CS2010 – Data Structures and Algorithms II

Lecture 09 – Algorithms on DAG

stevenhalim@gmail.com

National University of Singapore

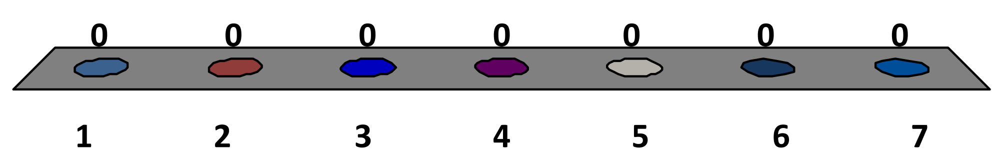
School of Computing

Outline

- What are we going to learn in this lecture?
  - (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
    - SSLP on DAG → Longest Increasing Subsequence (LIS)
    - Counting Paths on DAG
  - Reference: CP2.5 Section 3.5 & 4.7.1

Given a general graph, edge weights are positive integers, we want to solve SSSP, we should use:

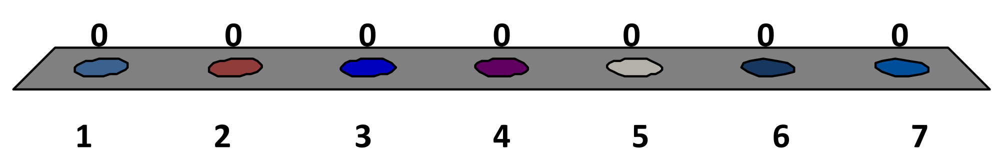
1. O(V + E) DFS
2. O(V + E) BFS
3. O(E log V) Kruskal’s
4. O(E log V) Prim’s
5. O(VE) Bellman Ford’s
6. O((V + E) log V) Dijkstra’s (Original Implementation)
7. O((V + E) log V) Dijkstra’s (Modified Implementation)



0 of 120

Given a connected weighted graph with one unique path between any two vertices, we want to solve SSSP, we should use:

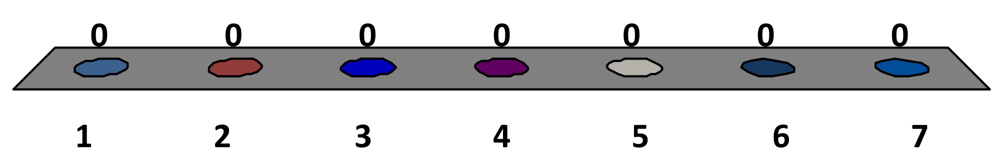
1. O(V + E) DFS
2. O(V + E) BFS
3. O(E log V) Kruskal's
4. O(E log V) Prim's
5. O(VE) Bellman Ford's
6. O((V + E) log V) Dijkstra's (Original Implementation)
7. O((V + E) log V) Dijkstra's (Modified Implementation)



0 of 120

Given a general graph, all edge weights are 7, we want to solve SSSP, we should use:

1. O(V + E) DFS
2. O(V + E) BFS
3. O(E log V) Kruskal's
4. O(E log V) Prim's
5. O(VE) Bellman Ford's
6. O((V + E) log V) Dijkstra's (Original Implementation)
7. O((V + E) log V) Dijkstra's (Modified Implementation)



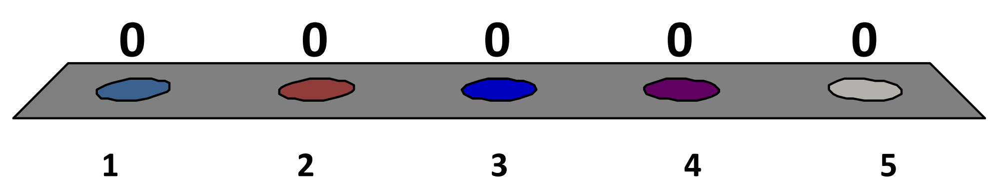
0 of 120

Which statements involving the original and the modified Dijkstra's implementations are true?

(click all that are applicable)

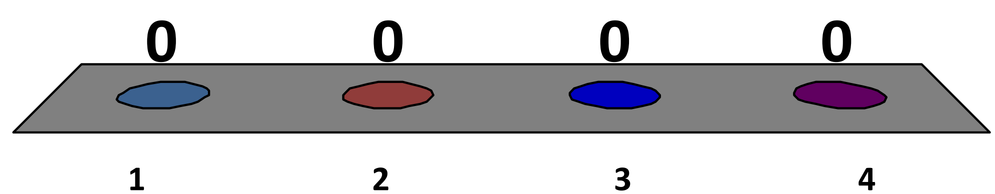
1. Both works well on graph with all positive weighted edges
2. Both works well on graph with positive/negative weighted edges but no negative weight cycle
3. Both terminates when run on graph with negative weight cycle
4. Both uses PriorityQueue
5. Both are equally easy to implement/code

0 of 120



Now we come to the last topic of CS2010: Dynamic Programming ☺

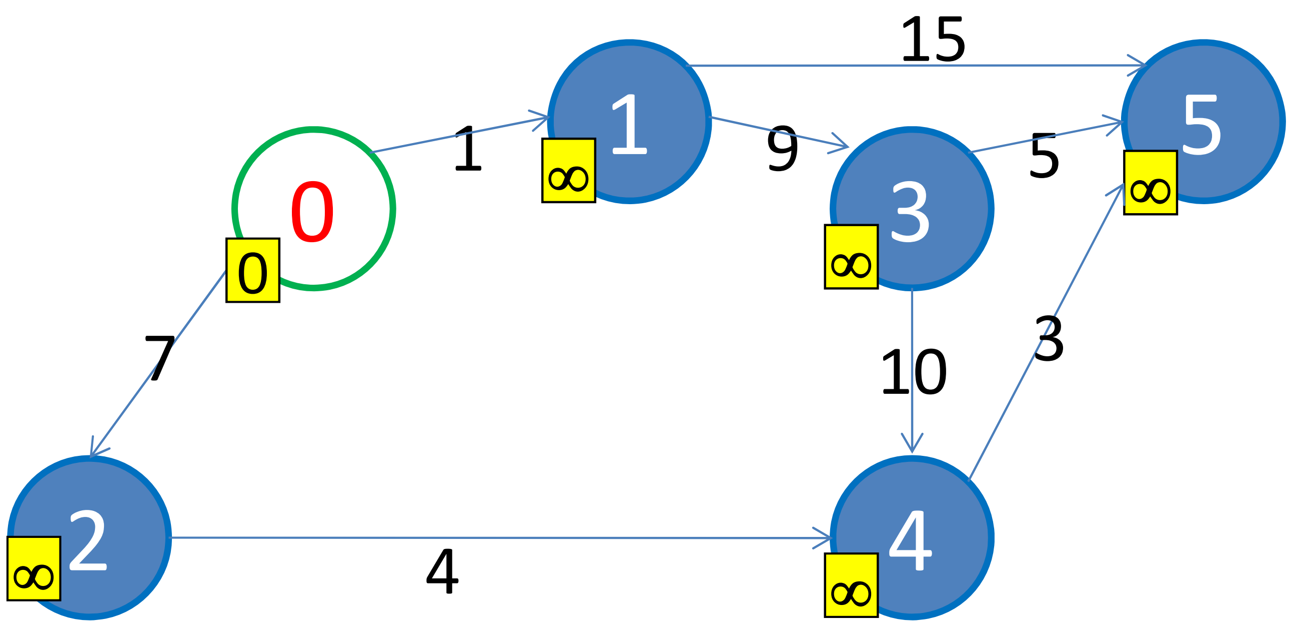
1. I have no problem with recursion and I happily passed the recursion-heavy module CS1101S ☺
2. I am not from CS1101S, but I am ready with lots of recursion, bring it on ☺
3. Hey, I have skimmed through this lecture note, where are the recursions?
4. I am afraid I will have problems with recursion :O



0 of 120

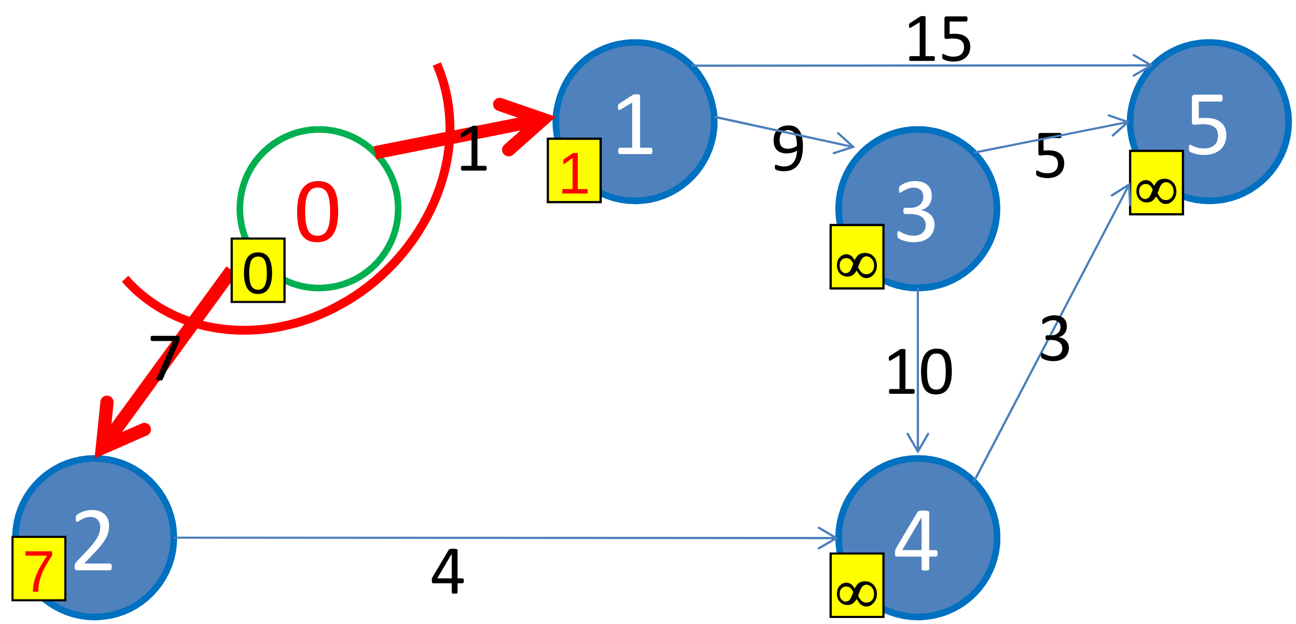
Review: SSSP in DAG (1)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Try relaxing the outgoing edges of the vertices listed in the toposort above
    - With just one pass, all vertex will have the correct D[v]



