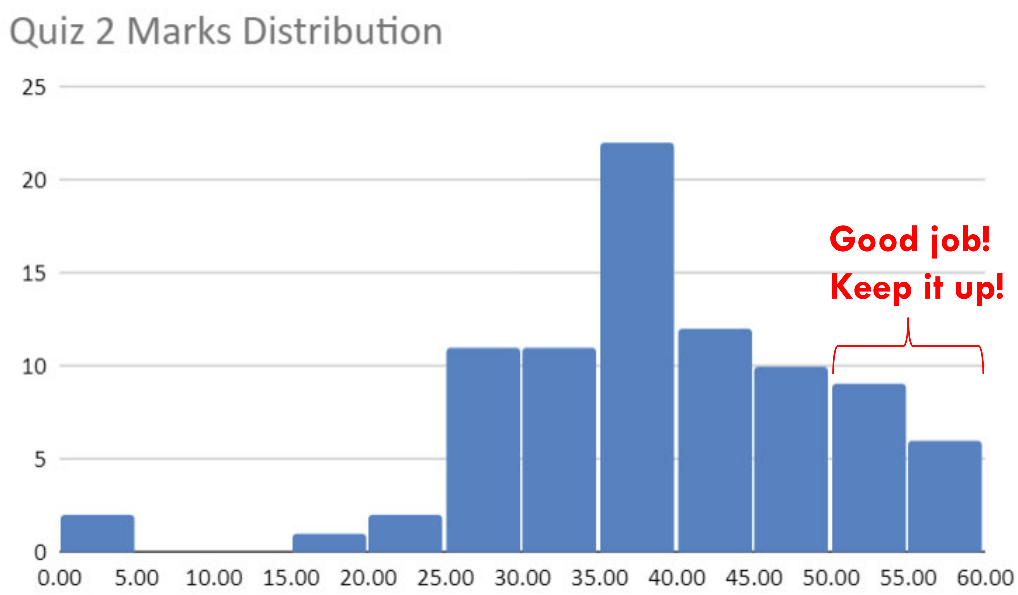
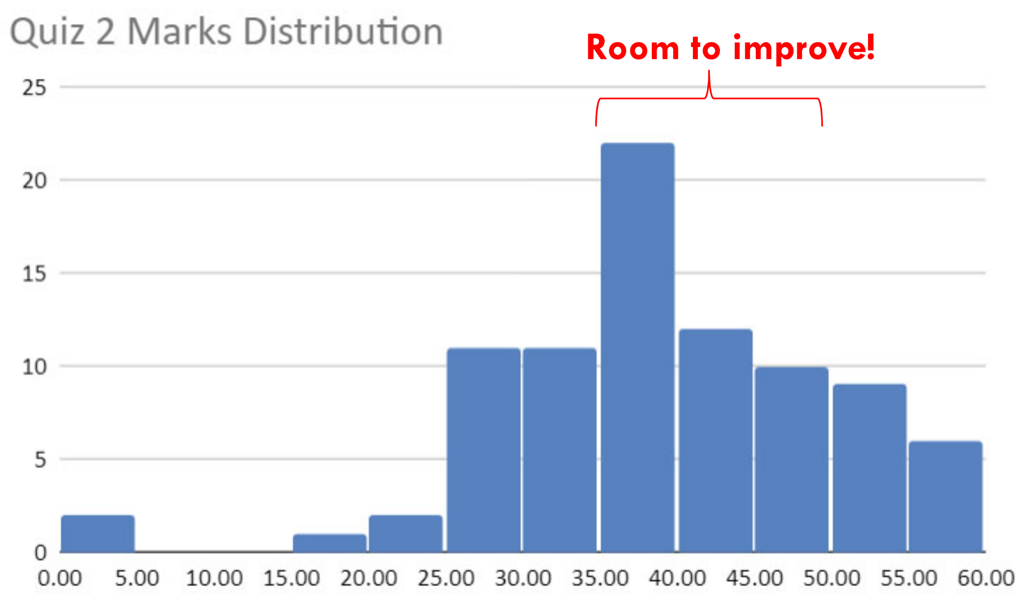
# LECTURE 14: SHORTEST PATHS (SPECIAL CASES)

