax a total of **E** edges
   - The total number of outgoing edges from all vertices is **E**
   - So again, it is O(**V**+**E**)

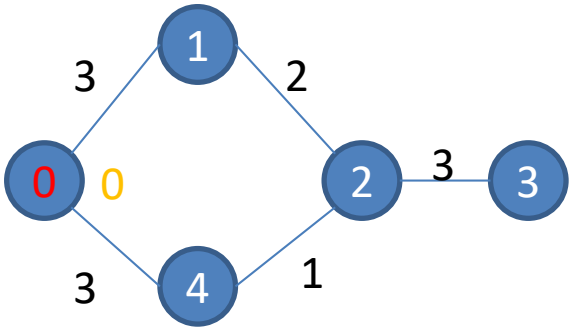
Thus, SSSP on DAG can be solved in **linear time**: O(**V**+**E**)

- Linear in terms of **V** and **E** → 1 pass of all **V** vertices and **E** edges
- Another name: "One-Pass Bellman Ford's"

# Why Does It Work? (1)

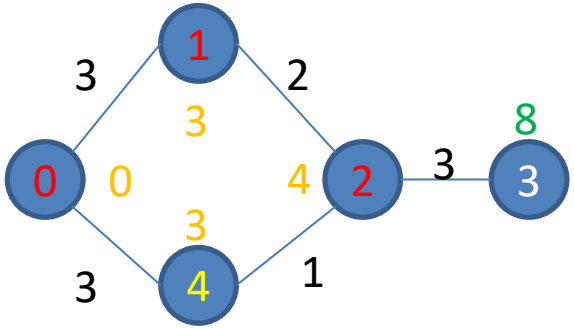
Cycle is a major issue in SSSP (from lecture 9)

- Can cause an edge to be relaxed multiple times (depending on order of edge relaxation) as multiple paths can use the same edge




Solve SSSP at source vertex 0

Order of edge relaxation
0-1, 0-4, 1-2, 2-3, 4-2




After one pass

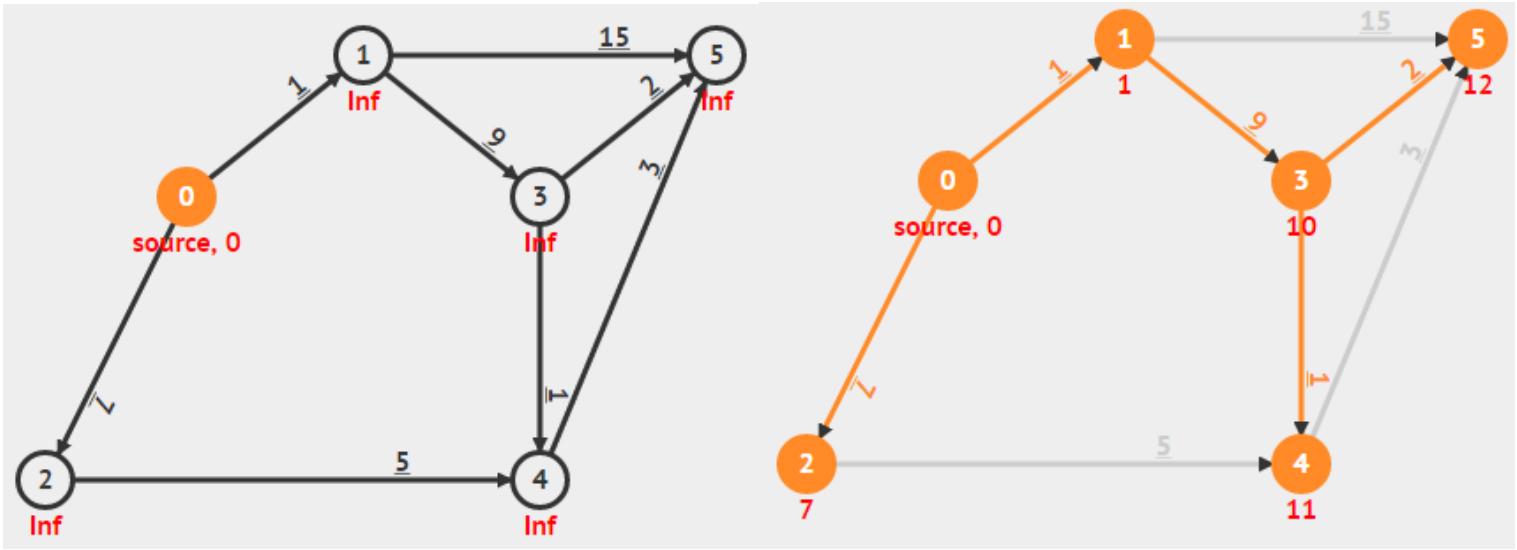
SP to vertex 3 not yet found because sequence of edge relaxation caused the the longer path (0,1,2,3) to be found first before the shorter path (0,4,2,3)

- Thus for general graph, we are only *sure* to have solved the SSSP after doing all-edges relaxation **V**-1 times (bellman ford)

# Why Does It Work? (2)

On DAG, there is **no cycle** → we have topological order

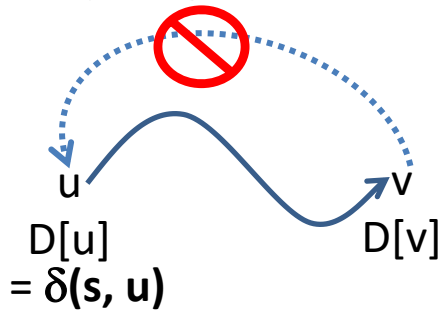
- Recall the meaning of topological order:
  - Linear ordering of vertices such that for every **edge(u, v)** in DAG, vertex **u** comes before **v** in the ordering




One Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}

# Why Does It Work? (3)

- If the vertices are processed according to topological order and **u** is the next vertex, then after **relaxation of all outgoing edges of u** is performed, there *will never be* any other better path in the future that reaches vertex **u** so that we need to relax all its edges again…
  - There is no way some vertex **v** to the right of **u** in the topological ordering can reach back to vertex **u** to improve D[u]
    - There is **no cycle** that allows
      **u** → ~ some other vertices ~→ **v** → **u**




  - That means **D[u]** = **δ(s, u)** and we can safely propagate this final shortest path value to all its neighbors
- Thus SSSP **on DAG** can be solved in O(**V**+**E**) time
  - We do not have to repeat this **V**-1 times ☺

# But where is the DP? (Part 1)

Dynamic Programming is a feature of both

1. The nature of the problem

and

2. The solution to the problem which exploits this nature

# Where is the Recursion/DP? (Part 2)

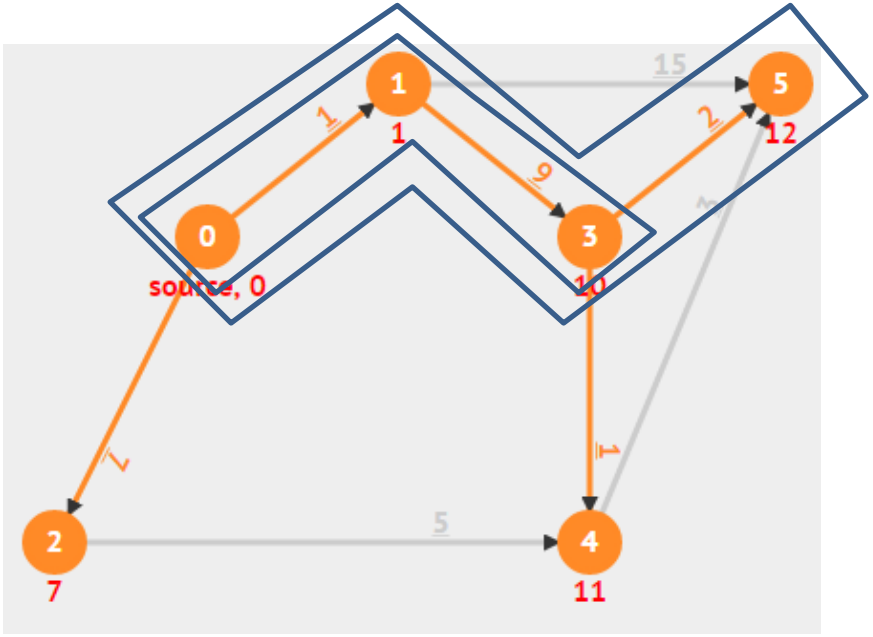
Observe, for example, shortest path 0→1→3→5