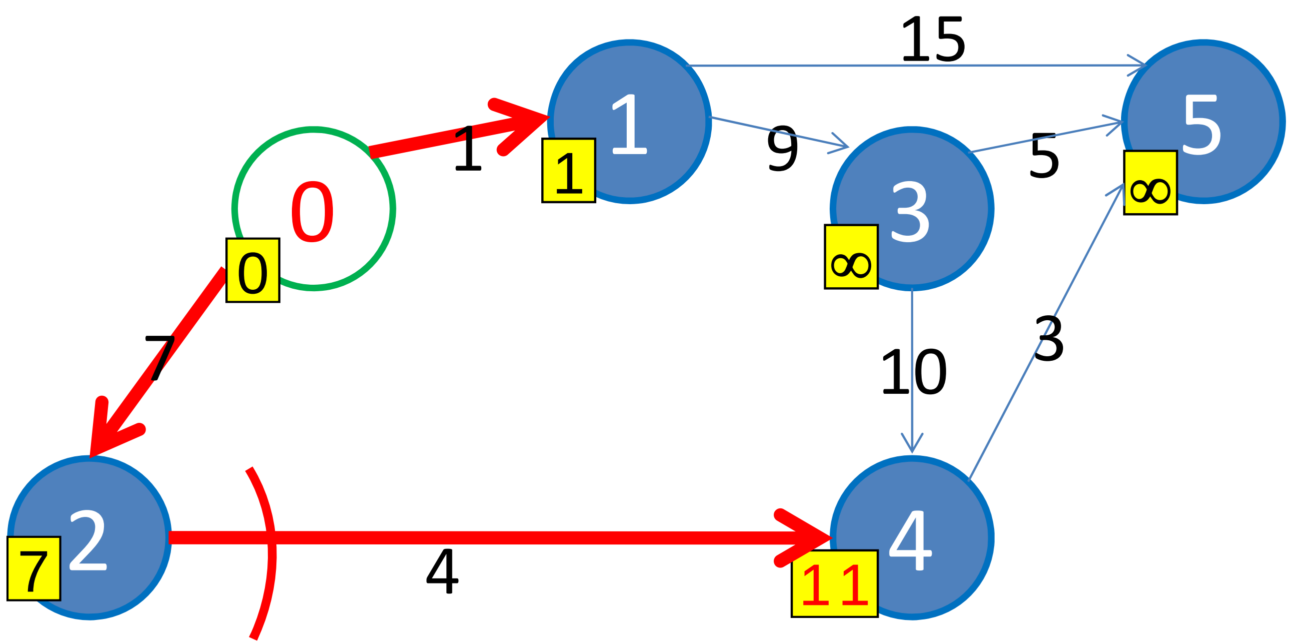
Review: SSSP in DAG (1)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Start from source (vertex 0)



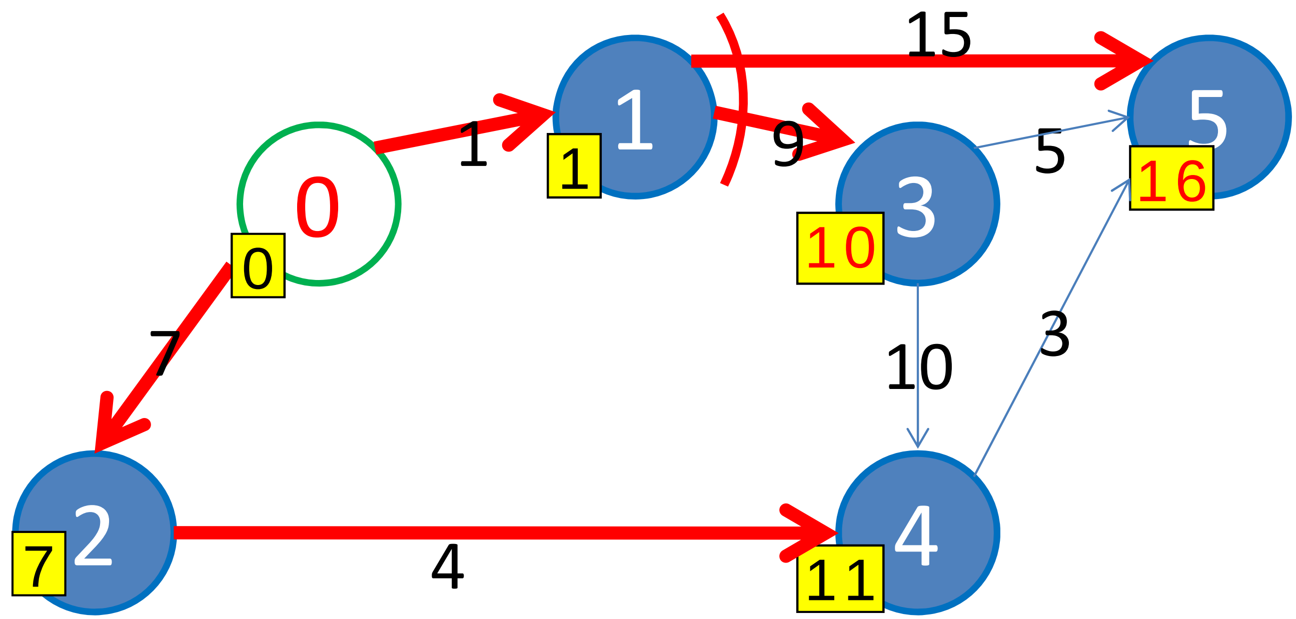
Review: SSSP in DAG (2)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Continue with vertex 2



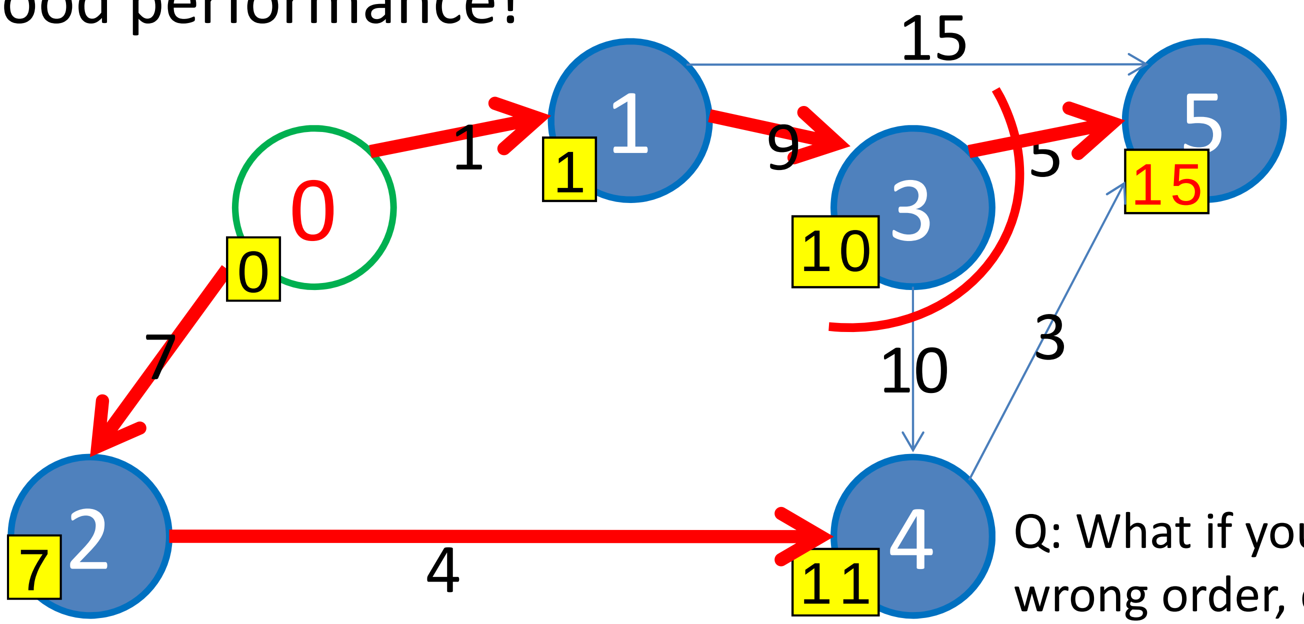
Review: SSSP in DAG (3)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Then vertex 1



Review: SSSP in DAG (4)

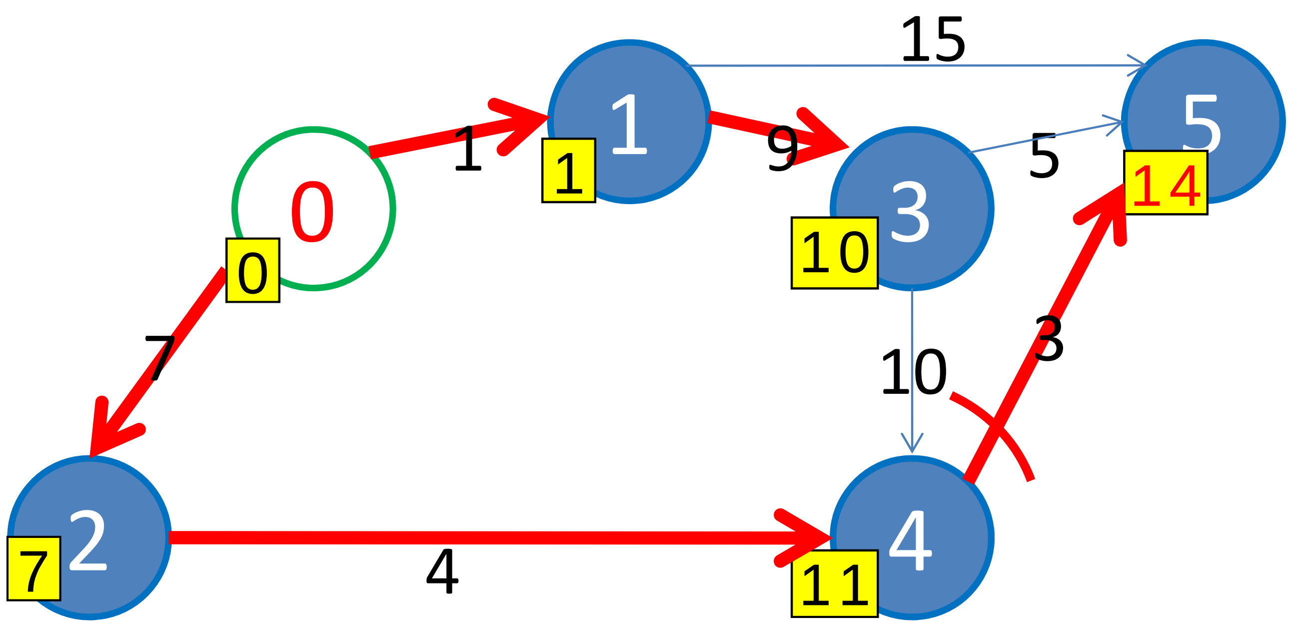
- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Then vertex 3; Vertices that **have been processed so far**, i.e. {0, 2, 1, 3} already have the correct final shortest path values, we **do not have to re-trace** our steps → key for good performance!



Q: What if you relax the edges in wrong order, e.g. at the start, we relax edge 1→3 first then 0→1, do you have to repeat 1→3 later?