Harold Soh
harold@comp.nus.edu.sg

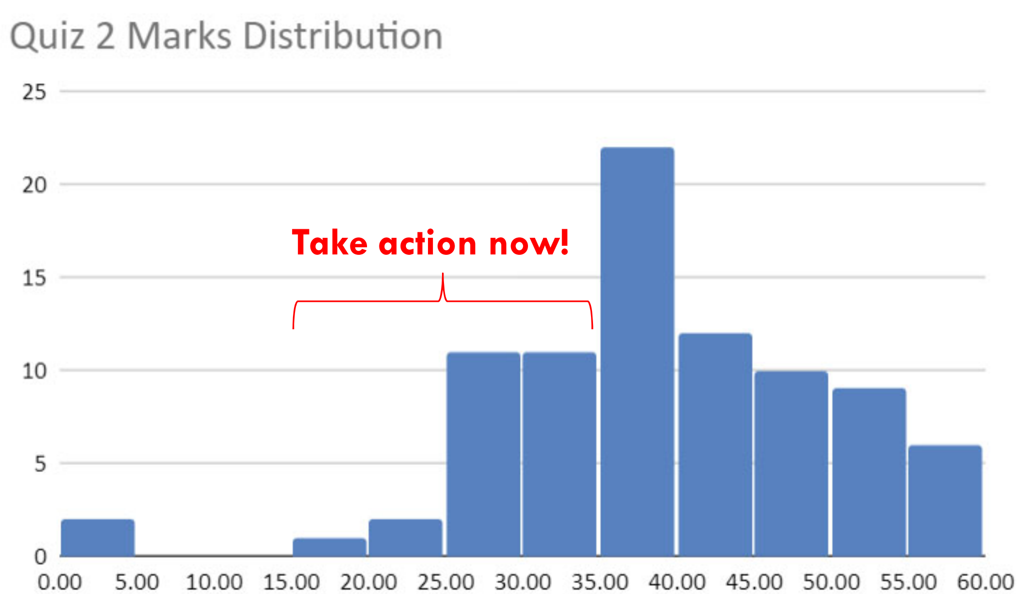
# ADMINISTRATIVE ISSUES: QUIZ 2



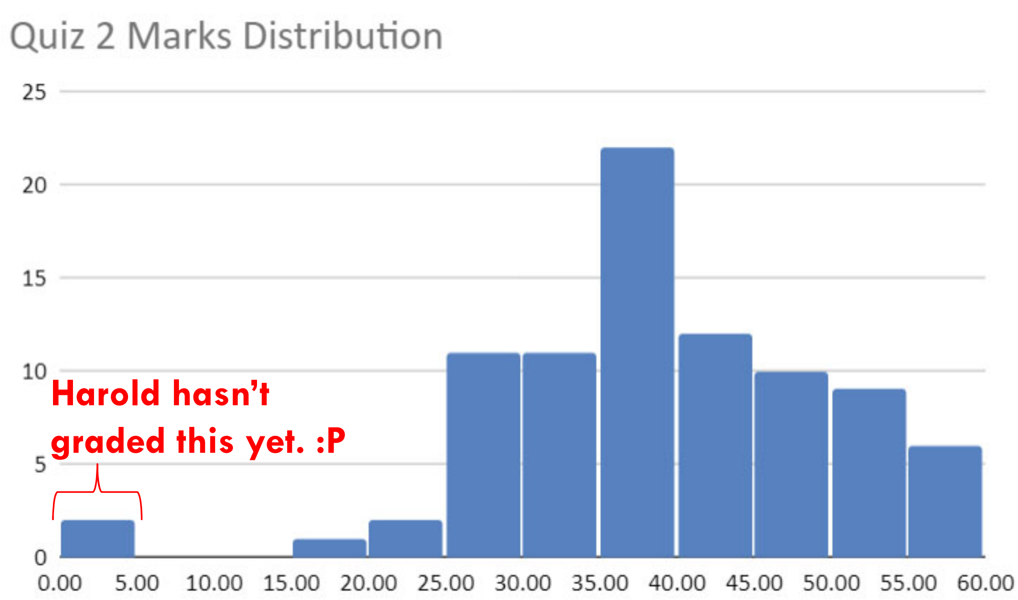
# ADMINISTRATIVE ISSUES: QUIZ 2



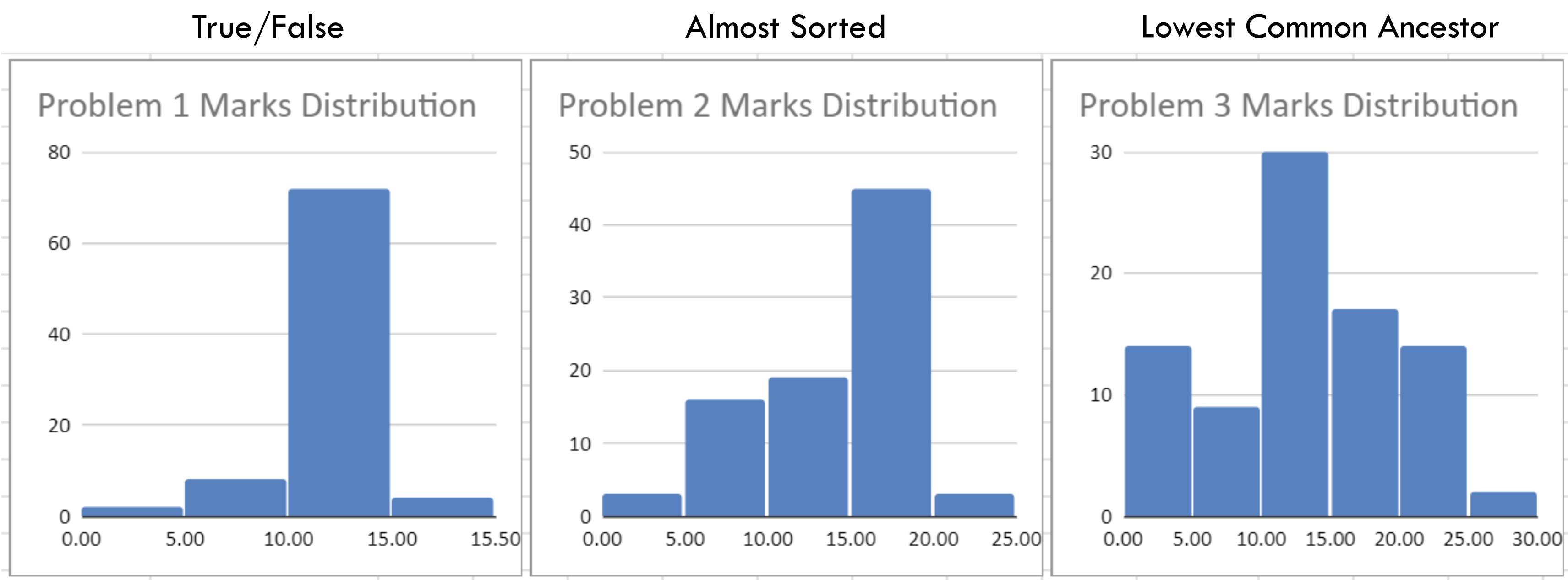
# ADMINISTRATIVE ISSUES: QUIZ 2



# ADMINISTRATIVE ISSUES: QUIZ 2



# GRADE DISTRIBUTION: SPECIFIC PROBLEMS



# QUIZ 2: LCA

In CS2040S, we learnt about ancestors and descendants in Binary Trees. For this problem, we will focus on the *lowest common ancestor* (LCA). The LCA of two nodes $a$ and $b$ is the node furthest from the root that is an ancestor of both $a$ and $b$. For example, given the Binary Tree shown in Fig 1, the LCA for 2 and 6 is 5. The LCA of 12 and 2 is 10. Your task is to design algorithms that compute the LCA given a Binary Tree $T$ with $n$ nodes and has height $h$.

**Problem 3.a.** [15 points] **Describe the most efficient algorithm to find the LCA given two nodes $a$ and $b$ in a Binary Tree $T$.** Prioritize time over space efficiency. **Each node does *not* have a parent pointer**. **Your method should work on any binary tree and not just binary search trees**.

Write the time and space complexity of your method below. Recall that $T$ has $n$ nodes and height $h$.