- Sub paths of this path (e.g. 0→1→3) are shortest paths too!
- We need to find sp from 0 to 3 before sp from 0 to 5

First ingredient of DP:

**Problem exhibits Optimal sub-structure!**

**Optimal** solution to problem can be constructed efficiently from **optimal** solutions of its sub-problems



# Where is the Recursion/DP? (Part 3)

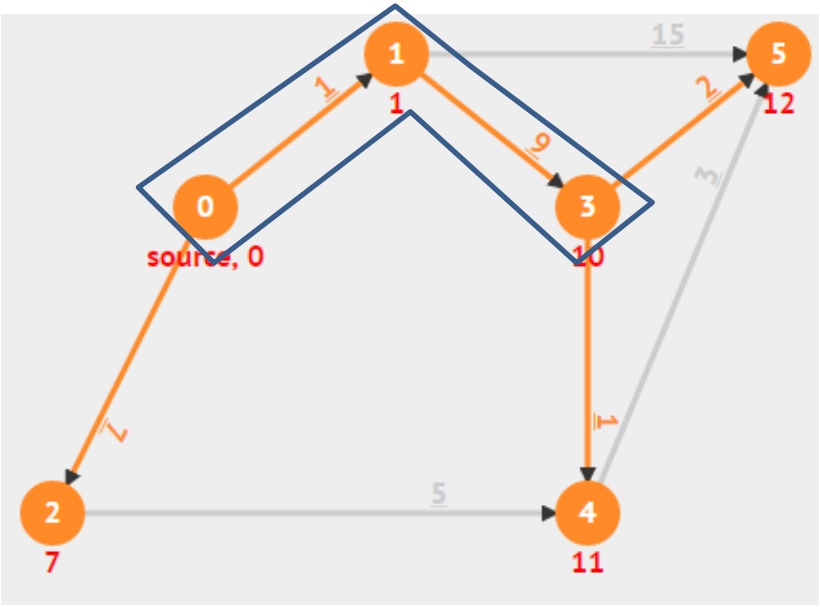
Observe, shortest path 0→1→3→4 & 0→1→3→5

- 0→1→3 is an overlapping subpath of these 2 shortest paths!

Second ingredient of DP:

**Problem exhibits Overlapping Sub-problems!**

problem can be broken down into sub-problems which are reused several times



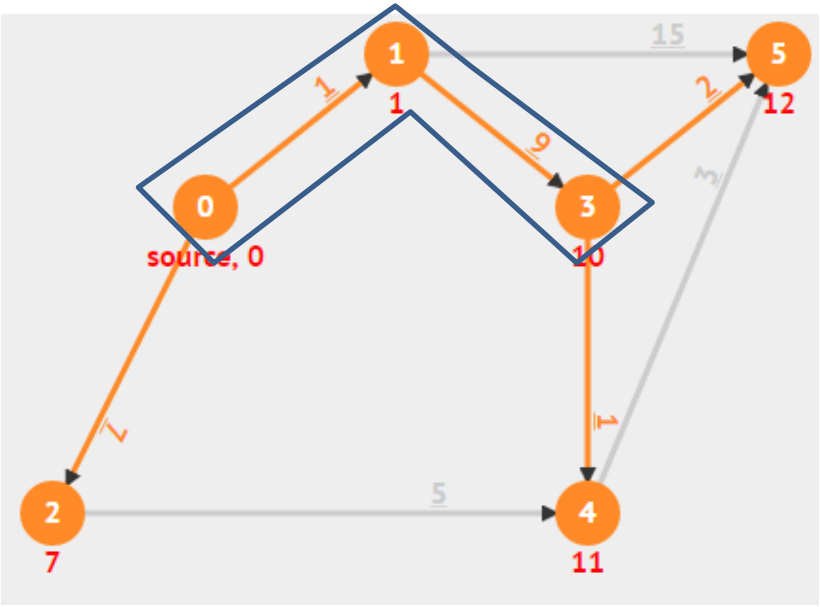
# Where is the Recursion/DP? (Part 4)

- We do ***not*** re-compute SP from 0 to 3 to get SP from 0 to 4 and SP from 0 to 5
  - Topological order is the correct order to avoid re-computations
    - This is called **"bottom-up" DP**: From known base case (distance to source is 0, compute the distance to other vertices using the topological order of DAG)

## Third ingredient of DP:

## No repeated computation of an overlapping sub-problem !

Solve sub-problem once, save the solution and re-use it !



# Some DP terminology

- In DP, a vertex in the (explicit or implicit) DAG is also known as a state
- The edge u->v is also known as a transition from state u to state v

Not harder than SSSP on DAG

# SS LONGEST PATHS ON DAG

# Longest Paths on DAG (1)

Program Evaluation and Review Technique (PERT)

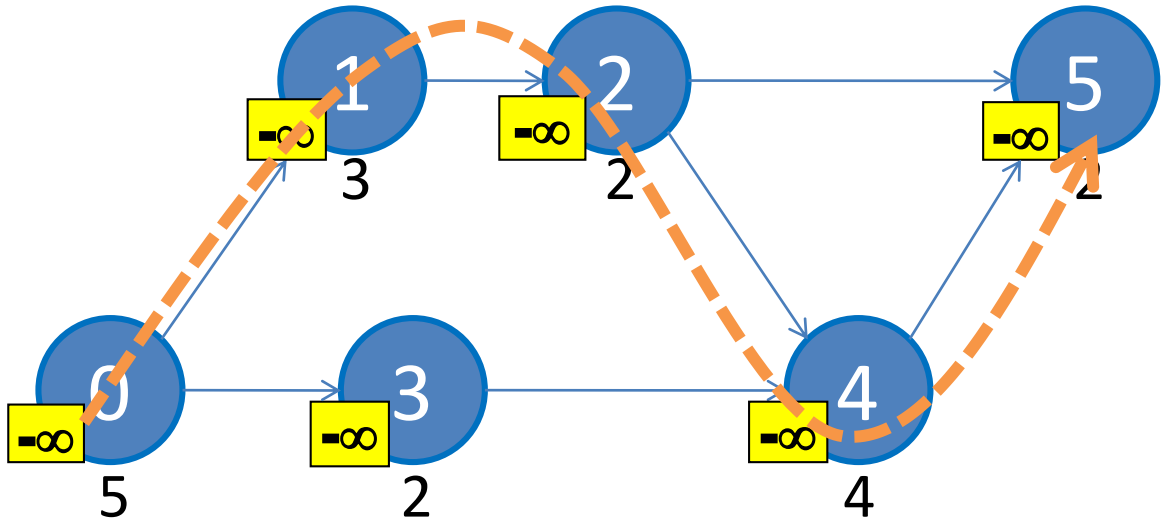
- PERT is a project management technique
- It involves breaking a large project into several tasks, estimating the time required to perform each task, and determining which tasks can not be started until others have been completed
  - This is similar to module pre-requisites!
  - This is a DAG!
- The project is then summarized in chart form
- See the next few slides for an example

# Longest Paths on DAG (2)

Problem source: [UVa 452 – Project Scheduling]

- Verify that this graph is a DAG!
  - The weight is **on vertices**, e.g. weight(0) = 5 (see next slide)
- The shortest way to complete this project is the…
  - **longest path** found in the DAG… (a bit counter intuitive)