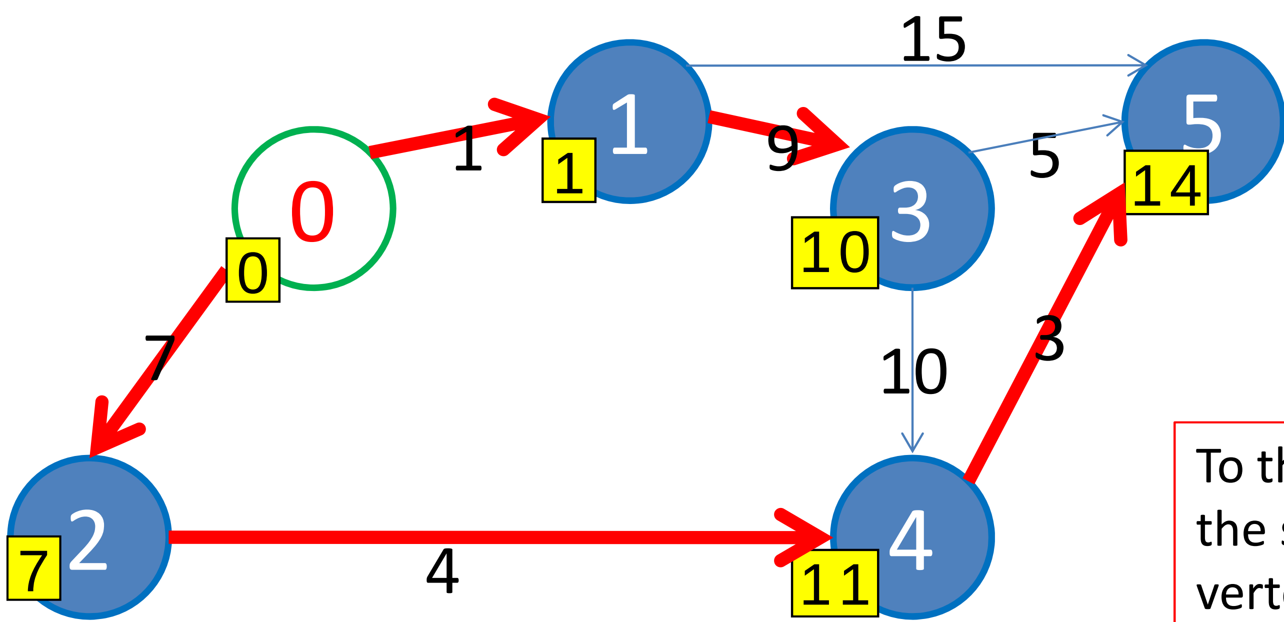
Review: SSSP in DAG (5)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Almost? done



Review: SSSP in DAG (5)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Final state
  - The **thick red edges** form the shortest paths spanning tree



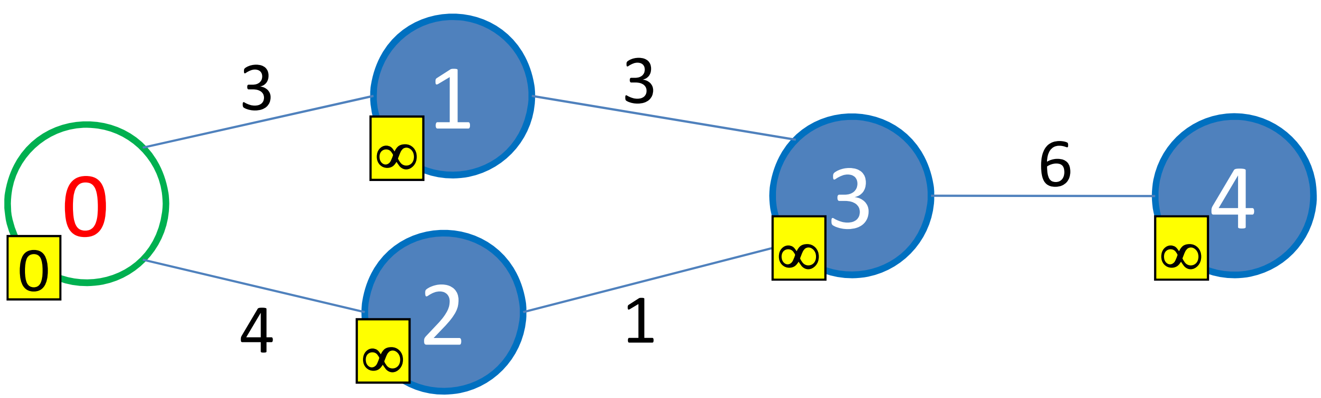
To think about: What if the source is not vertex 0 (first vertex in topological order?

Analysis of SSSP on DAG

- Pre-processing step: Topological sort
  - This can be done in O(V + E) using modified DFS as shown in Lecture 05
- Then, following this topological order (V items), relax a total of E edges
  - The total number of outgoing edges from all vertices = E
  - So again, it is O(V + E)
- In overall, SSSP on DAG can be solved in **linear time**: O(V + E)
  - Linear in terms of V and E

Why It Works? (1)

- On general graph, Bellman Ford's algorithm has to repeat this all-edges O(E) relaxation V-1 times
  - Thus Bellman Ford's runs in O(<u>**V**</u>E) time
    - Reason: there exist (non negative) cycles in general graph
    - After relax(u, v, w_u_v) is performed, there *may be* better other path *in the future* that reaches vertex **u** (the origin) so that this relax(u, v, w_u_v) has to be repeated…
    - We can only *be sure* after we have done this all-edges relaxation V-1 times (recall the proof of correctness of Bellman Ford's)



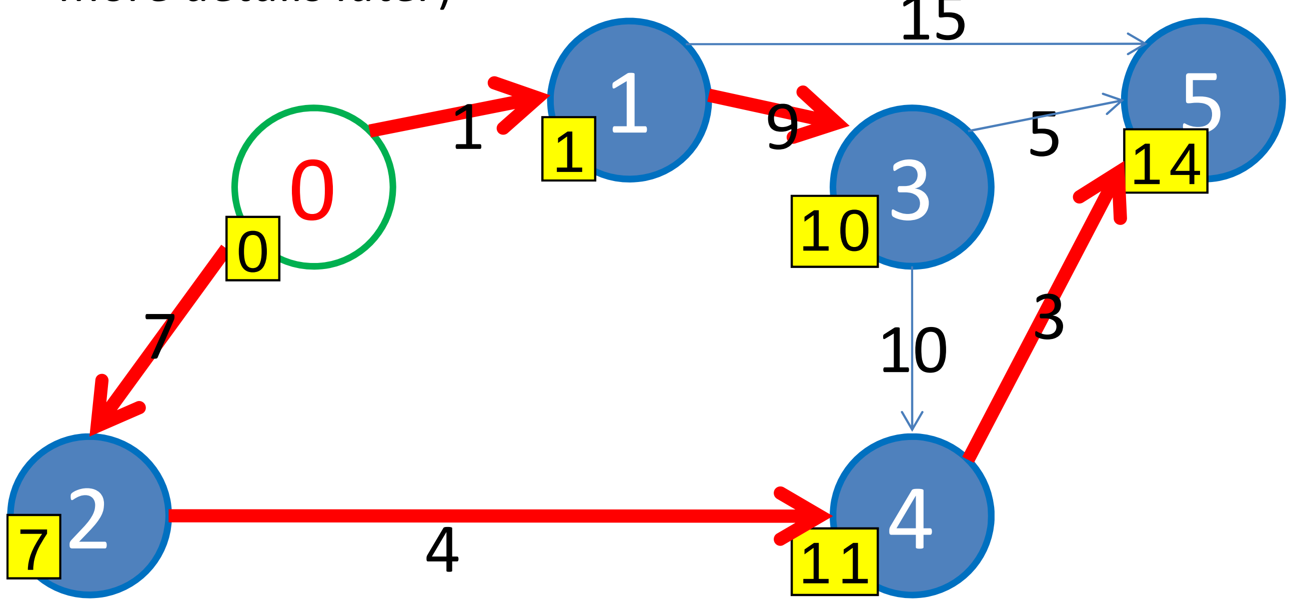
Assume edge ordering:
0-1
1-3
3-4
0-2
2-3

Why It Works? (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
  - If the vertices are processed according to this topological order, then after relax(u, v, w_u_v) is performed, there *will never be* any better other path in the future that reaches vertex **u** so that this relax(u, v, w_u_v) has to be repeated…
    - There is no way vertex **v** can reach back to vertex **u** because vertex **v** appears later in the topological ordering and there is **no cycle** that allows **v** ~→ some other vertices ~→ **u**!
  - Thus SSSP on DAG can be solved in O(V + E) time
    - We do not have to repeat this V-1 times ☺

Where is the Recursion/DP? (Part 1)

- Observe, for example, shortest path 0→2→4→5
  - Sub paths of this path, e.g. 0→2→4, are shortest paths too!
  - We do ***not*** re-compute these (clearly) overlapping sub paths
    - 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 with help of topological order of DAG, *more details later*)



SS LONGEST PATHS ON DAG

e.g. O(VE) of Bellman Ford's or O((V+E) log V Dijkstra's

If we can do SSSP problem efficiently, can we do **longest (simple) paths** on **general graph** too?

1. Yes, why not?
2. No, **longest** (simple) paths is not an easy problem because ________________

0 of 120

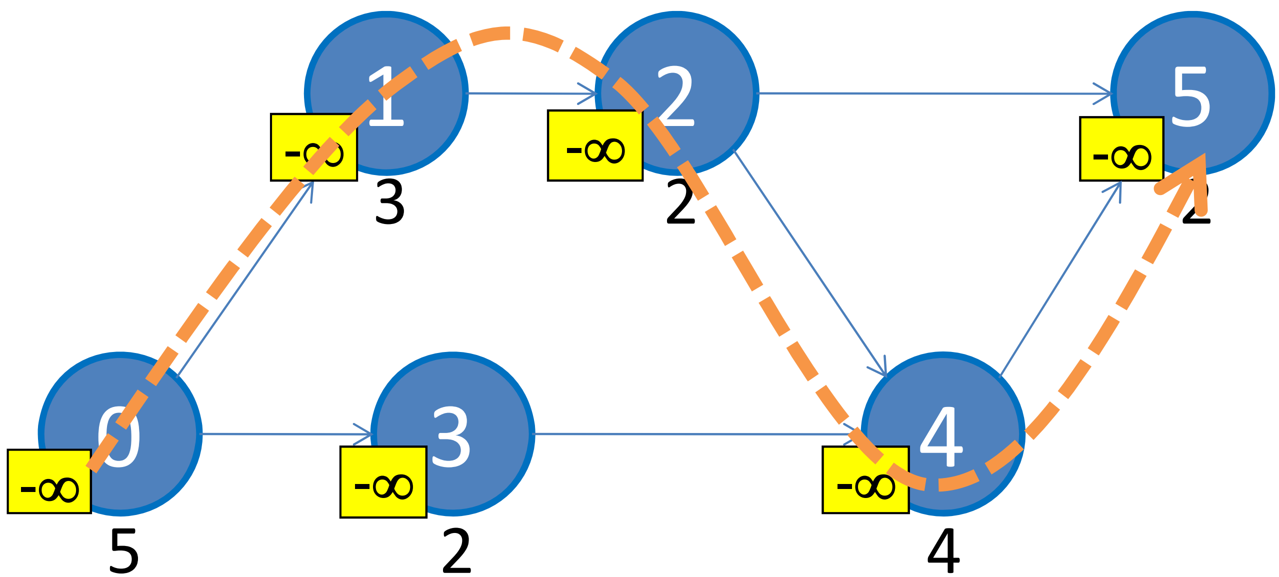
Longest Paths on DAG (1)

- Program Evaluation and Review Technique (PERT)
  - PERT is a project management technique
  - It involves breaking a large project into a number of 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)