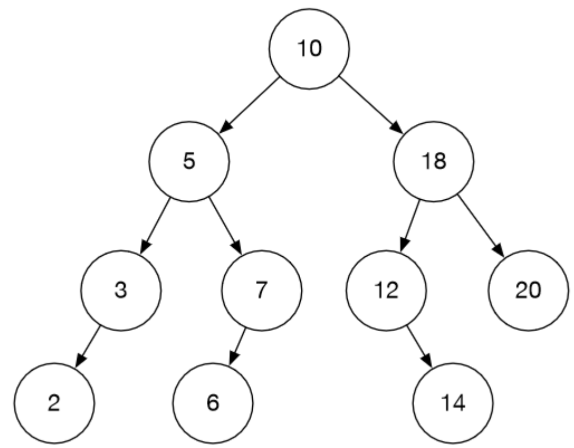


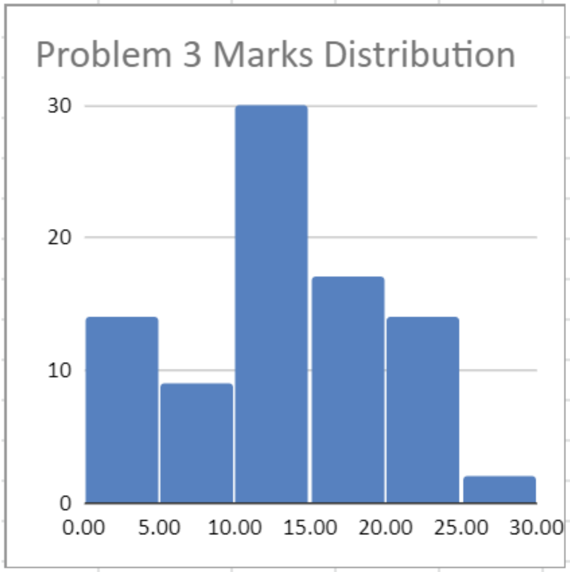

**Figure 1**: An example Binary Tree.

# GRADE DISTRIBUTION: SPECIFIC PROBLEMS

Problem 3 Key Issues:

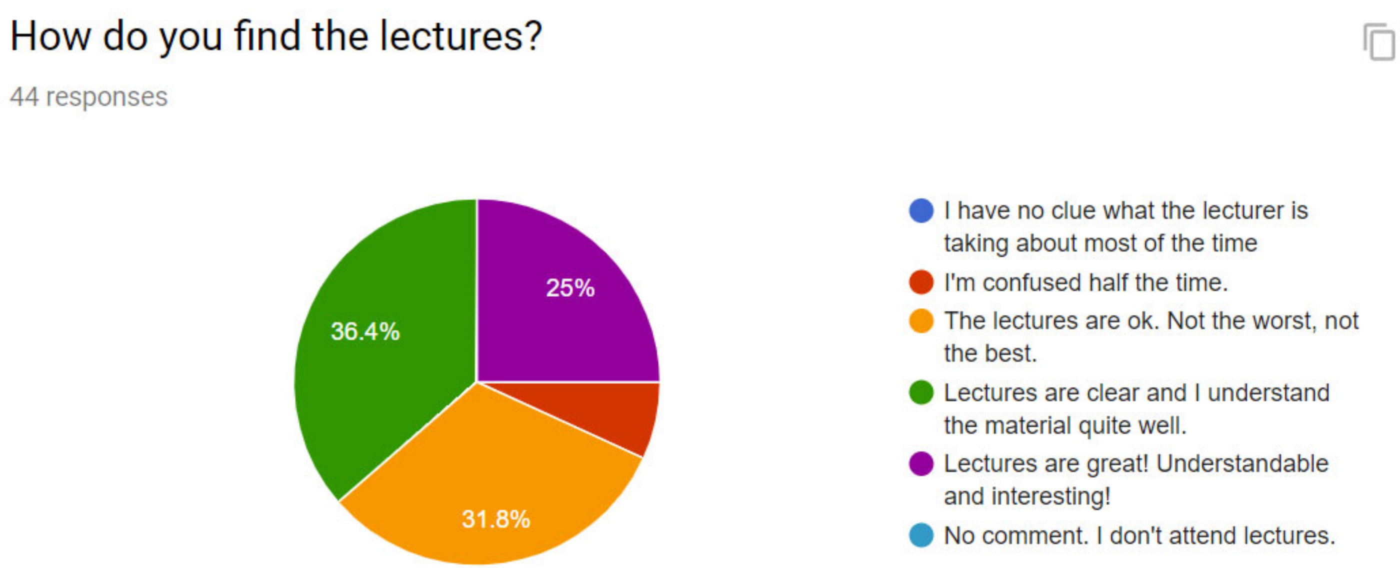
- *Binary Tree* v.s. Binary **Search** Tree v.s. **Balanced** Binary Search Tree
- No parent pointers!
- Please read instructions **very** carefully.

Tutorial will go over problems again.



# ADMINISTRATIVE ISSUES: MID-SEMESTER SURVEY

44 Respondents (out of 86 students, 51%)



# FEEDBACK

Speed check:

- "talk slower, its too fast"
- "…sometimes the solution is discussed quite quickly or briefly and it's easy to miss"
- "Right now the pace is rather fast for me. I would prefer if it were slower."
- "Slow down the pace by a little (esp the maths parts) for students to better digest and appreciate the material"

Live Questions:

- "Include live question polls online during lecture for students to ask questions periodically"


Ok! Will slow down and do periodic polling to ask for questions.

# FEEDBACK

Webcasts:

- “Webcast pls”
- “Webcasts for every lecture.”
- ““"I understand that webcast is not provided for students to attend the lectures. But perhaps it could be posted after the lecture week? I find it hard to revise for the quiz without the recorded lecture. :(“
- “Release webcast faster”

Others

- “The only issue I have is with the content of the course. It seems to be way too heavy.”
- “Tell me how to answer your exam questions.”

Web Lectures

Web lectures are video recordings of lectures that are conducted at NUS lecture theatres. They are used to reinforce revision and study. Click here for more information on web lectures.