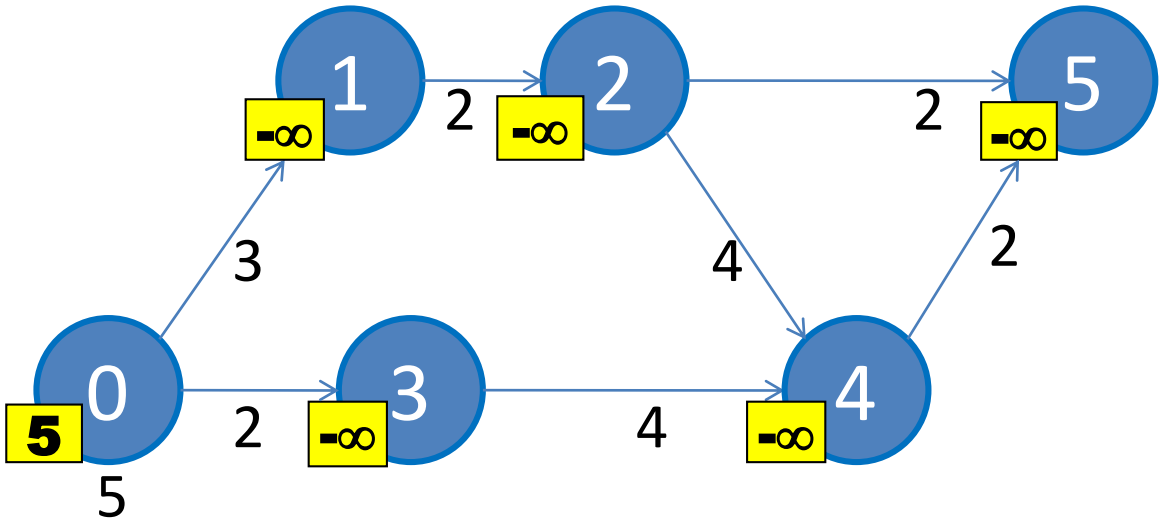
Notice the -∞



# Longest Paths on DAG (3)

Dealing with vertex weight issue

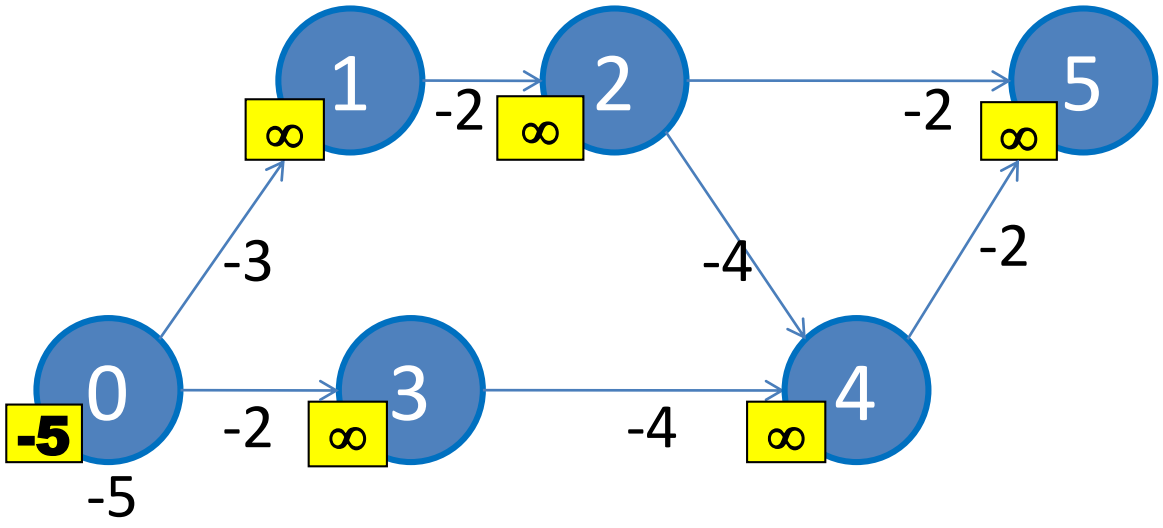
- Initially, we set D[0] = weight(0) = 5
- Then, we use the weight of destination vertex as the "edge weight" of incoming edges, i.e. we transform the graph like this



# Longest Paths on DAG (4)

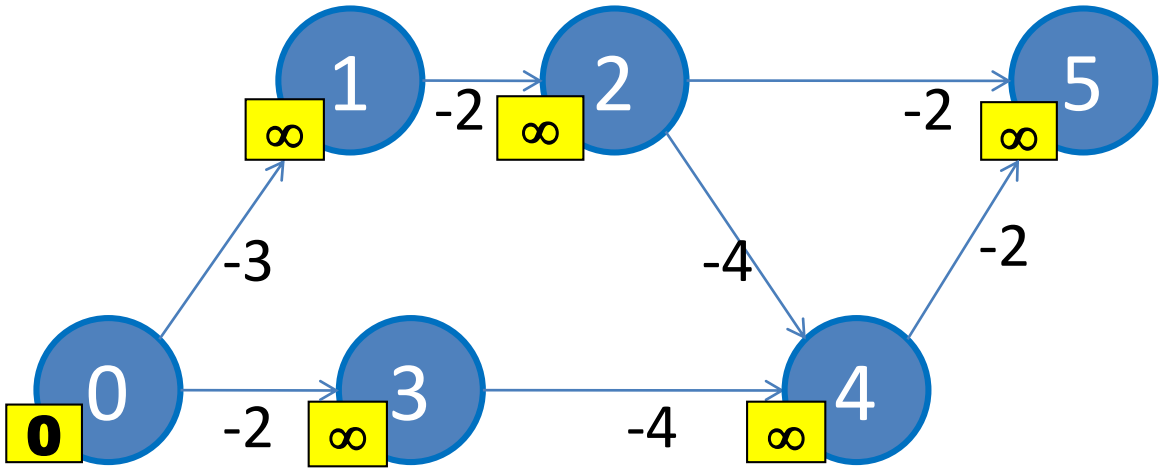
Reducing Longest Path back to Shortest Path **on DAG**

- We negate all weights (also D[0] = -weight(0) = -5)
- Then, we simply run the O(**V**+**E**) Shortest Path on DAG algorithm discussed earlier



# Longest Paths on DAG (5)

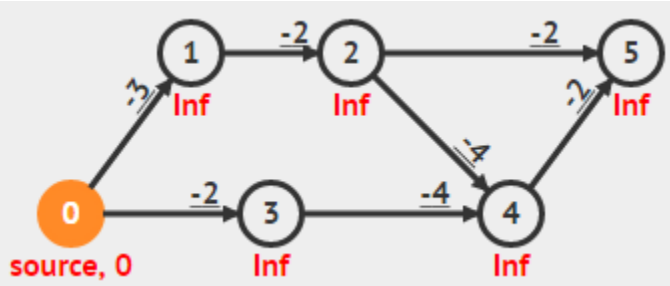
As VisuAlgo is not designed for this graph transformation, we assume that the source vertex weight is 0, we will add -5 to all vertices later

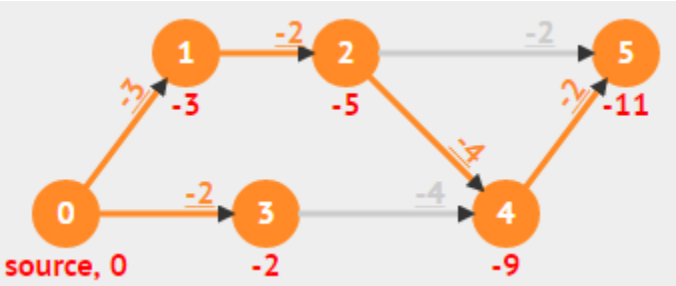


# Longest Paths on DAG (6)

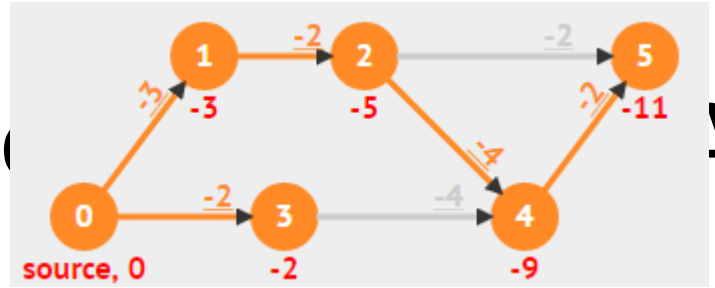
https://visualgo.net/en/sssp?create={"vl":{
"0":{"x":120,"y":240},"1":{"x":180,"y":160},
"2":{"x":260,"y":160},"3":{"x":220,"y":240},
"4":{"x":340,"y":240},"5":{"x":400,"y":160}},
"el":{
"0":{"u":0,"v":1,"w":"-3"},"1":{"u":1,"v":2,"w":"-2"},
"2":{"u":0,"v":3,"w":"-2"},"3":{"u":2,"v":4,"w":"-4"},
"4":{"u":3,"v":4,"w":"-4"},"5":{"u":2,"v":5,"w":"-2"},
"6":{"u":4,"v":5,"w":"-2"}}}