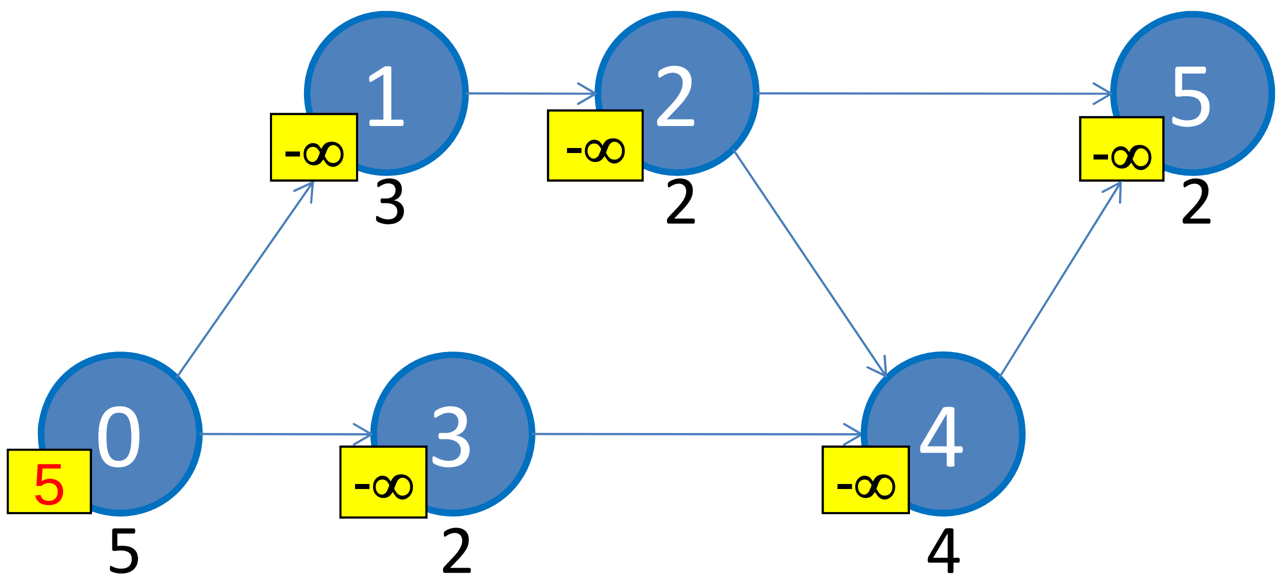
- Problem source: UVa 452 – Project Scheduling
  – Verify that this graph is a DAG!
    - Notice that the weight is **on vertices**, e.g. weight(0) = 5
  – The shortest way to complete this project is the…
    - **longest path** found in the DAG… (a bit counter intuitive)

Notice the -∞



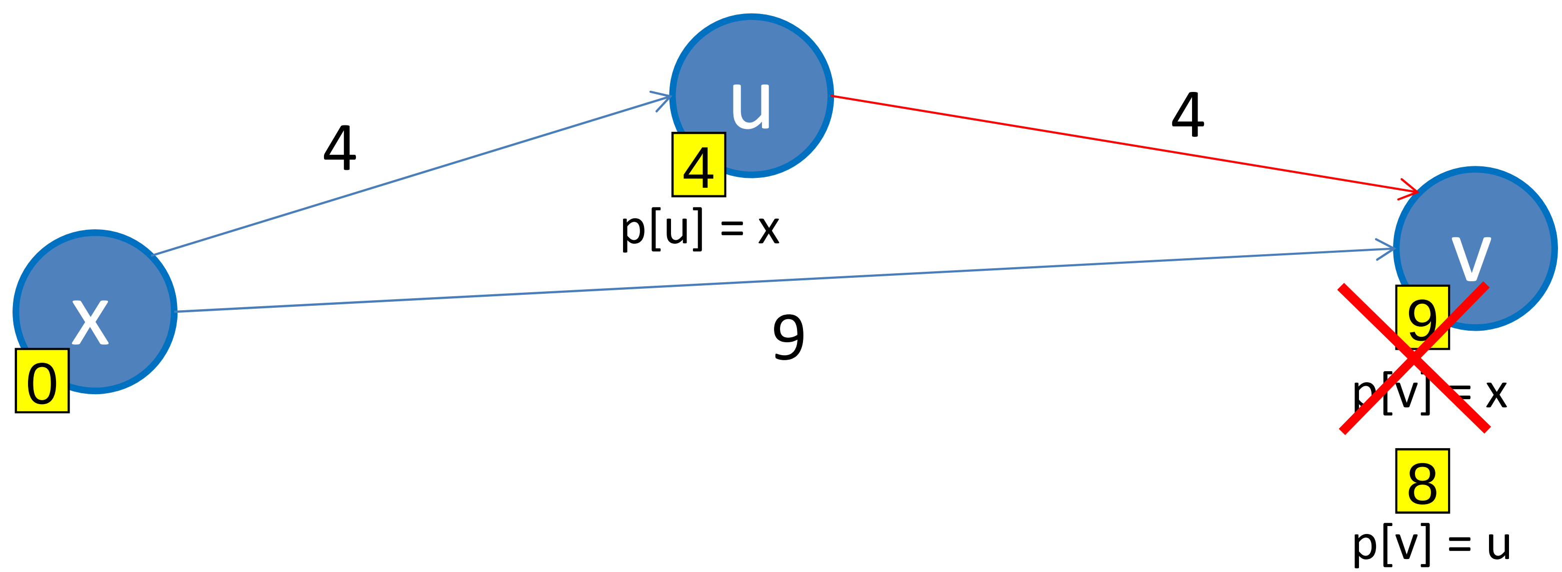
Longest Paths on DAG (3)

- First, find one topological order: {0, 3, 1, 2, 4, 5}
  - Can be found with O(V + E) modified DFS as in Lecture 5
  - Initially, set D[0] = weight(0) = 5
- Then "stretch" (antonym of "relax") the outgoing edges of the vertices listed in this topological order



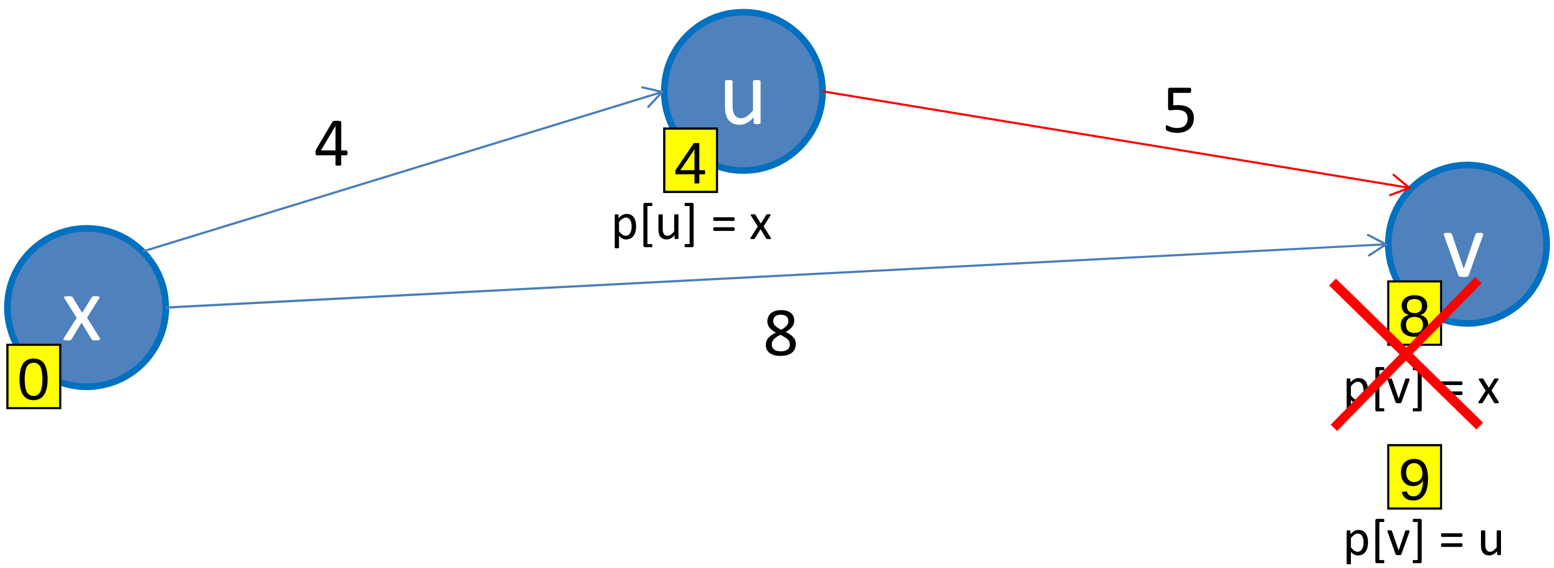
Review: “Relax” Operation

```
relax(u, v, w_u_v)
  if D[v] > D[u] + w_u_v // if SP can be shortened
    D[v] ← D[u] + w_u_v // relax this edge
    p[v] ← u // remember/update the predecessor
```



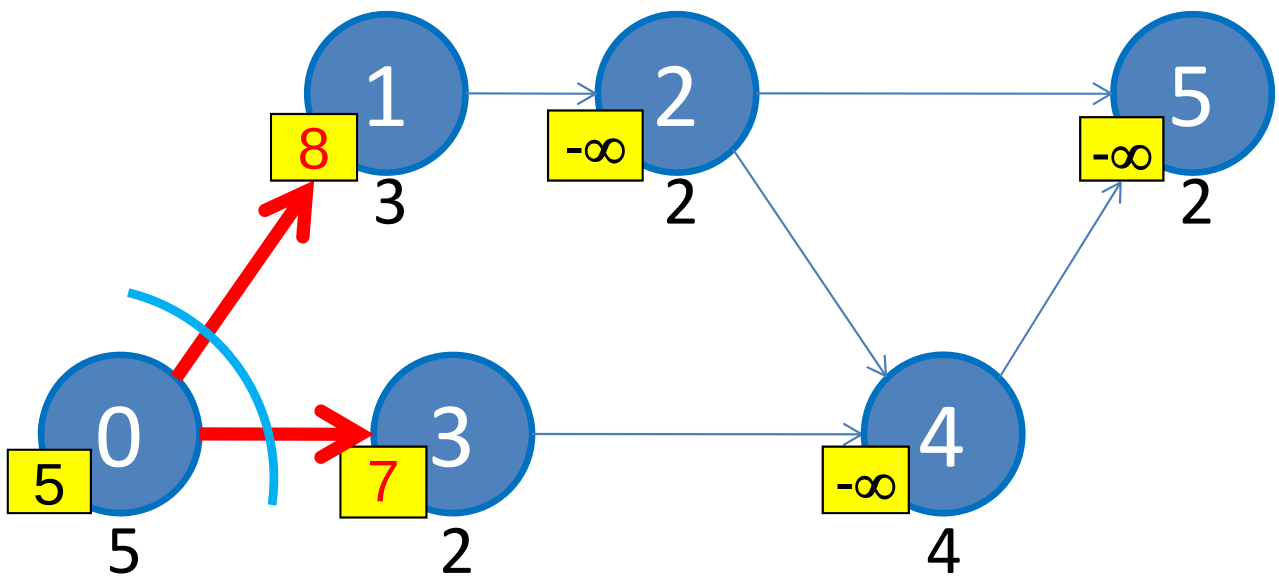
"Stretch" Operation

```
stretch(u, v, w_u_v)
  if D[v] < D[u] + w_u_v // if LP can be lengthened
    D[v] ← D[u] + w_u_v // stretch this edge
    p[v] ← u // remember/update the predecessor
```