Data Structures and Algorithms

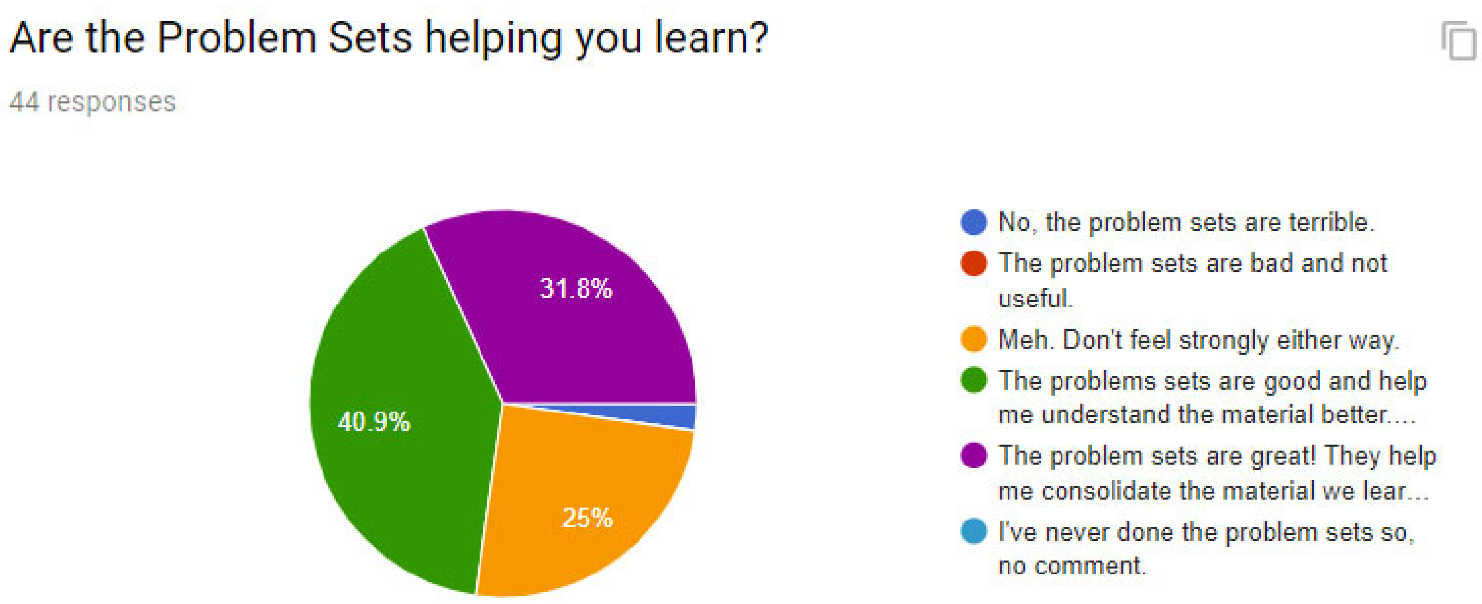
| Lecture Title | Lecture Date/Time | View Count | |
|---|---|---|---|
| CS2040S Hashing (Part 2) on 9/18/2019 (Wed) | 18 Sep 2019 12:00 | 10 | ... |
| CS2040S Hashing (Part I) on 9/17/2019 (Tue) | 17 Sep 2019 12:00 | 12 | ... |
| CS2040S Balanced Binary Search Tree(AVL) on 9/11/2019 (Wed) | 11 Sep 2019 12:00 | 27 | ... |
| CS2040S Binary Search Trees on 9/10/2019 (Tue) | 10 Sep 2019 12:00 | 21 | ... |
| CS2040S Heaps on 9/3/2019 (Tue) | 3 Sep 2019 12:00 | 46 | ... |

Will *try* to release webcasts faster

Sorry! No answers in advance. ☹

But we cover answers after!

# PROBLEM SETS



# PROBLEM SETS

Difficulty and Timings

- “Its abit too difficult and place at wrong timings.. its so hard to do and so time consuming and worst part of all, placed on weeks with exams/submission dates”
- “They should be released after we learn the concept (Not before/during)”
- “Maybe have a spectrum of difficulties for the problem sets, instead of all difficult”
- “Would be more useful if they're closer to what we're learning”

Sorry, I can’t make the problem easier.
We include 2-3 “easy/medium” problems, and 1 “medium” problem.
Based on material covered.

Timings: Please start early. 2 weeks is as long as I can make it.

# PROBLEM SETS

Kattis Output

- "Not being able to see what went wrong really makes debugging a VERY big pain"
- "I don't like the binary pass/fail system Kattis has. I like to see the output of the test cases as I want to know where my code failed."
- "It would be nice to know expected output and our output when solution is wrong. But actually this is good for us to think of our own test cases."