Then Run Dynamic Programming from source vertex 0



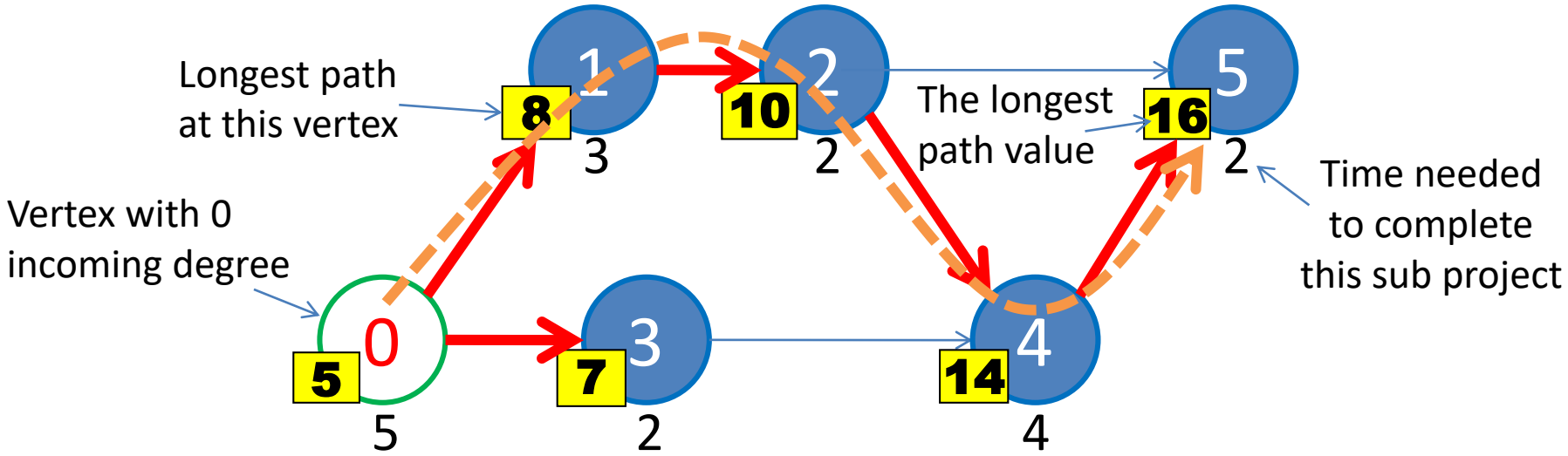



# Longe AG (7)




Finally, we add -5 to all vertices then negate all values

- The **thick red edges** are the LP Spanning Tree
  - Scan the whole D[v], find the largest one
    - In this example D[5] = 16 is the largest
    - Use predecessor information (the **thick red edges**) to reconstruct the longest path: 0→1→2→4→5



# Analysis of Longest Paths on DAG

(The same as SSSP on DAG)

1. Pre-processing step: Topological Sort, O(**V**+**E**)
2. Negate all edge weight in O(**E**)
3. Then, following this topological order (**V** items), relax a total of **E** edges, O(**V**+**E**)
4. Negate all distance values back, O(**V**)

In overall, longest paths on DAG can be solved in linear time: O(**V**+**E**)

- Linear in terms of **V** and **E**
- Later in CS3230, you will learn that the longest paths problem on **general graph** is NP-Complete (i.e. 'very hard'), Q: Why?

# Longest Paths ← → LIS :O

There is one more classical CS problem that can be modelled as longest paths in (implicit) DAG

- The Longest Increasing Subsequence (LIS)

While we are at this topic, let's discuss it as well ☺

- In the next few slides, we will see LIS, the implicit DAG in LIS, and the solution

A sister problem that is **very related** to SSLP on DAG

*(actually in https://visualgo.net/recursion, alternative view)*

# LONGEST INCREASING SUBSEQUENCE (LIS)

# Longest Increasing Subsequence (1)

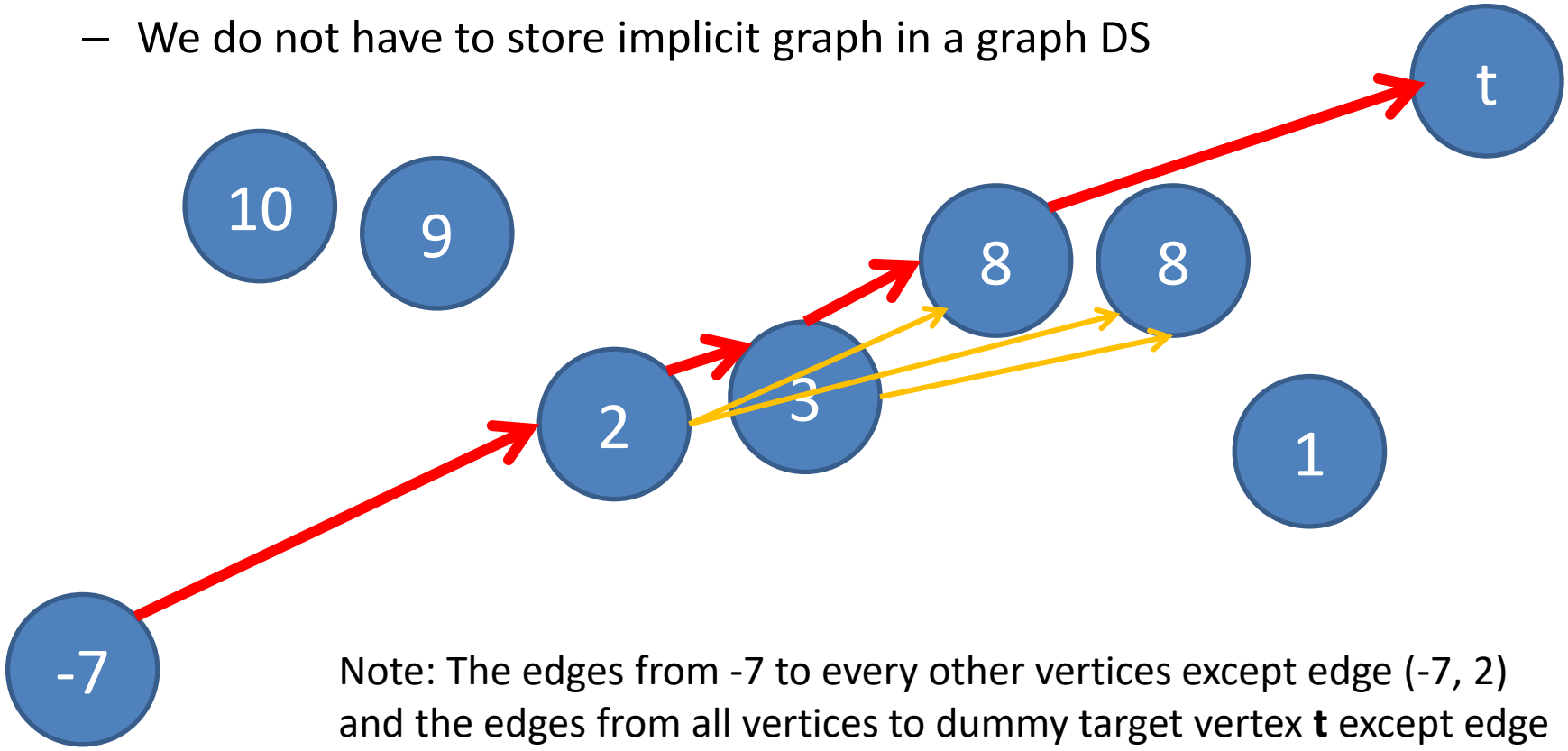
Problem Description (Abbreviated as LIS):

- As implied by its name….
  Given a sequence {A[0], A[1], …, A[**N**-1]} of length **N**, determine the Longest Increasing Subsequence
  - Subsequence is not necessarily contiguous
- Example: **N** = 8, sequence A = {-7, 10, 9, 2, 3, 8, 8, 1}
  - LIS is {-7, 2, 3, 8} of length 4
- Variants:
  - Longest Decreasing Subsequence
  - Longest Non Decreasing Subsequence

# Longest Increasing Subsequence (2)

There is **an implicit DAG** in this sequence A

- See the implicit DAG of sequence A = {-7, 10, 9, 2, 3, 8, 8, 1, ∞}
  - We do not have to store implicit graph in a graph DS




Note: The edges from -7 to every other vertices except edge (-7, 2) and the edges from all vertices to dummy target vertex **t** except edge (8, t) are not shown to avoid cluttering this picture

# Longest Increasing Subsequence (3)

D array is now of size N+1 (include the ∞)

Let D[i] = length of the best LIS ending at vertex representing A[i]