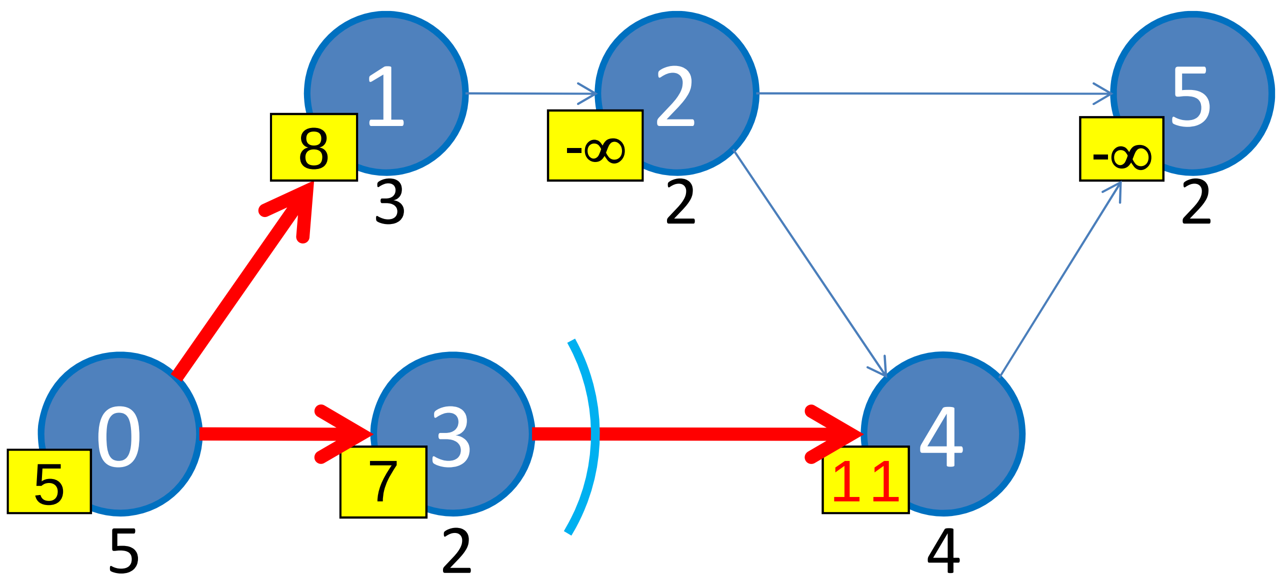
Longest Paths on DAG (4)

- The topological order: {0, 3, 1, 2, 4, 5}
  - Now relax outgoing edges from vertex 0
  - A tweak to transform vertex to edge weight in this problem:
    - Set distance of source to be the weight of source, e.g. D[0] = weight(0); then use the weight of destination vertex as the “edge weight” e.g. stretch(0, 1, weight(1)); stretch(0, 3, weight(3))



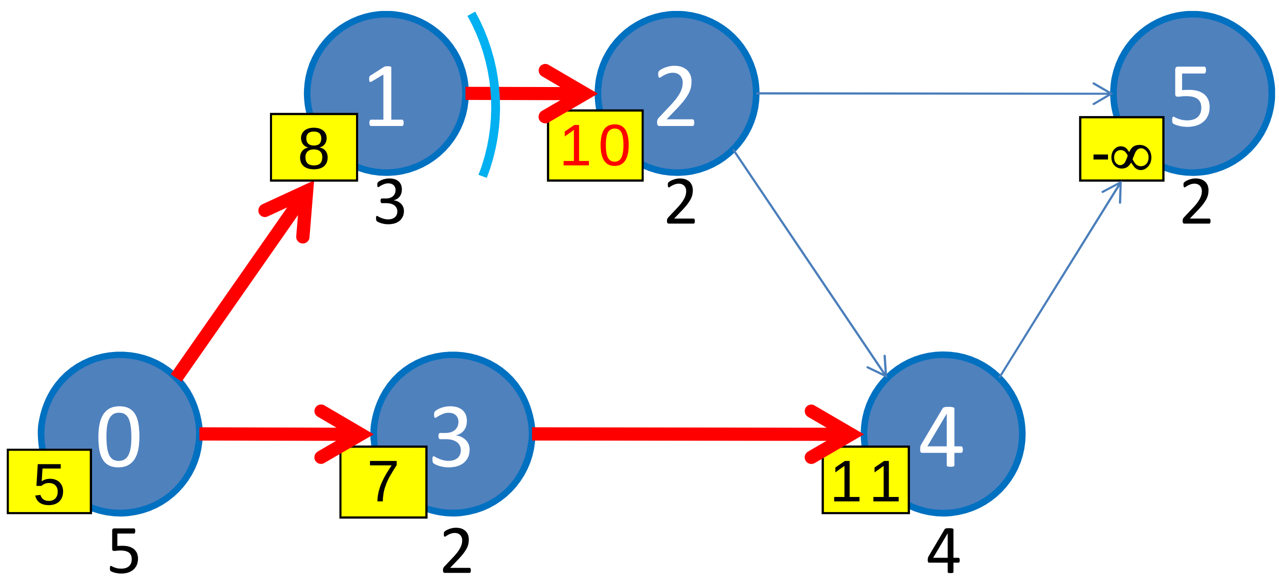
Longest Paths on DAG (5)

- The topological order: {0, 3, 1, 2, 4, 5}
  - Continue with vertex 3



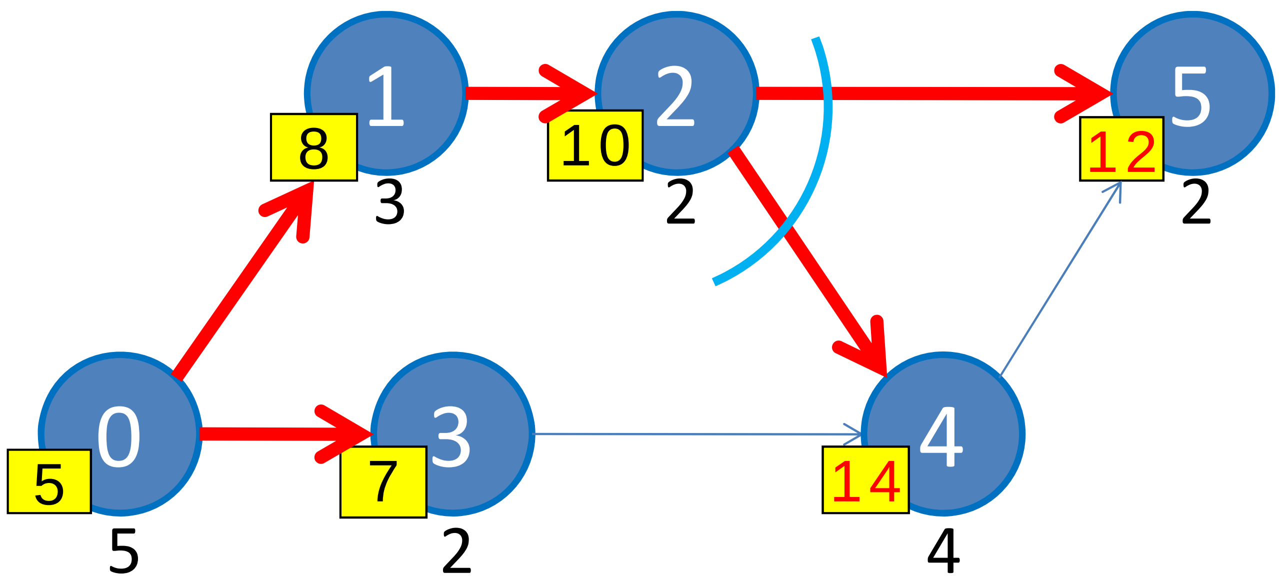
Longest Paths on DAG (6)

- The topological order: {0, 3, 1, 2, 4, 5}
  - Continue with vertex 1



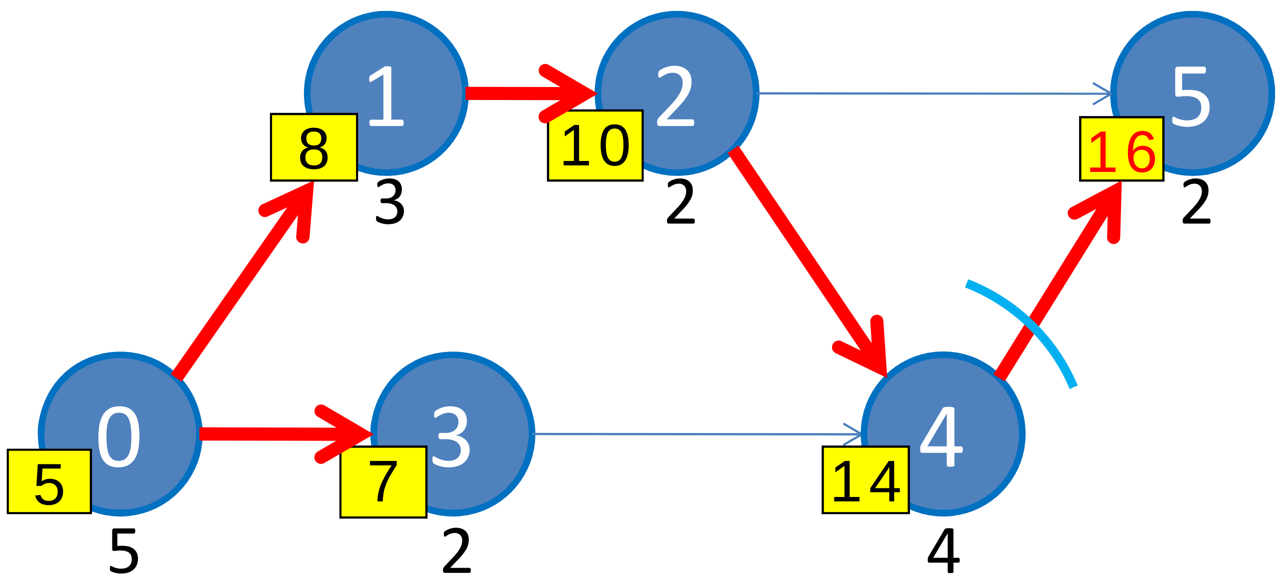
Longest Paths on DAG (7)

- The topological order: {0, 3, 1, 2, 4, 5}
  - Continue with vertex 2
  - Notice (again) that the vertices that have been processed so far, i.e. {0, 3, 1, 2} already have the correct final longest path values ☺, we do not have to re-trace our steps



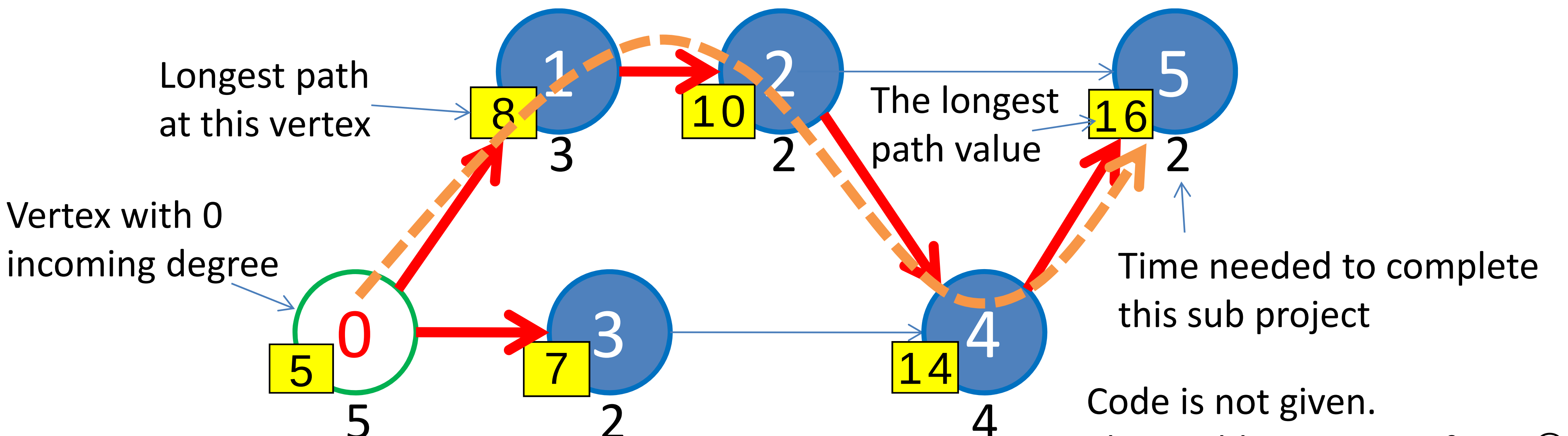
Longest Paths on DAG (8)

- The topological order: {0, 3, 1, 2, 4, 5}
  - Question: Can we stop here? i.e. the second last vertex?
    - Give a proof or provide counter example!