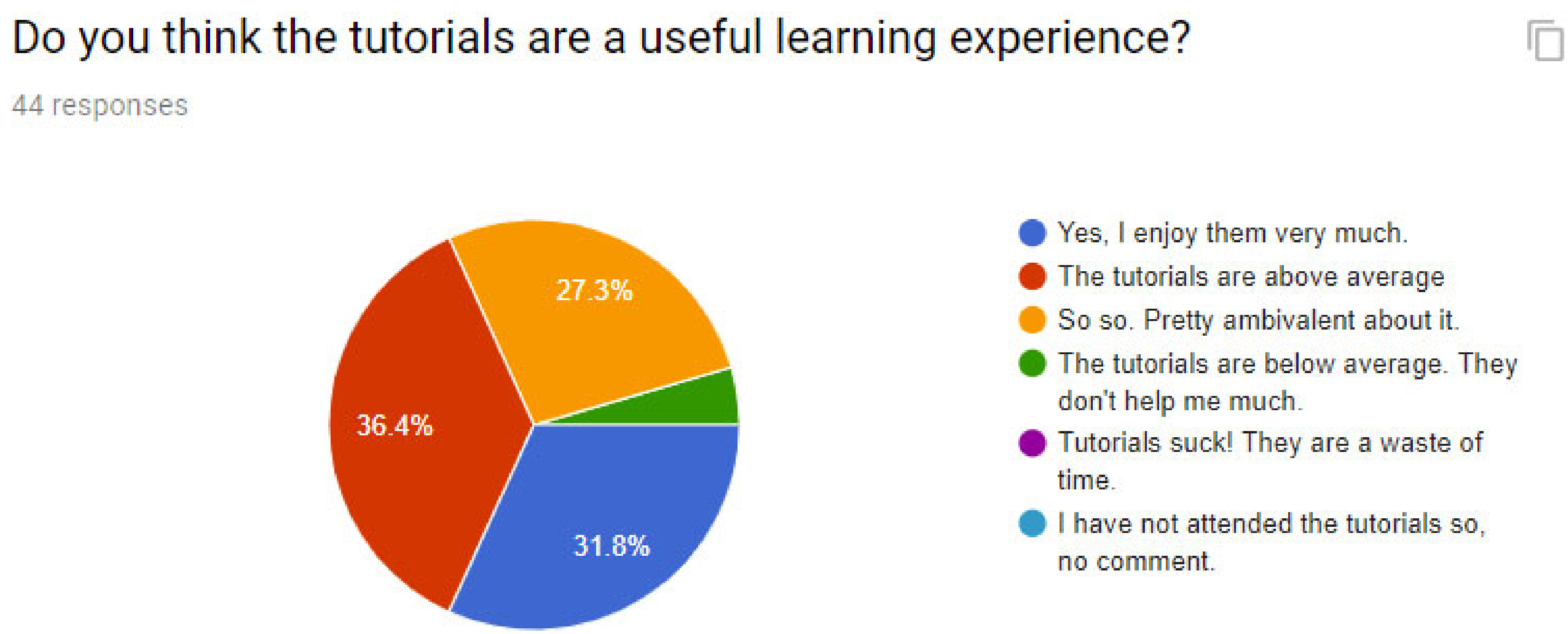
PS3 on different website

- "I prefer Kattis. Why is PS3 speed demon on a separate website :((("

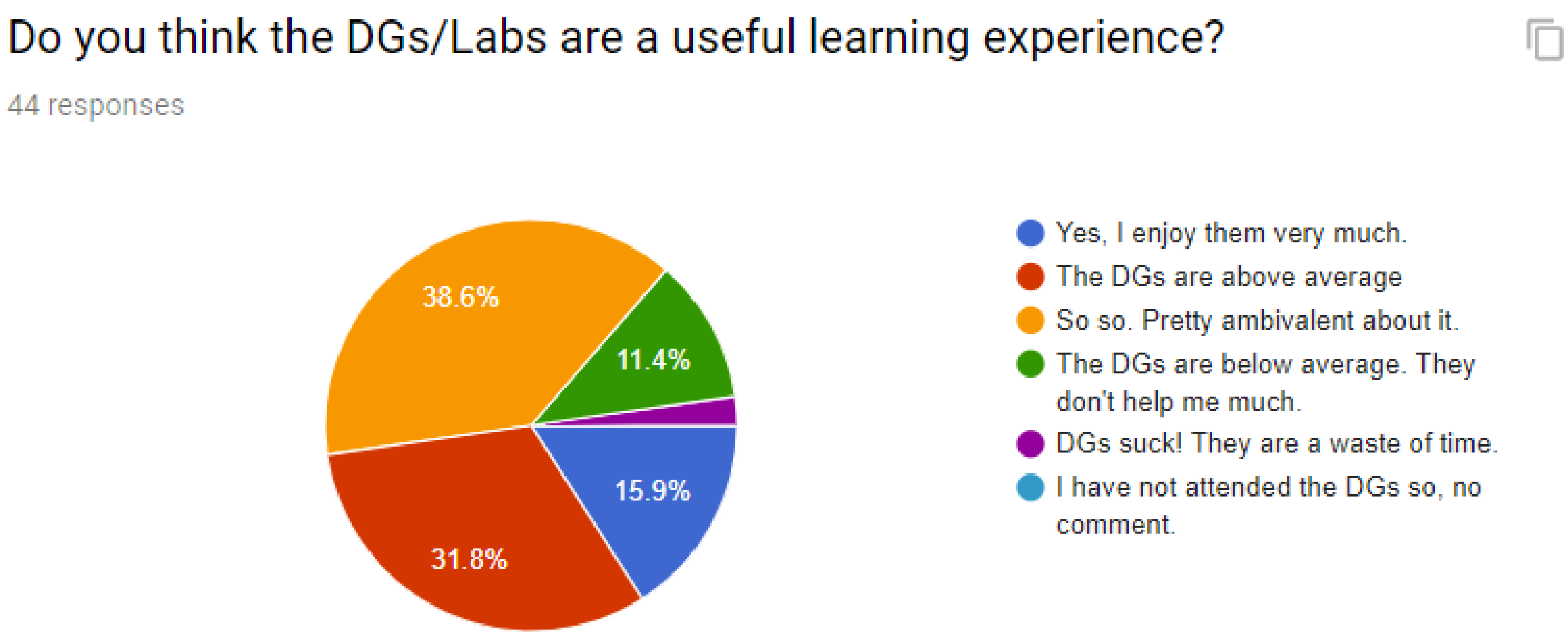
Sorry, I cannot change Kattis much. ☹
Test cases are important. Will discuss with TAs to focus some part on constructing test cases.

PS3: We had to use a different website because of large test cases.