- Initialize D[0] to D[N] = 0 **Q:Why?**

The topological order is obviously an ordering of the indices and it is {0, 1, 2, …, **N**}, **Q: Why?**

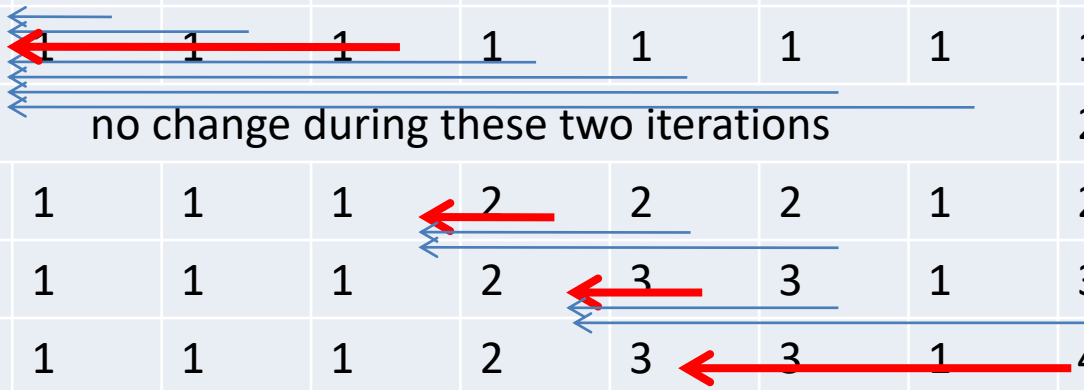
- "Stretch" all i (vertex) one by one using this order

```
D[0] to D[N] = 0 // base case
for i = 0 to N-1 // this is O(N^2) Bottom-Up DP
  for j = i+1 to N
    if A[i] < A[j] // an implicit edge!
      stretch(i, j, 1) // edge weight is 1
report D[N] // the answer is here

//The sequence itself can be gotten from p by
//following the predecessors from p[N] until hit -1
```

# Longest Increasing Subsequence (4)

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| A | -7 | 10 | 9 | 2 | 3 | 8 | 8 | 1 | ∞ |
| D (initial) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D (i = 0) | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| D (i = 1-2) | no change during these two iterations | | | | | | | | 2 |
| D (i = 3) | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 2 |
| D (i = 4) | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 1 | 3 |
| D (i = 5) | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 1 | 4 |
| D (i = 6-8) | no more change | | | | | | | | |



This LIS problem can be solved in O($\mathbf{n^2}$), analysis:

1. Use the fact that there are two nested loops of size n, or
2. Use the analysis of the longest paths on (implicit) DAG where there are **V** = **n** vertices and **E** = O($\mathbf{n^2}$) edges. **Q: why?**

# Top-Down DP solution for LIS (Part 1) – The Recurrence

This LIS problem is more naturally solved in "Top-Down Dynamic Programming (DP)" fashion

- Let **LIS(i)** be the length of the LIS starting from index **i** and going to **N-1** (the end of the sequence)
  - This can be written as a function with one parameter, index **i**
- We can write the solution using this recurrence relations:
  - LIS(**N**-1) = 1 // at last position, we cannot extend the LIS anymore
  - LIS(**i**) = max(LIS(**j**)+1), $\forall$**j** $\in$ [**i**+1 .. **N**-1] where A[**i**] < A[**j**]

# Top-Down DP solution for LIS (Part 2) – Recursive Solution

This can be written using (Java) recursive function

- Notice that this version is very slow due to recomputations

```
private static int LIS(int i) {
  if (i == N-1) return 1;

  int ans = 1; // at least A[i] itself
  for (int j = i+1; j < N; j++)
    if (A[i] < A[j])
      ans = Math.max(ans, LIS(j)+1);
  return ans;
}
// to compute LIS for a given sequence
// return Max(LIS(0), LIS(1),…, LIS(N-1))
```

# Top-Down DP solution for LIS (Part 3) – Turn Recursion into Memoization!

*Key Point: Space-Time Tradeoff*

initialize memo table in the main method

return_value recursive_function(state) {
  if state already calculated, simply return its value
  calculate the value of this state using recursion
  save the value of this state in the memo table
  return the value
}

# Top-Down DP solution for LIS (Part 4) – Turn Recursion into Memoization!

A much better version (see LISDPDemo.java):

```
private static int LIS(int i) {
  if (i == N-1) return 1;
  if (memo.get(i) != -1) return memo.get(i);

  int ans = 1; // at least A[i] itself
  for (int j = i+1; j < N; j++)
    if (A[i] < A[j])
      ans = Math.max(ans, LIS(j)+1);
  memo.set(i, ans);
  return ans;
}
// values in memo are set to -1 in main method
```

# Top-Down DP solution for LIS (Part 4) – Analysis

- For each call to LIS(i), either solution to i has been computed or need to compute it.
- If need to compute it, there is a for loop which run O(**n**) iteration
- Need to compute the solution to all n indices, total # of iterations of for loop is O($\mathbf{n^2}$) and each iteration is O(1) time.
- Subsequent calls to an LIS(i) after solution to i has been computed is O(1) time. At most O(n) such calls.
- Thus total time to compute solution for LIS is O($\mathbf{n^2}$) (same as bottom-up DP)
- However to get this improvement in speed over simple recursion we need extra O(n) space to save the solution for the DP

Final discussion for today, again about DAG ☺

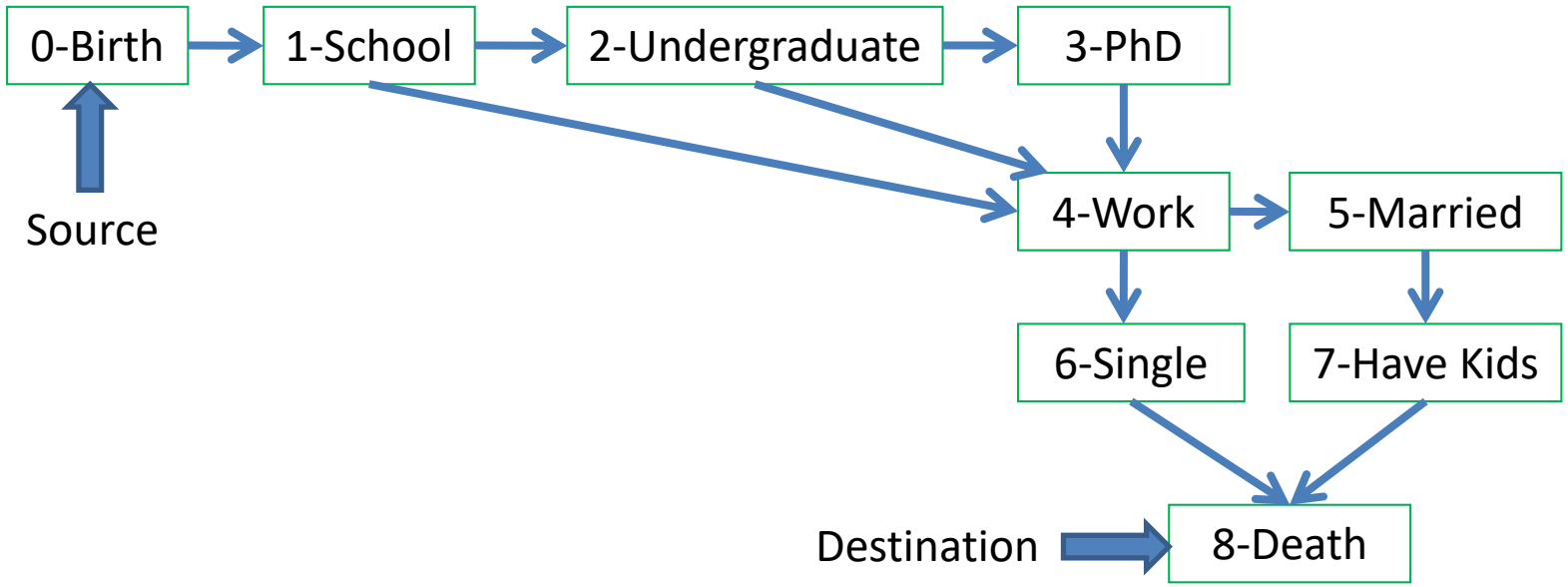
(not in VisuAlgo yet)

# COUNTING PATHS ON DAG

# Counting Paths on DAG

Given some real-life time line (obviously a DAG, Q: Why?)

- How many different possible lives can you live (from birth/vertex 0 to death/vertex 8)?