Longest Paths on DAG (9)

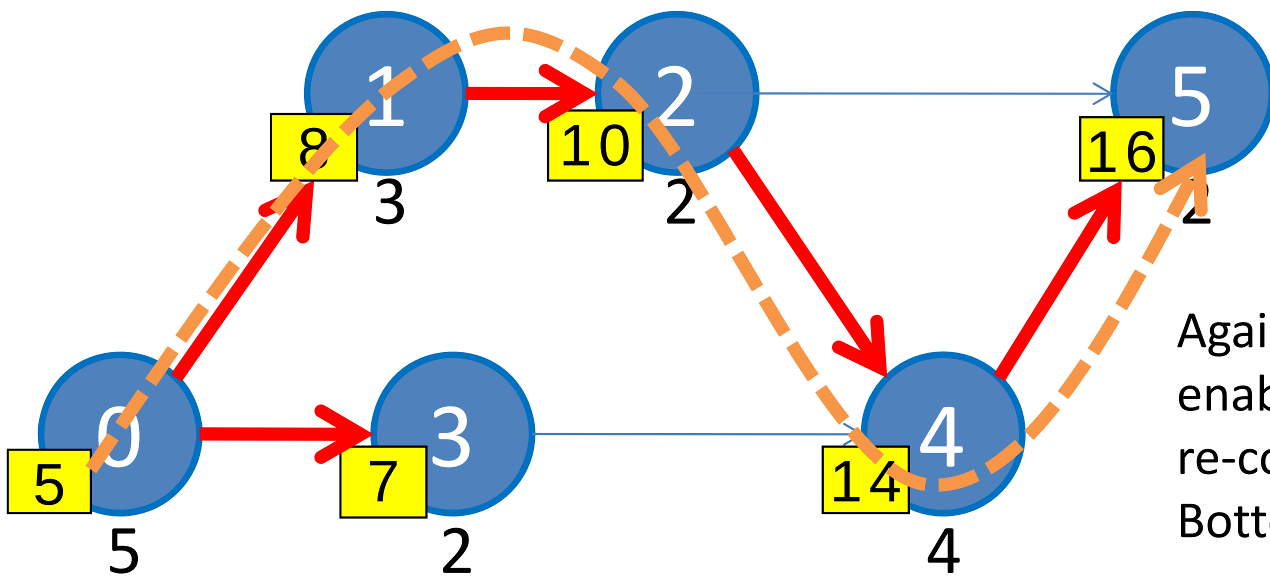
- The topological order: {0, 3, 1, 2, 4, 5}
  - Final solution, again the **thick red edges** are the LP Sp Tree
  - Scan the whole D[v], find the largest one
    - In this example D[5] = 16 is the largest
    - Use predecessor information (the **thick red edges**) to reconstruct the longest path: 0→1→2→4→5



Code is not given.
This problem is part of PS6 ☺

Where is the Recursion/DP? (Part 2)

- These two major ingredients for DP technique are also present in the SSLP on DAG problem:
  - Optimal sub-structure
    - Sub paths of longest paths **<u>on DAG</u>** are longest paths too
  - Overlapping sub-problem
    - Longest path 0->1->2->4->5 contains longest path 0->1->2->4, etc



Again, topological order enable us to avoid re-computations: Bottom-up DP

Analysis of Longest Paths on DAG

(The same as SSSP on DAG)

- Pre-processing step: topological sort
  - This can be done in O(V + E) using modified DFS
- Then, following this topological order (V items), "stretch" a total of E edges
  - Again, it is O(V + E)
- In overall, longest paths on DAG can be solved in linear time: O(V + E)
  - Linear in terms of V and E

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

Have you heard about LIS?

5 minutes break after this

1. This is my first time, tell me!
2. Yes, but only the problem, not the solution…
3. Yes, I know the $O(n^2)$ algorithmic solution for LIS
4. Yes, I have solved/coded some $O(n^2)$ LIS solution before
5. Yes, I have solved/coded some $O(n \log k)$ LIS solution before (if you say “why there is a log factor?”, do not select this)

0 of 120

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

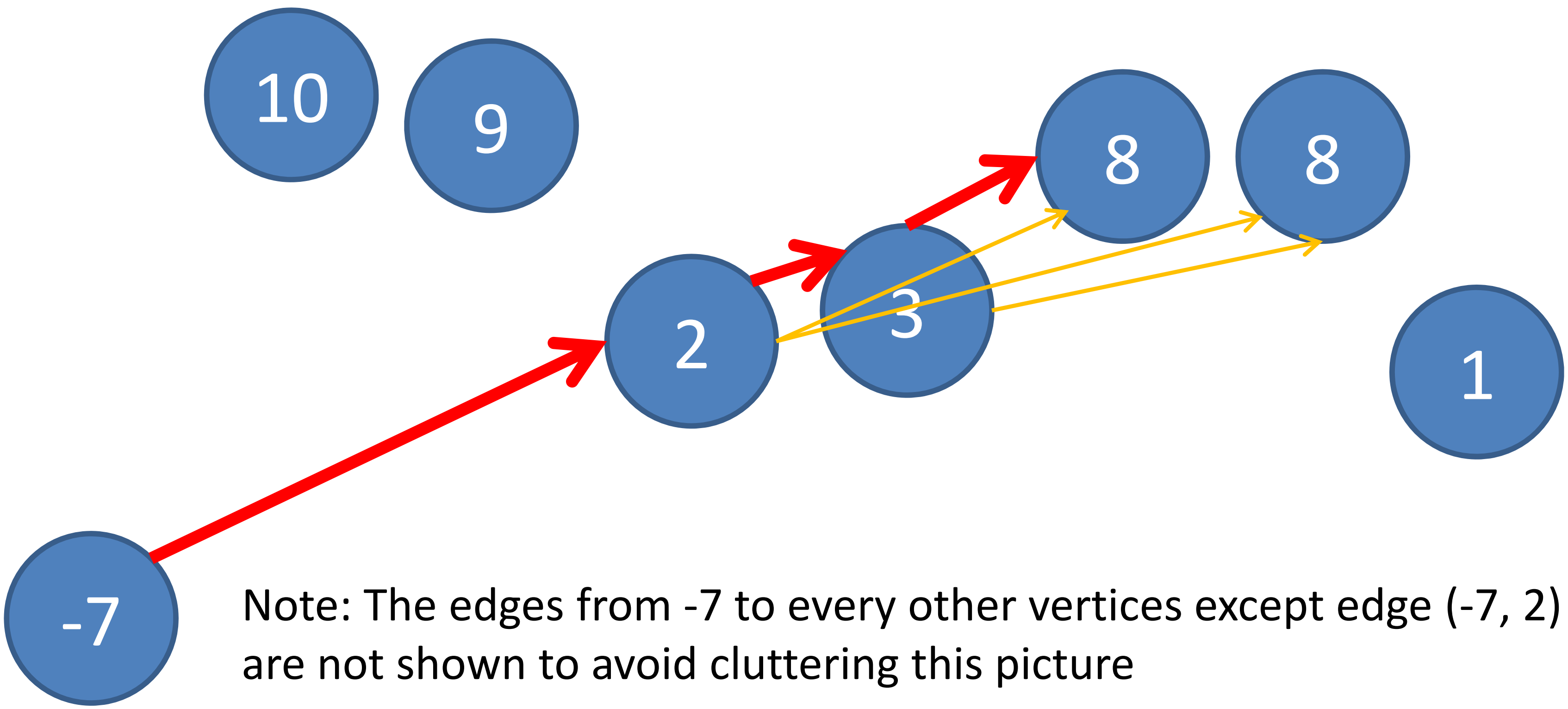
LONGEST INCREASING SUBSEQUENCE (LIS)

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) are not shown to avoid cluttering this picture

Longest Increasing Subsequence (3)

- Let D[i] = the best LIS ending at index (vertex) i (minus 1)
  - **Q1: Why there exists a minus 1?**
- The topological order is obviously {0, 1, 2, …, N - 1}, **Q2: Why?**
  - "Stretch" all index (vertex) one by one using this order

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

- The answer is `max(D[i]) + 1`, `∀i ∈ [0 ... N - 1]`
  - **Q3: Why we add plus 1 at the end?**

Longest Increasing Subsequence (4)

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

- This LIS problem can be solved in $O(n^2)$, 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 $E = n^2$ edges

Longest Increasing Subsequence (5)

- LIS is actually solvable in O(n log **k**)
  - Where **k** is the length of LIS
    - This is called "output-sensitive analysis"
- How?
  - Utilize the fact that the LIS is sorted
    - We can use binary search
  - A greedy solution
  - Not important for CS2010/CS2020
    - But it is for CS3233

Where is the Recursion/DP? (Part 3)

- This LIS problem is more naturally solved in "Top-Down Dynamic Programming (DP)" fashion
  - Let **LIS(i)** be the value of the longest LIS starting from index i until N - 1
    - 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 $X[i] < X[j]$
  - To avoid recomputations, memoize the LIS value of each index/vertex **i**
    - This term "memoize" (memo table) will be explained soon

Where is the Recursion/DP? (Part 4)

- 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.get(i) < A.get(j))
      ans = Math.max(ans, LIS(j) + 1);
  return ans;
}
```

Turn Recursion into Memoization

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
}
```

Where is the Recursion/DP? (Part 5)

- A better version (see LISDPDemo.java):