# TUTORIALS

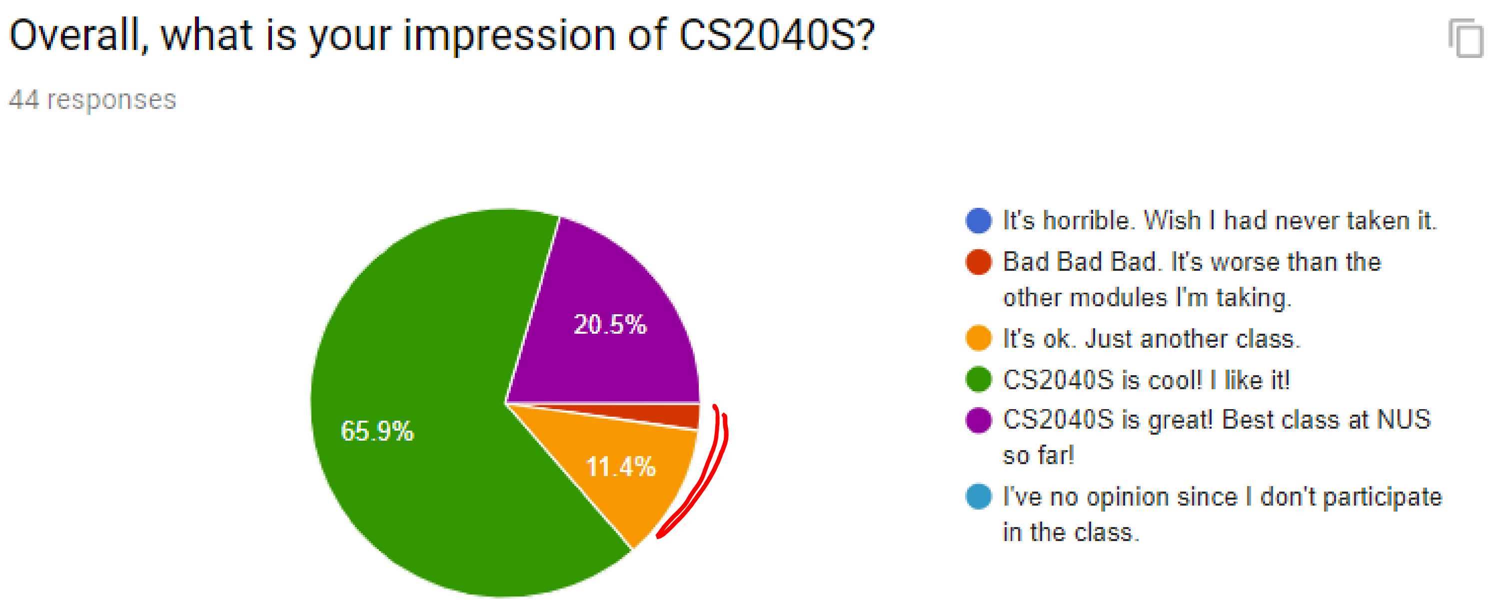

Do you think the tutorials are a useful learning experience?
44 responses
27.3%
36.4%
31.8%
Yes, I enjoy them very much.
The tutorials are above average
So so. Pretty ambivalent about it.
The tutorials are below average. They don't help me much.
Tutorials suck! They are a waste of time.
I have not attended the tutorials so, no comment.

# DGS


Do you think the DGs/Labs are a useful learning experience?
44 responses
38.6%
11.4%
15.9%
31.8%
Yes, I enjoy them very much.
The DGs are above average
So so. Pretty ambivalent about it.
The DGs are below average. They don't help me much.
DGs suck! They are a waste of time.
I have not attended the DGs so, no comment.

# OVERALL IMPRESSION

**Room to improve!**



# ADMINISTRATIVE ISSUES

**Start** 2019-10-13 17:59 CEST **Problem Set 4** **End** 2019-10-28 16:59 CET

**Time elapsed** 34:28:01 **Time remaining** 325:32:00

PS4 has been released!

Due in two weeks

Only 4 graded problems: **A, B, C, D**

The rest are practice.

TAs have also prepared a Pre-PS4 practice to help

**PS Feedback:**

- "Perhaps we could have more problems to facilitate our learning. I can't help but to feel that there are only few problems we have been exposed to…"
- "I would like additional practice!"

Poll Everywhere

https://bit.ly/2LvG9bq

# QUESTIONS?

# WHAT WE COVERED LAST WEEK

Shortest Paths in graphs (with no negative weight cycles) are **simple paths**

The **relax** method

- **Path Relation property**

**Bellman-Ford** Algorithm

# SIMPLE PATHS

***Lemma 2:*** If $G = (V, E)$ contains **no negative weight cycles**, then the shortest path $p$ from source vertex $s$ to a vertex $v$ is a **simple path.**

A **simple path** is defined as path $p = \{v_0, v_1, v_2, \ldots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\ 0 \leq i \leq (k-1)$ and there is **no** repeated vertex along this path.

This means that the shortest path can have *at most* $\underline{\quad |\boldsymbol{V}| - \mathbf{1} \quad}$ edges

# SHORTEST PATHS

```
relax(int u, int v){
                 5          3     +   1  = 4
      if (dist[v] > dist[u] + weight(u,v))
         dist[v] = dist[u] + weight(u,v);
}
```

relax(A, B)

Maintain estimate for each distance:

relax(S, B)

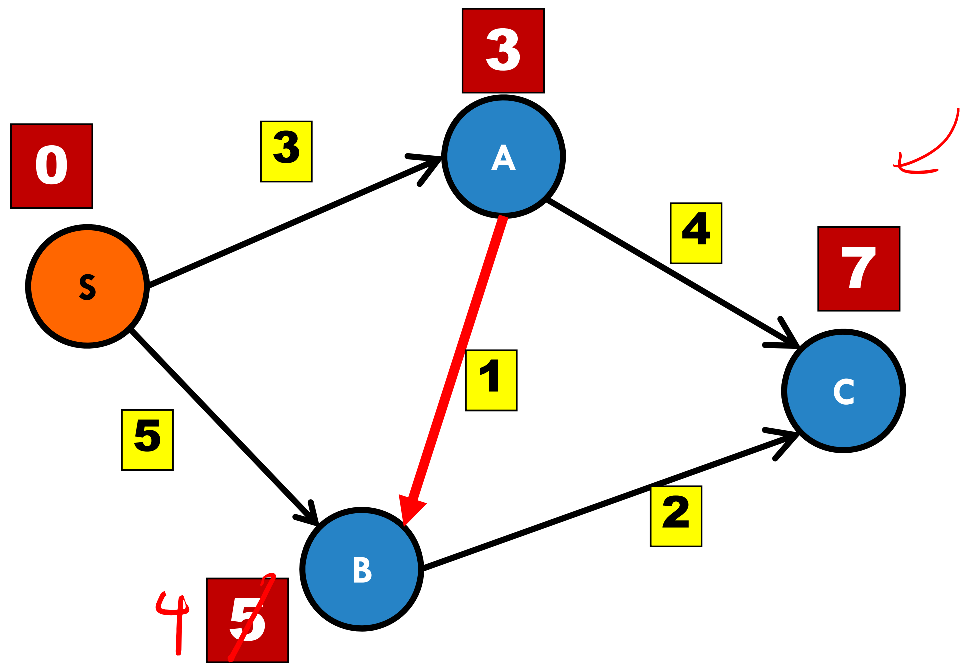
**The idea:**

relax($w$,$v$):

- Test if the best way to get from $s \to v$ is to go from $s \to w$, then $w \to v$.