Answer = 6

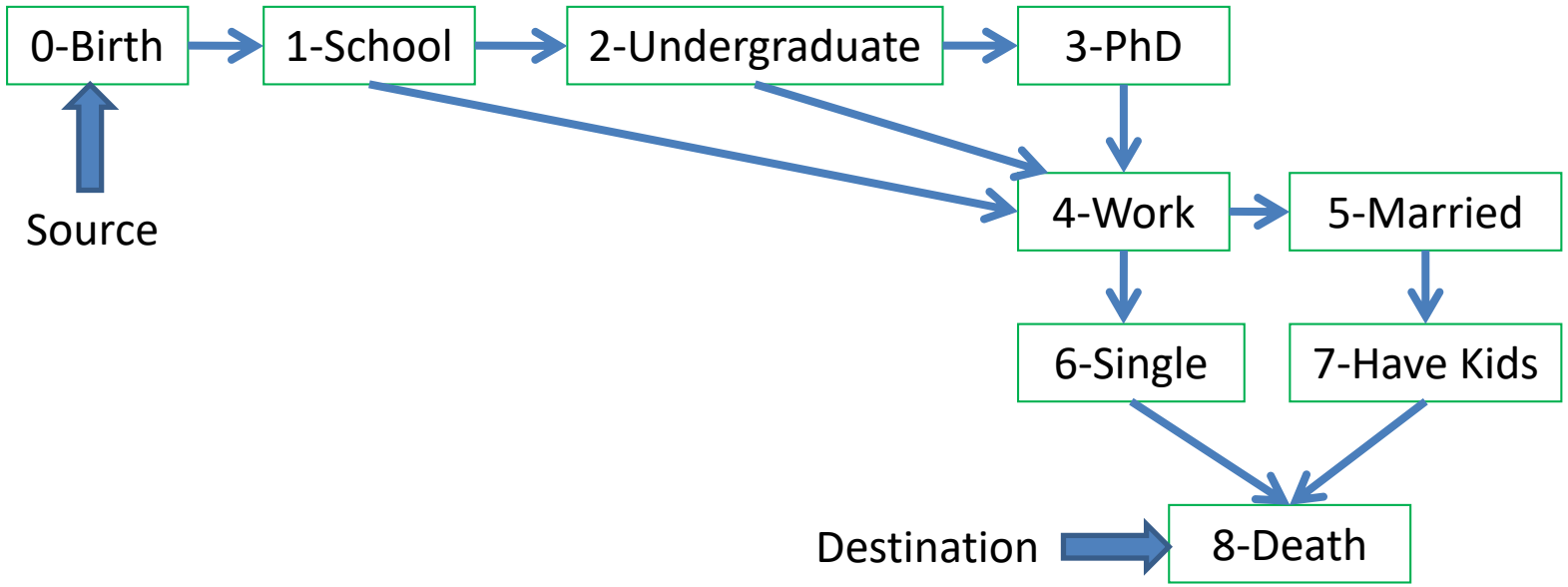
# Toposort/Graph/Bottom-Up Way (1)

Let **numPaths** be an integer array of size N (# vertices in graph)

**numPaths[i]** = number of paths from source vertex to vertex **i**

At the start **numPaths[0]** = 1, the rest is 0

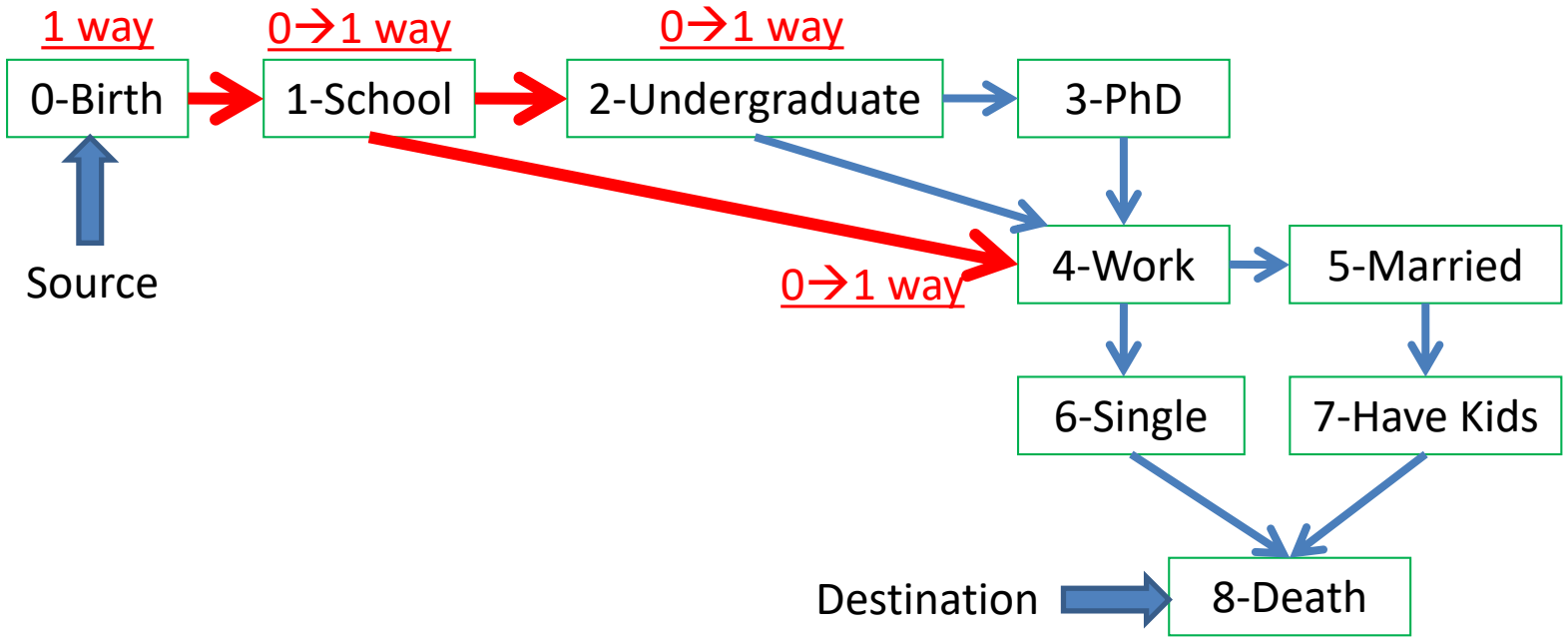
Find toposort: {0, 1, 2, 3, 4, 6, 5, 7, 8}



# Toposort/Graph/Bottom-Up Way (2)

Toposort = {0, 1, 2, 3, 4, 6, 5, 7, 8}

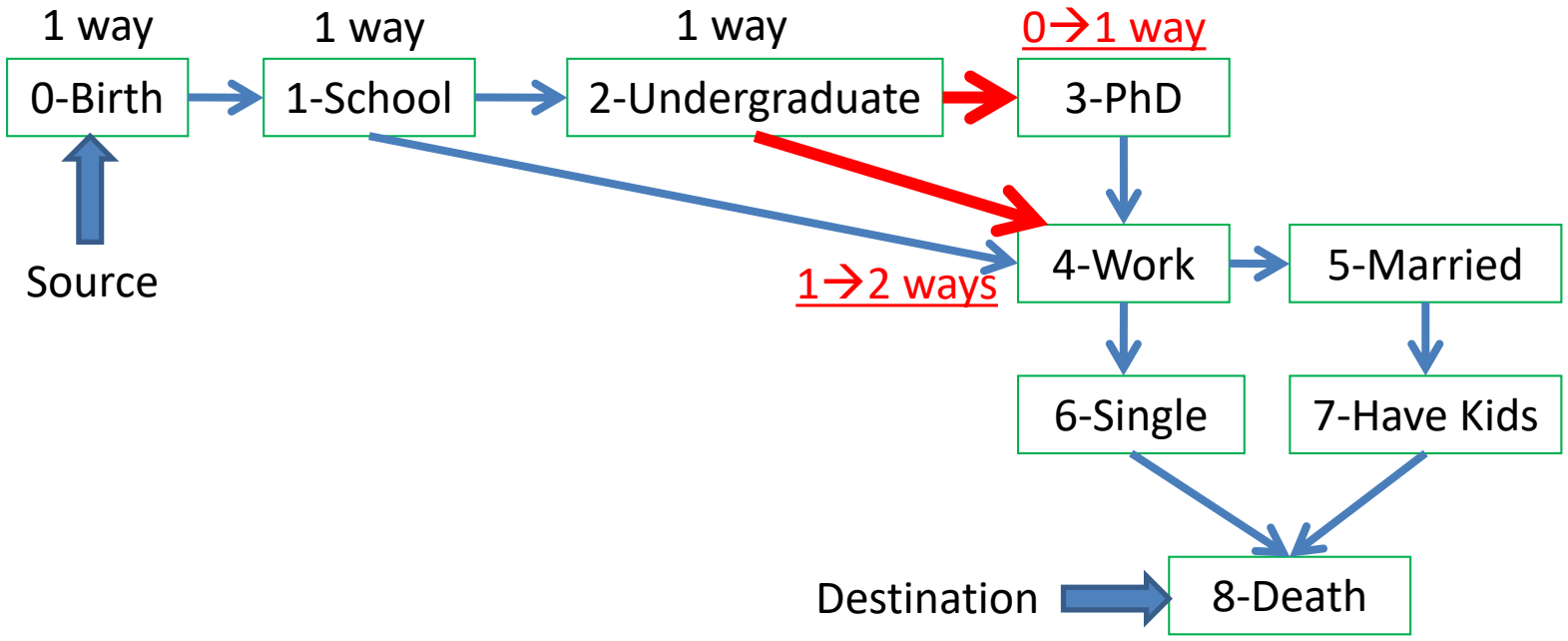
- numPaths[0] = 1, propagate to vertex 1, and then 2, 4