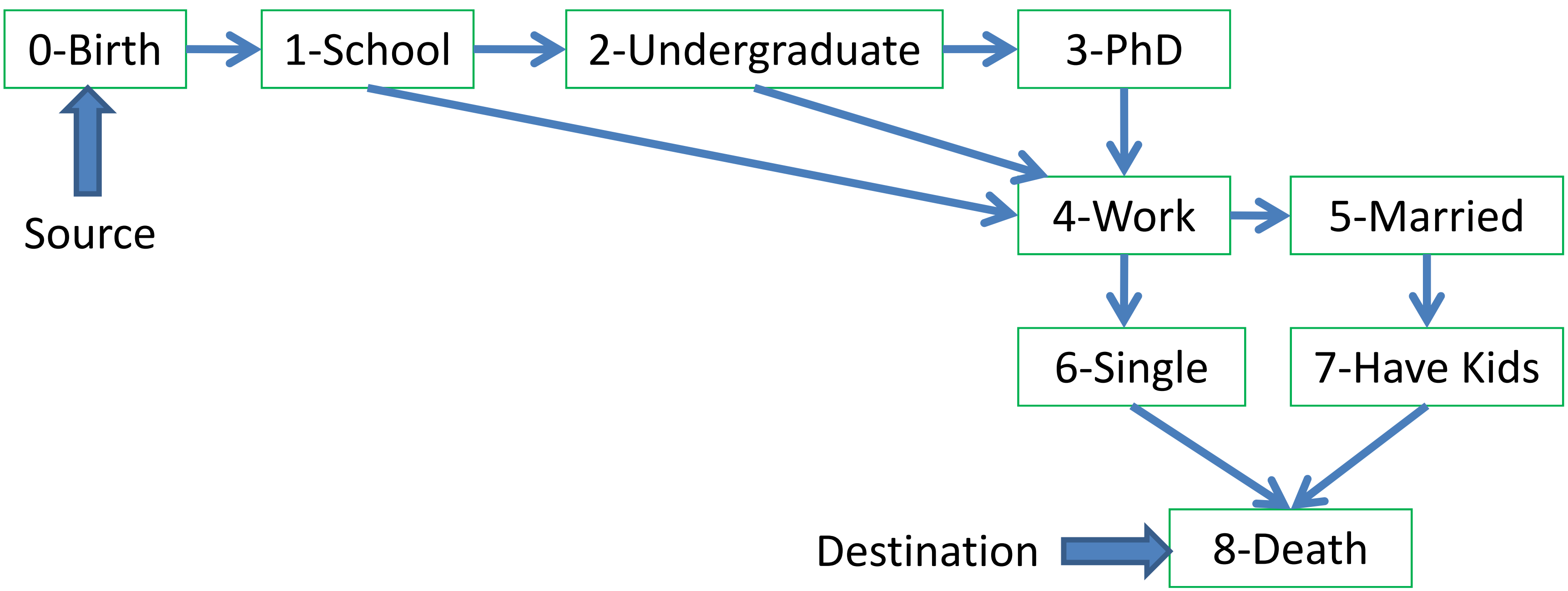
```
private static int LIS(int i) {
  if (i == N - 1) return 1;
  if (memo.get(i) != -1) return memo.get(i);

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

Final discussion for today, again about DAG ☺

COUNTING PATHS ON DAG

Counting Paths on DAG

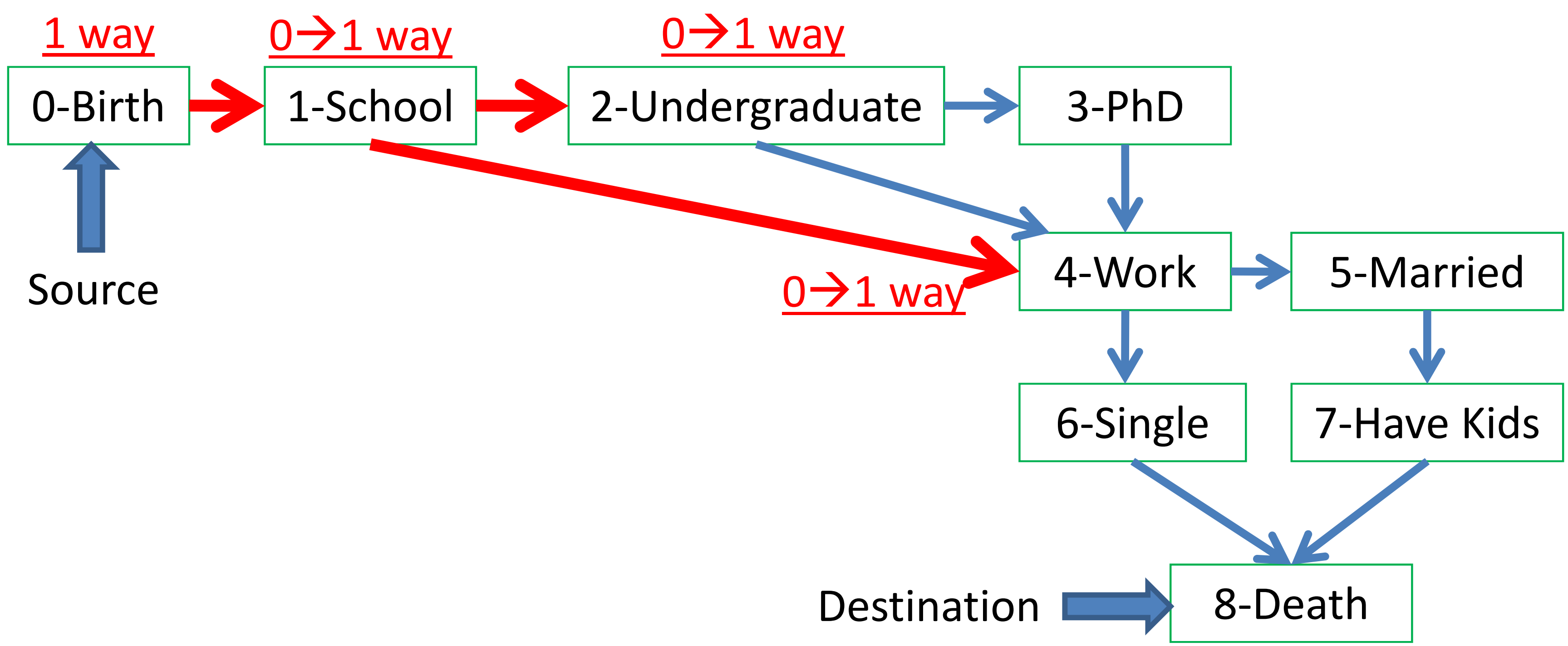
- Given some real-life time line (obviously a DAG)
  - How many different possible lives that you can live (from birth/vertex 0 to death/vertex 8)?



Answer = 6

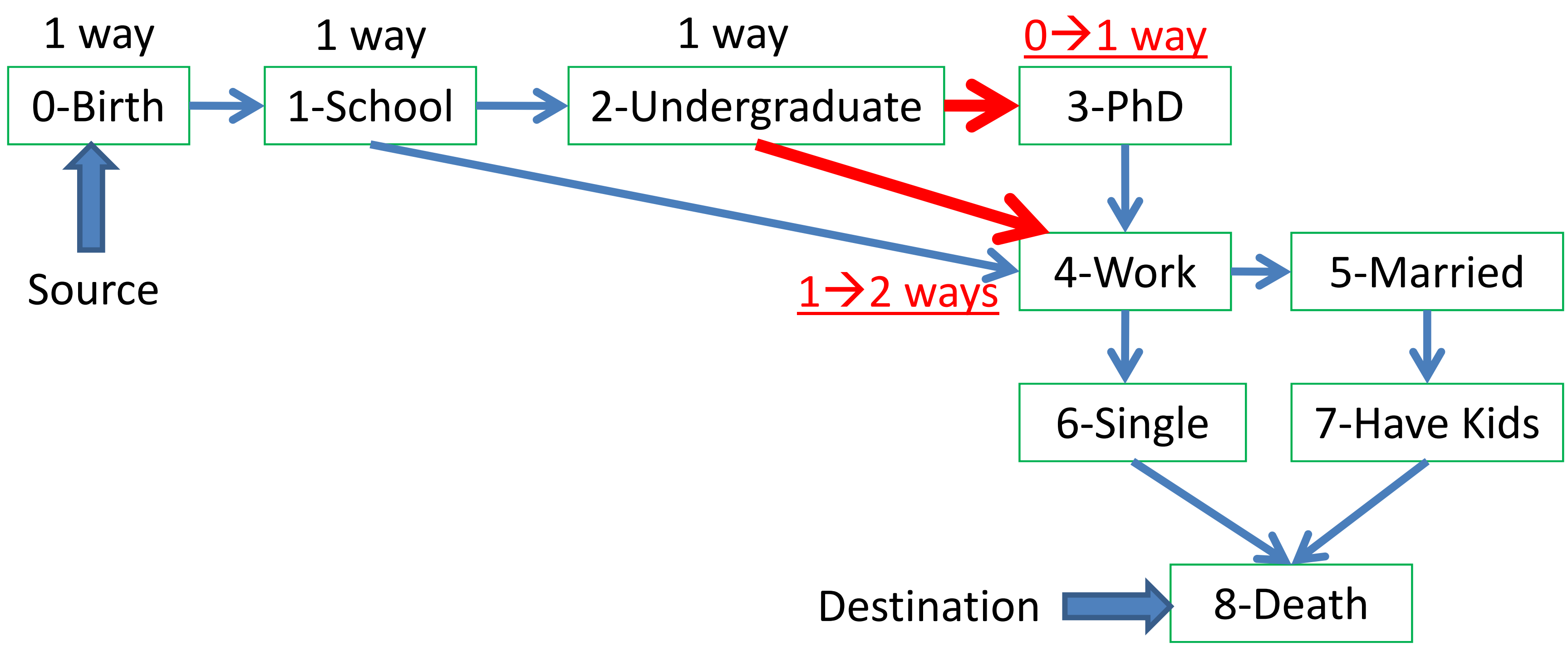
Toposort/Graph/Bottom-Up Way (1)

- Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}
  - numPaths[0] = 1, propagate to vertex 1, and then 2, 4



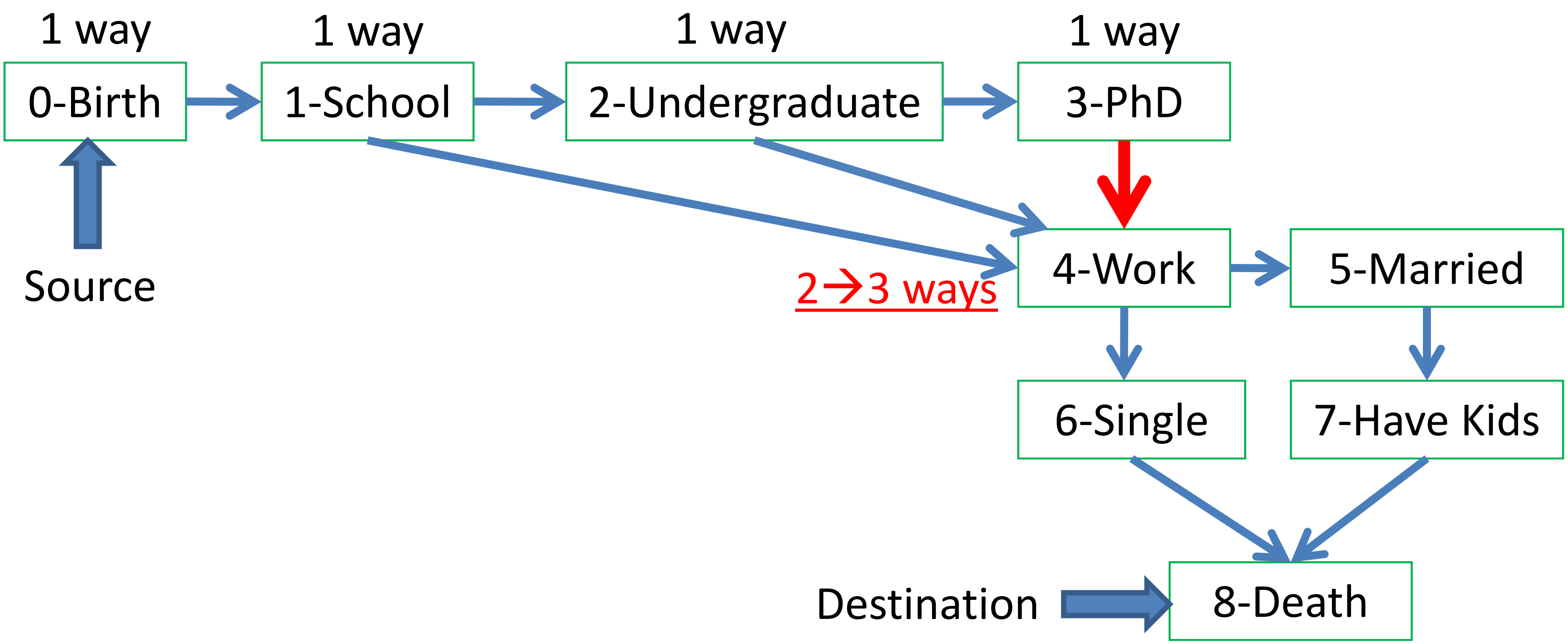
Toposort/Graph/Bottom-Up Way (2)

- Find the toposort first: {<u>0, 1, 2</u>, 3, 4, 6, 5, 7, 8}
  - numPaths[2] = 1, propagate to vertex 3 and 4



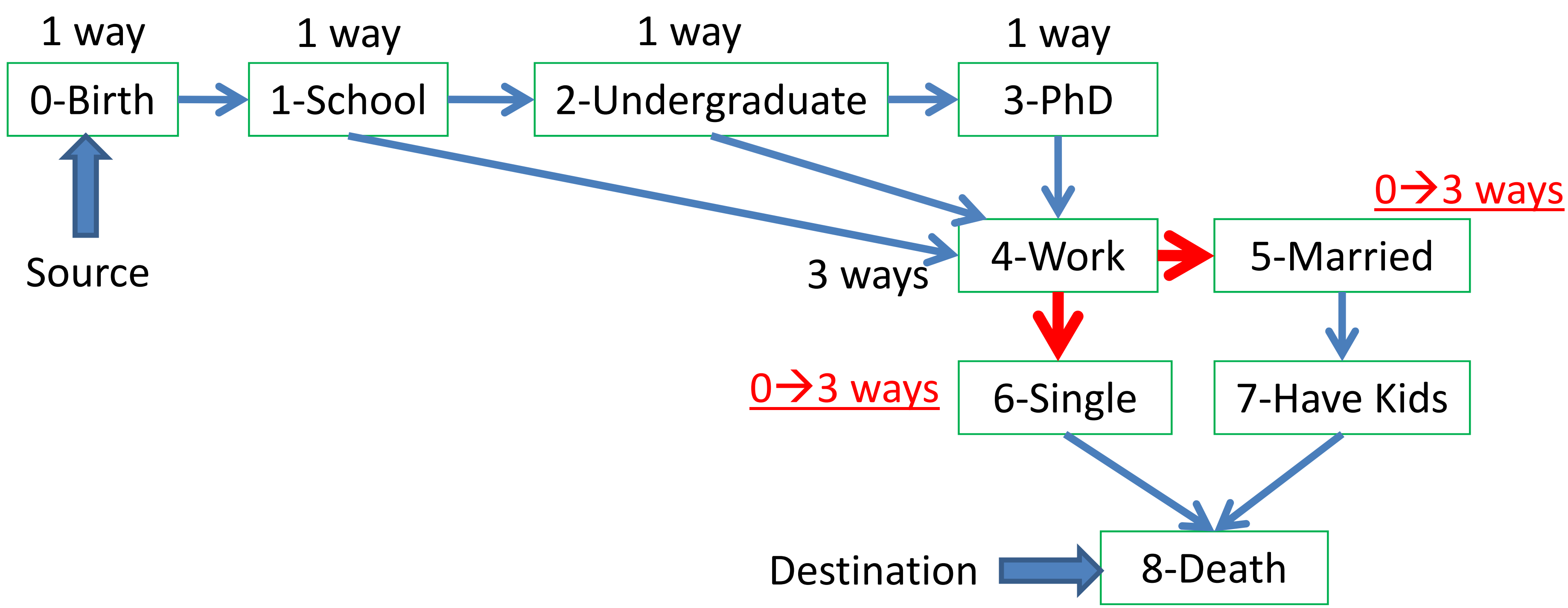
Toposort/Graph/Bottom-Up Way (3)

- Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}
  - numPaths[3] = 1, propagate to vertex 4



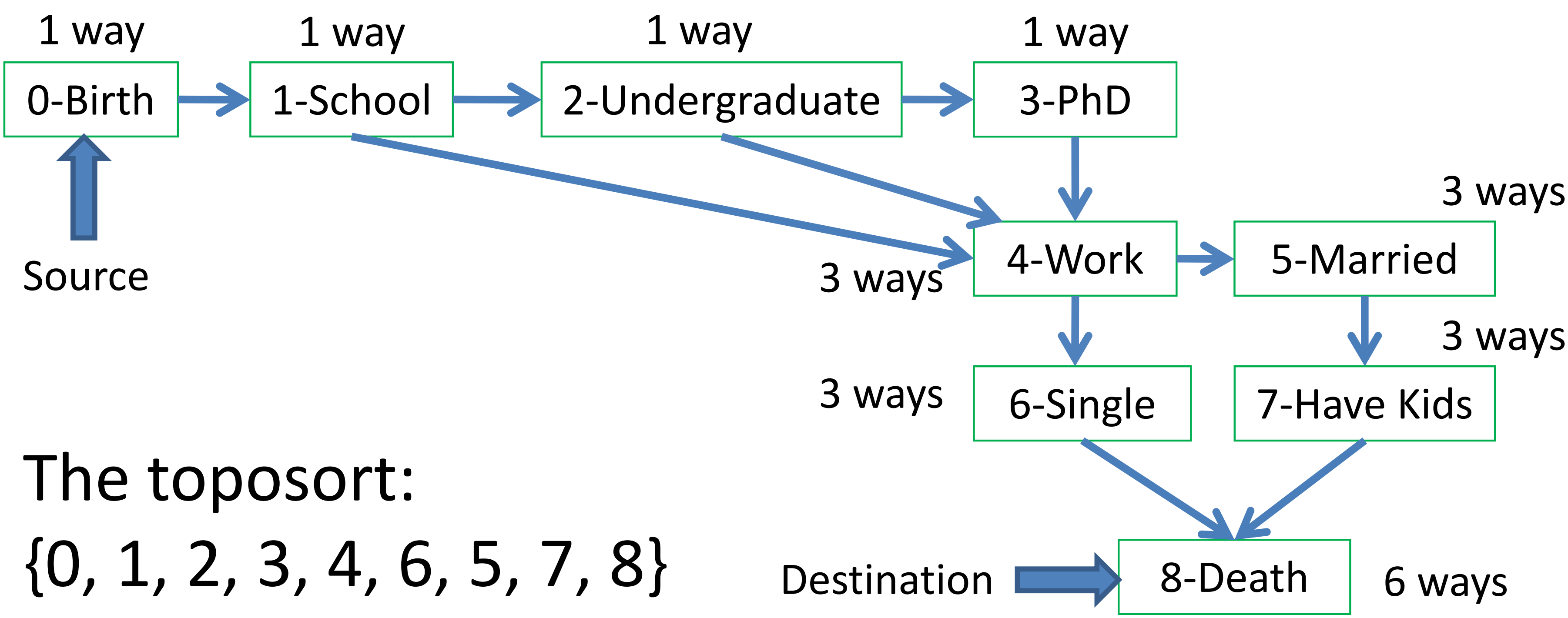
Toposort/Graph/Bottom-Up Way (4)

- Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}
  - numPaths[4] = 3, propagate to vertex 5 and 6



Toposort/Graph/Bottom-Up Way (5)

- Find the toposort first: {<u>0, 1, 2, 3, 4, 6, 5, 7, 8</u>}
  - >> >> Fast forward…, this is the final state



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

Where is the Recursion/DP? (Part 6)

- 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**
  - 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

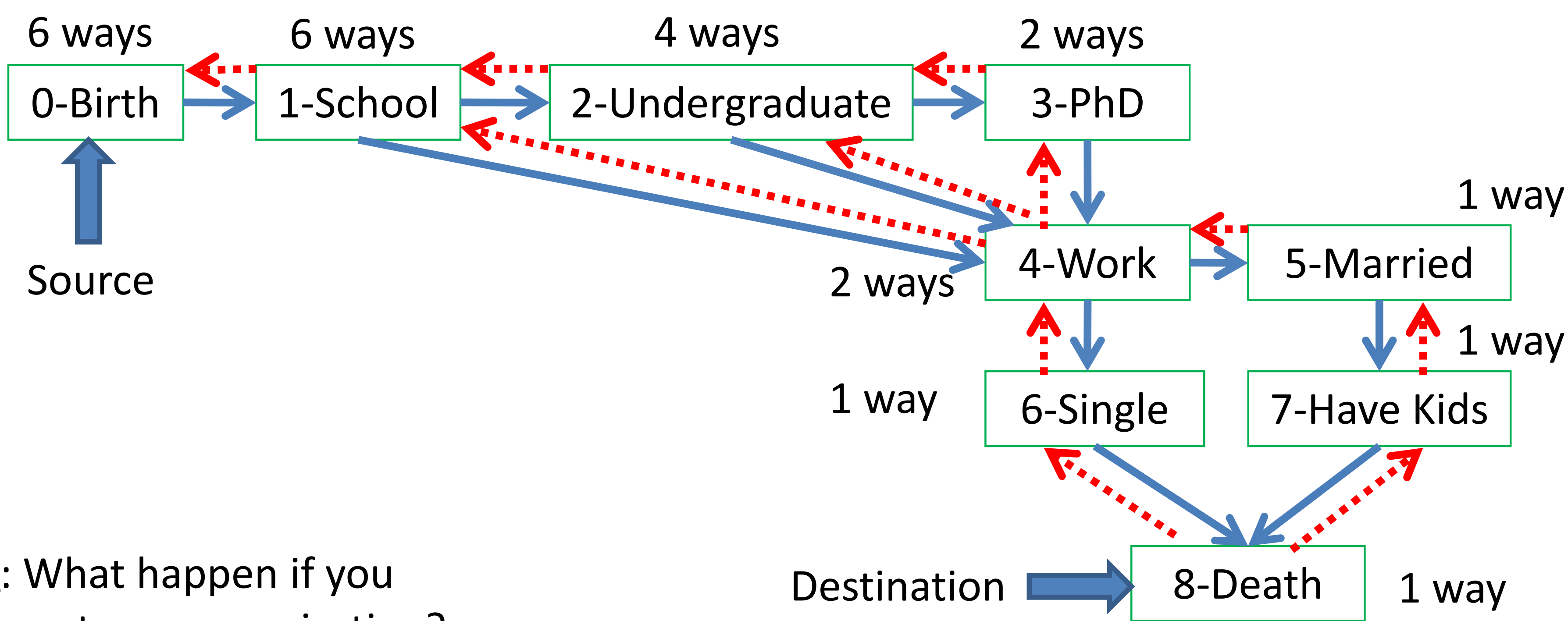
Where is the Recursion/DP? (Part 7)

- The (Java) recursive function