If yes:

- Update dist[v]
- Update edgeTo[v]



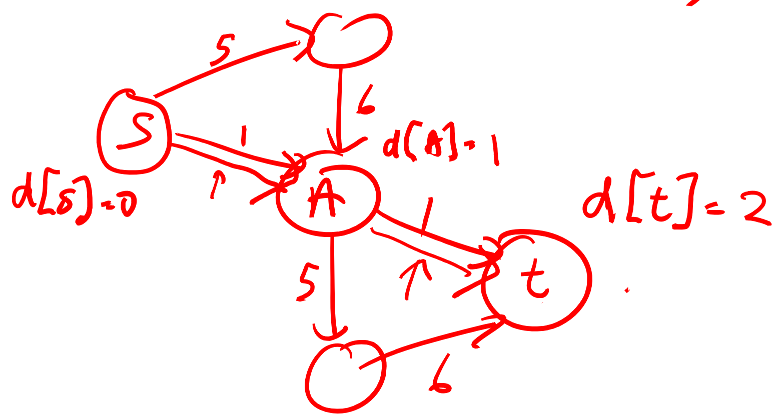
# PATH RELAXATION PROPERTY

**Lemma 5.** If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to $v_k$ and we relax the edges of $p$ in the order

$$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$$

(s, a, t)

Then $d[v_k] = \delta[v_k]$.



# PATH RELAXATION PROPERTY

**Lemma 5.** If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to $v_k$ and we relax the edges of $p$ in the order

$$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$$

Then $d[v_k] = \delta[v_k]$.

This property holds ***regardless of any other relaxation steps that occur*** (even **intermixed**)

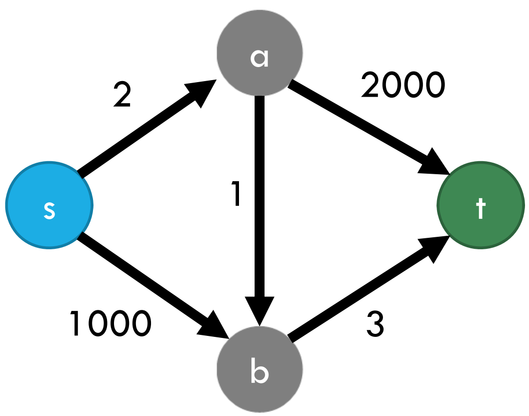
- E.g., $(v_0, v_1), (v_i, v_j), (v_1, v_2), \dots, (v_{k-1}, v_k)$ will *still* result in $d[v_k] = \delta[v_k]$.

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**If we knew the shortest path,** path relaxation tells us how to compute the distance.

For the graph,

$d[s] = 0$

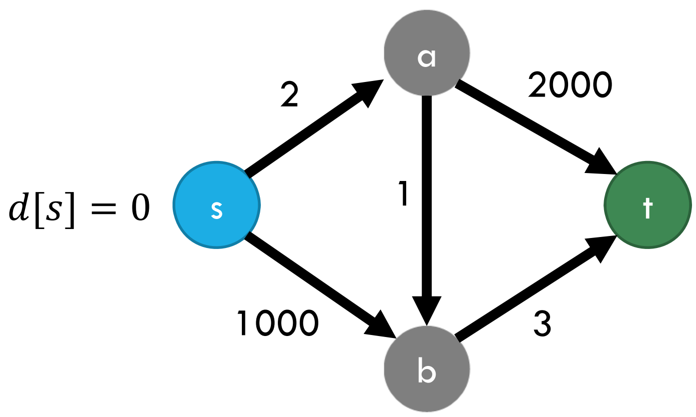



The shortest path is: ____________

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**If we knew the shortest path,** path relaxation tells us how to compute the distance.

For the graph,




The shortest path is: s,a,b,t

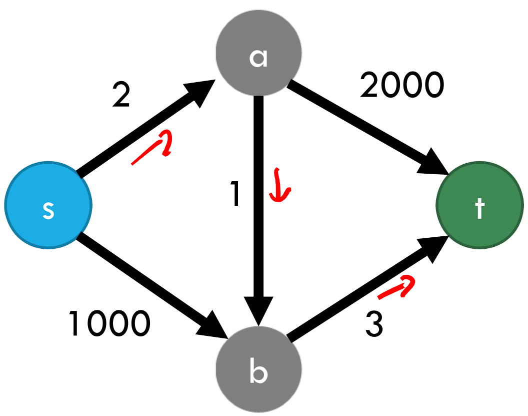
How to compute the distance from s →t?

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**If we knew the shortest path,** path relaxation tells us how to compute the distance.

For the graph,

$d[s] = 0$




The shortest path is: s,a,b,t

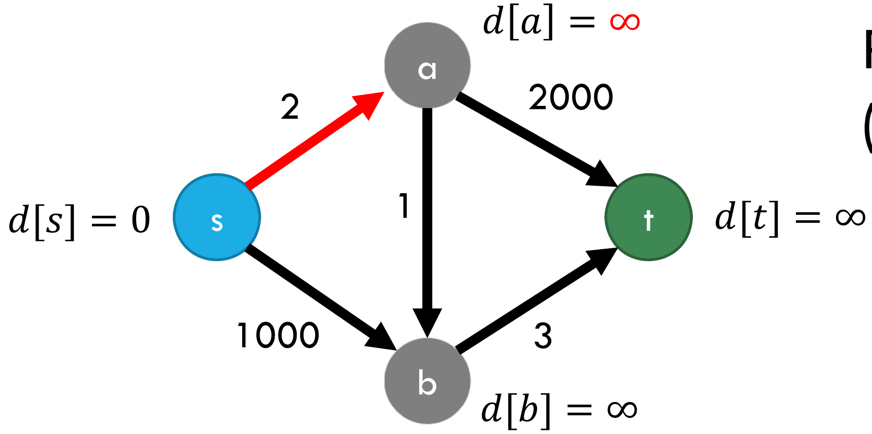
Relax in order:

(s,a), (a,b), (b,t)

# PATH RELAXATION PROPERTY → BELLMAN-FORD

If we knew the shortest path, path relaxation tells us how to compute the distance.