# Toposort/Graph/Bottom-Up Way (3)

Toposort = {0, 1, 2, 3, 4, 6, 5, 7, 8}

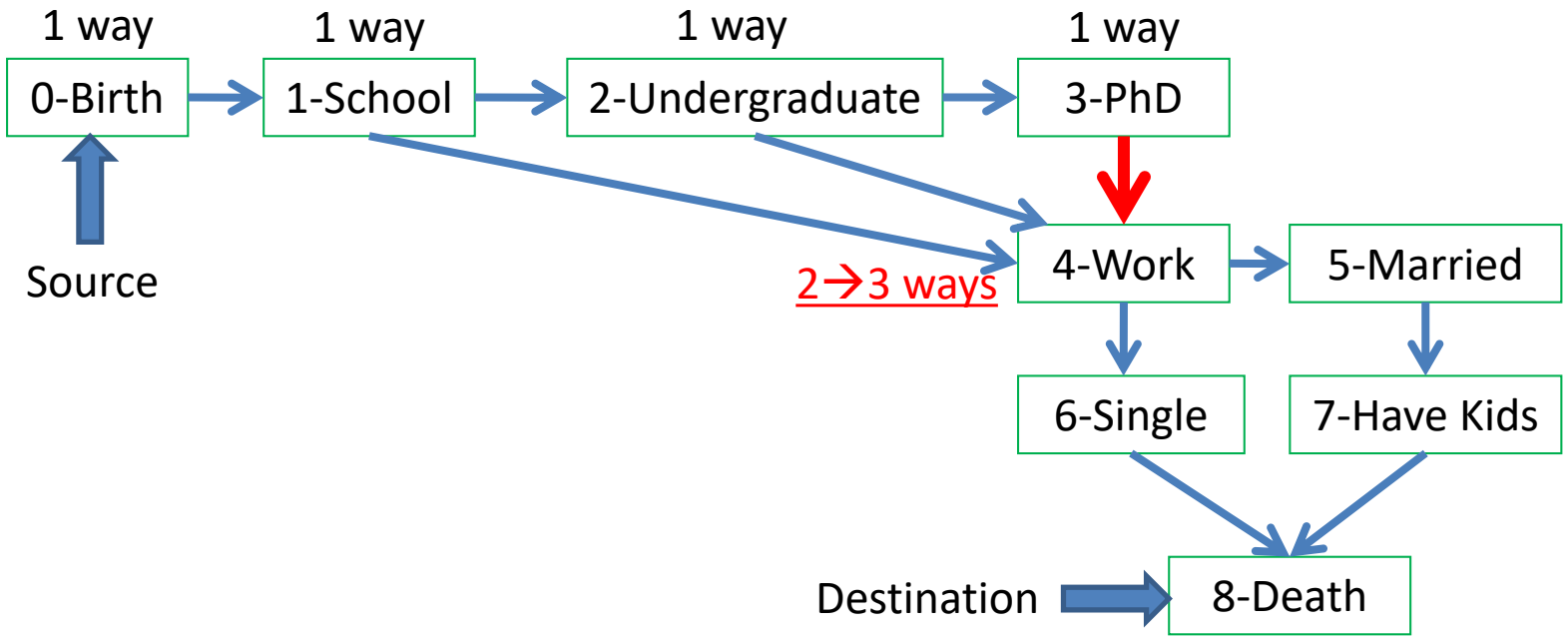
- numPaths[2] = 1, propagate to vertex 3 and 4



# Toposort/Graph/Bottom-Up Way (4)

Toposort = {0, 1, 2, 3, 4, 6, 5, 7, 8}

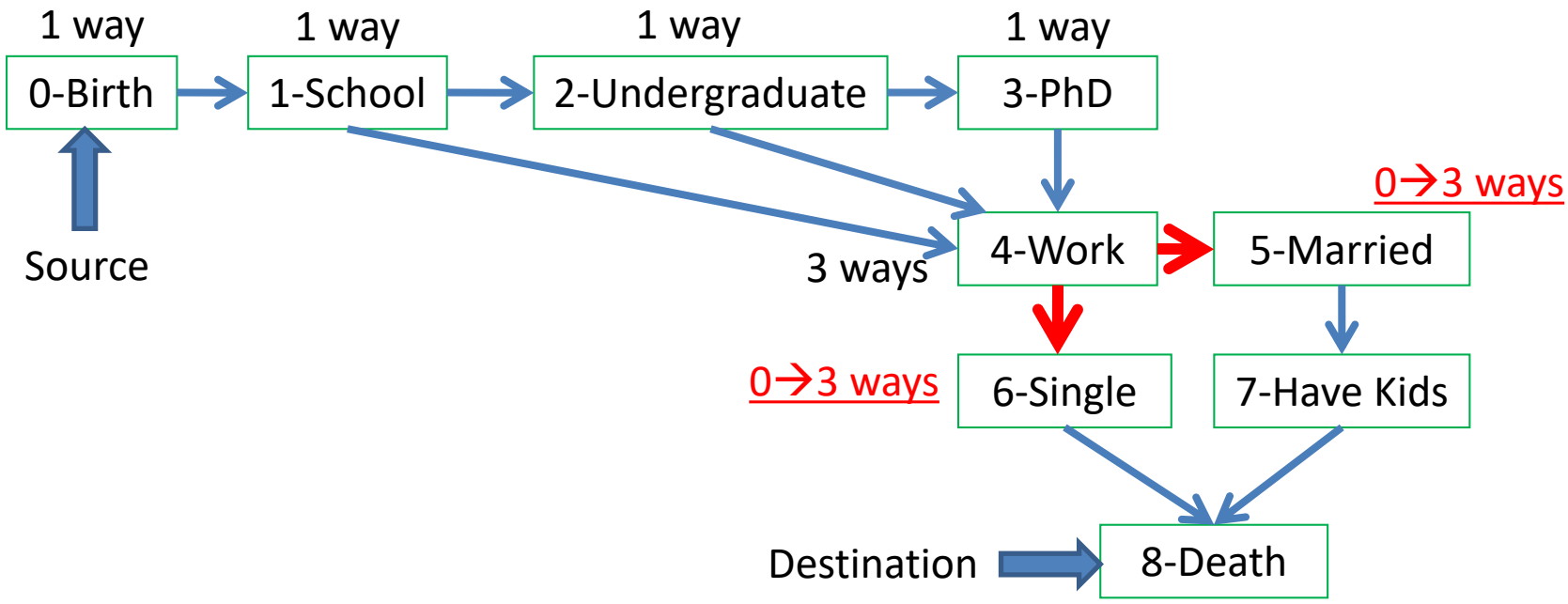
- numPaths[3] = 1, propagate to vertex 4