```
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;
}
```

Where is the Recursion/DP? (Part 8)

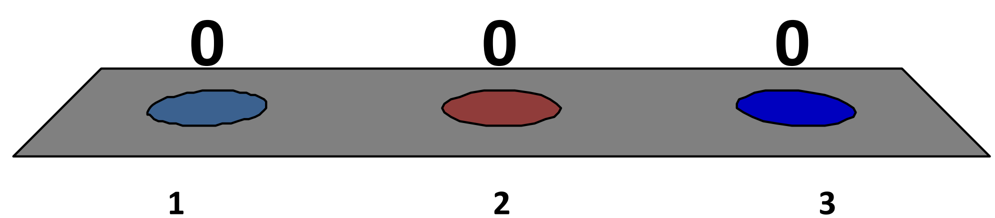
- The way the answer is computed is now from destination to source



Q: What happen if you do not use memoization?

That’s the **preview** of DP

1. Manageable
2. Scary
3. Very scary…



0 of 120

Summary / 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)
  - In the next 2 lectures, 3 tutorials, and 3 PSes, we will use more implicit DAGs (on more structured problems) and use more DP terminologies rather than graph terminologies ☺
    - Ingredients:
      - Optimal sub-structure and Overlapping sub-problem
    - Terminologies:
      - Vertices → States; Edges → Transitions
      - |V| → Space Complexity; |E| → Time Complexity