# Toposort/Graph/Bottom-Up Way (5)

Toposort = {0, 1, 2, 3, 4, 6, 5, 7, 8}

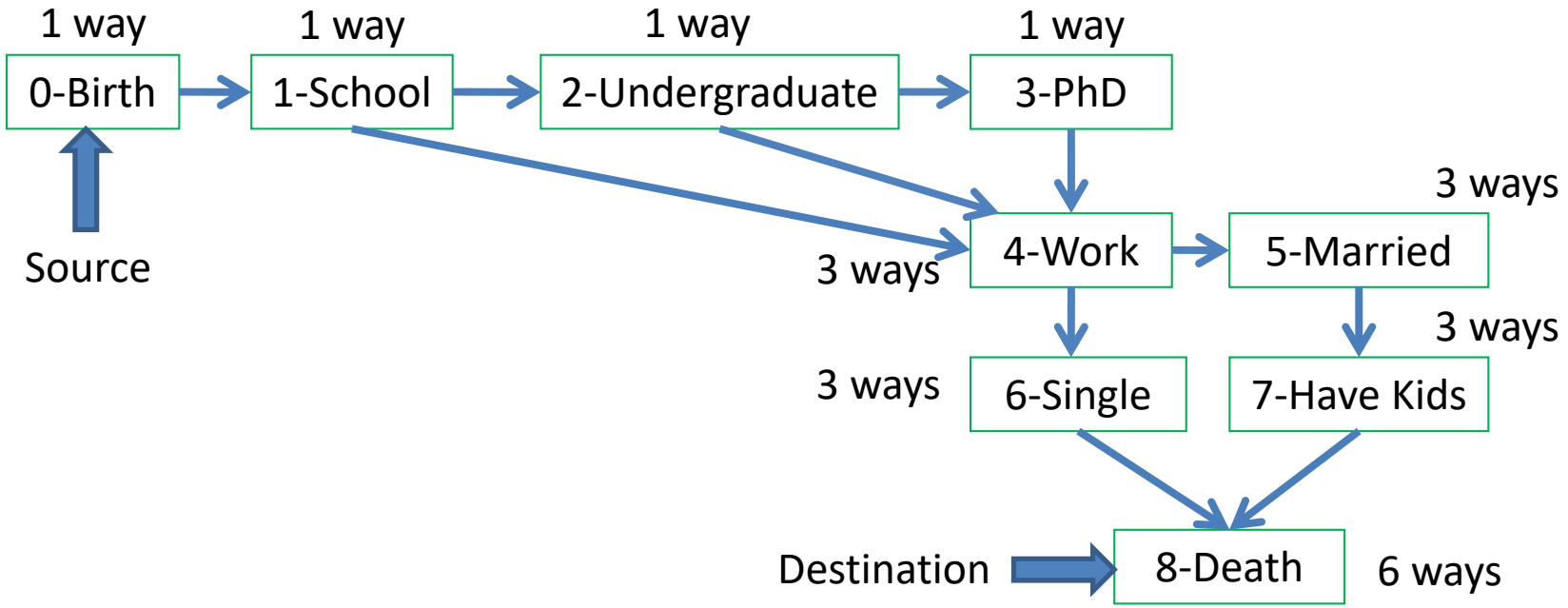
- numPaths[4] = 3, propagate to vertex 5 and 6



# Toposort/Graph/Bottom-Up Way (6)

Toposort = {0, 1, 2, 3, 4, 6, 5, 7, 8}

- >> >> Fast forward…, this is the final state



# Top-Down DP Solution (Part 1)

We can solve "counting paths in DAG" with Top-Down DP

- That is: Using functions, parameters, and "memo table"
- *Let **numPaths(i)** be the number of paths starting from vertex **i** to destination **t** (vertex 8 for the example)
- We can write the solution using this recurrence relations:
  - numPaths(**t**) = 1 // at destination t, obviously only one path
  - numPaths(**i**) = ∑ numPaths(**j**), for all **j** adjacent to **i**
- To avoid recomputations,
  memoize the number of paths for each vertex **i**
- Only brief code is shown in the next slide
  - The overall code is similar to the LISDPDemo.java shown earlier

*Note that definition for numPaths(i) is changed to make it amenable for top-down DP

# Top-Down DP Solution (Part 2)

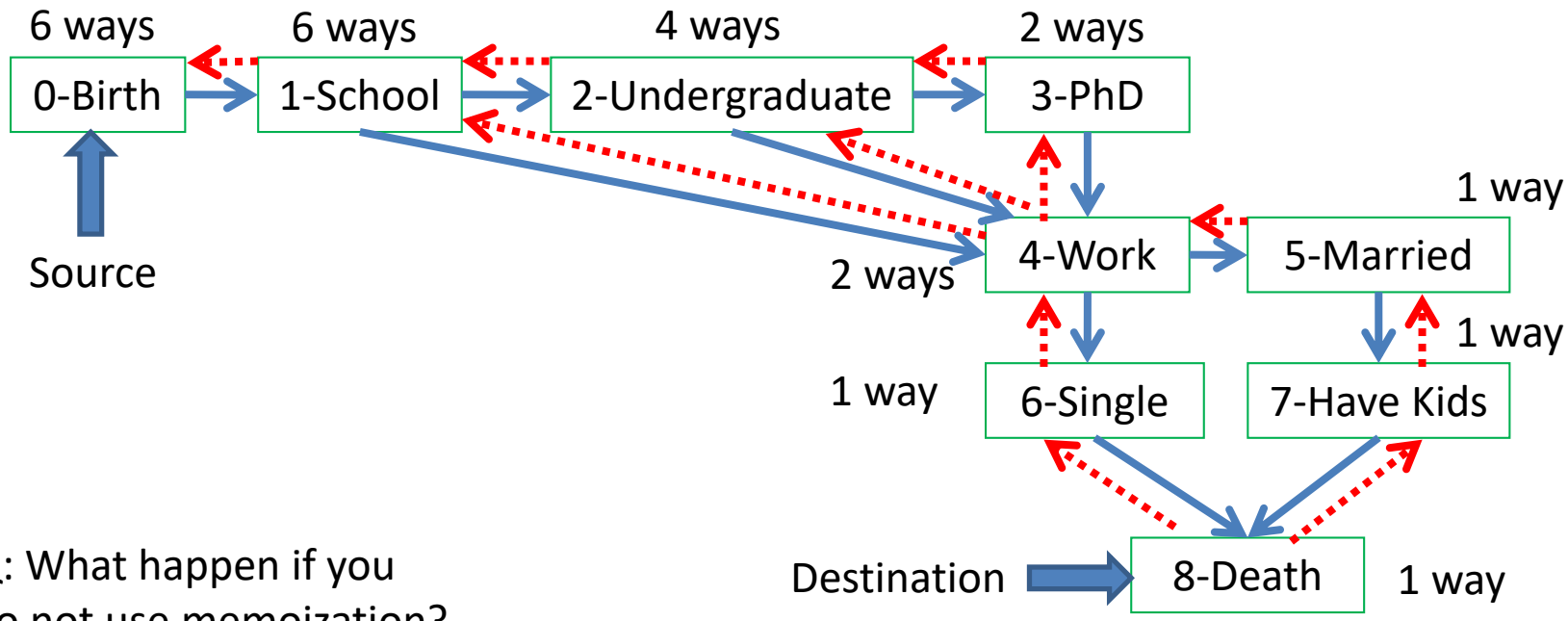
The (Java) recursive function with memoization

```
private static int numPaths(int i) {
  if (i == V-1) return 1;
  if (memo.get(i) != -1) return memo.get(i);

  int ans = 0;
  for (int j = 0; j < AdjList.get(i).size(); j++)
    ans += numPaths(AdjList.get(i).get(j).first());
  memo.set(i, ans);
  return ans;
}
```

# Top-Down DP Solution (Part 3)

The way the answer is computed is now from destination to source




Q: What happen if you do not use memoization?

# Transition to DP Topics

In this lecture, we link graph topic (DAG) to DP

- SSSP on DAG (revisited)
- SSLP on DAG ←→ LIS
- Counting Paths on DAG
- We show both "graph way" (algorithms on DAG/also known as Bottom-Up DP) and recursive way (also known as Top-Down DP)
- DP Ingredients:
  - Optimal sub-structure and Overlapping sub-problem
- Mapping the Terminologies:
  - Vertices → States; Edges → Transitions
- Space/Time complexity
  - |V| → Space Complexity; |V+E| → Time Complexity

# Plan

In the next 2 lectures (11-12), 3 tutorials (09-10-11), and 1 PS (6), we will use more implicit DAGs (on more structured problems) and use more DP terminologies rather than graph terminologies ☺