For the graph




The shortest path is: s,a,b,t

Relax in order:

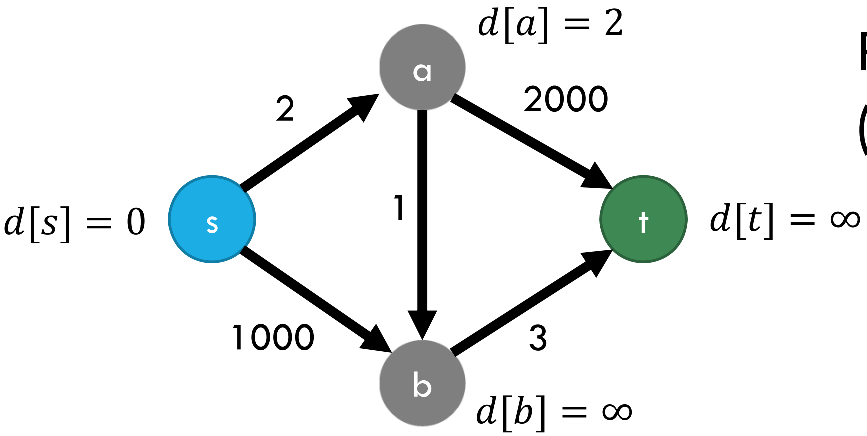
(s,a), (a,b), (b,t)

| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (s,a) | (a,b) | (b,t) |
| $d[t]$ | | | |

# PATH RELAXATION PROPERTY → BELLMAN-FORD

If we knew the shortest path, path relaxation tells us how to compute the distance.

For the graph



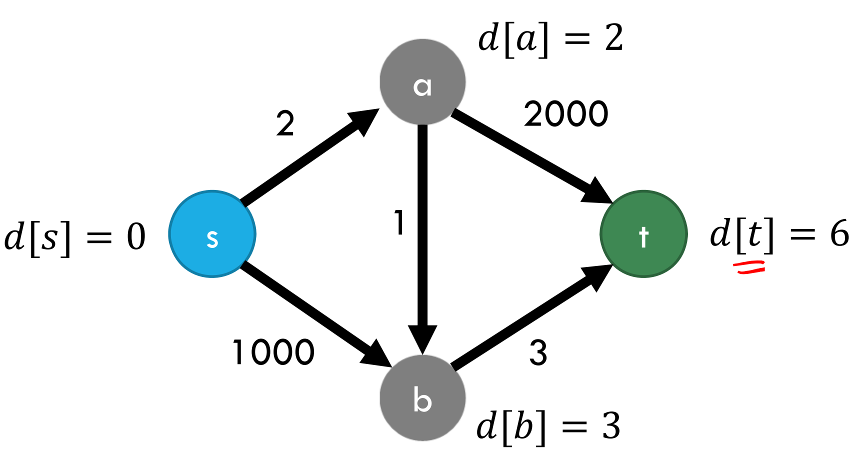

The shortest path is: s,a,b,t
Relax in order:
(s,a), (a,b), (b,t)

| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (s,a) | (a,b) | (b,t) |
| $d[t]$ | $\infty$ | | |

# PATH RELAXATION PROPERTY → BELLMAN-FORD

If we knew the shortest path, path relaxation tells us how to compute the distance.

For the graph




The shortest path is: s,a,b,t

Relax in order:

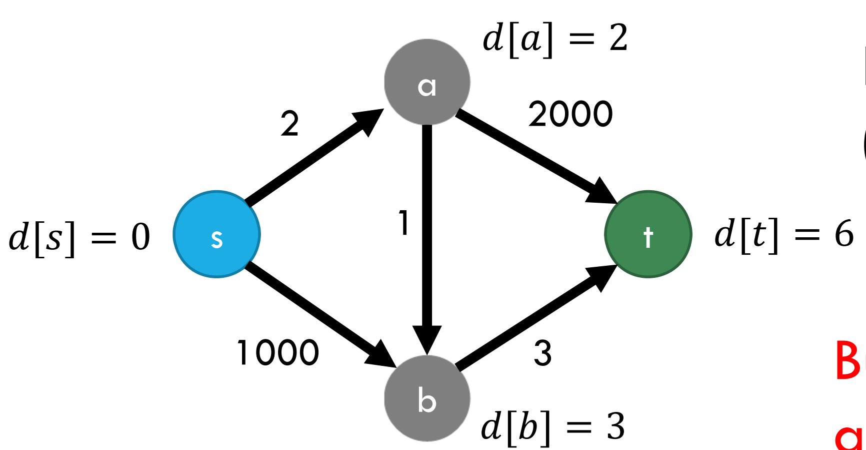
(s,a), (a,b), (b,t)

| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (s,a) | (a,b) | (b,t) |
| $d[t]$ | $\infty$ | $\infty$ | 6 |

# PATH RELAXATION PROPERTY → BELLMAN-FORD

If we knew the shortest path, path relaxation tells us how to compute the distance.

For the graph




The shortest path is: s,a,b,t
Relax in order:
(s,a), (a,b), (b,t)
Distance d[s,t] is 6

But we don't know the shortest path in general. **What can we do?**

# PATH RELAXATION PROPERTY

**Lemma 5.** If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to $v_k$ and we relax the edges of $p$ in the order

$$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$$

Then $d[v_k] = \delta[v_k]$.

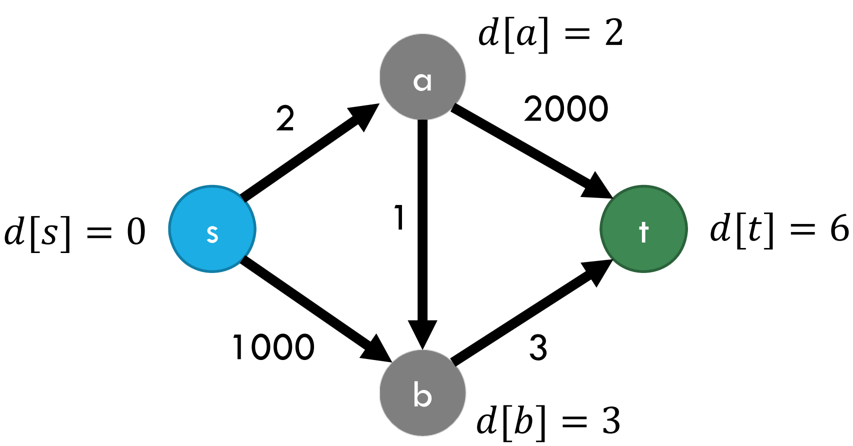
This property holds ***regardless of any other relaxation steps that occur*** (even **intermixed**)

- E.g., $(v_0, v_1), (v_i, v_j), (v_1, v_2), \dots, (v_{k-1}, v_k)$ will *still* result in $d[v_k] = \delta[v_k]$.

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**If we knew the shortest path,** path relaxation tells us how to compute the distance.

For the graph




The shortest path is: s,a,b,t

Relax in order:

(s,a), (a,b), (b,t)

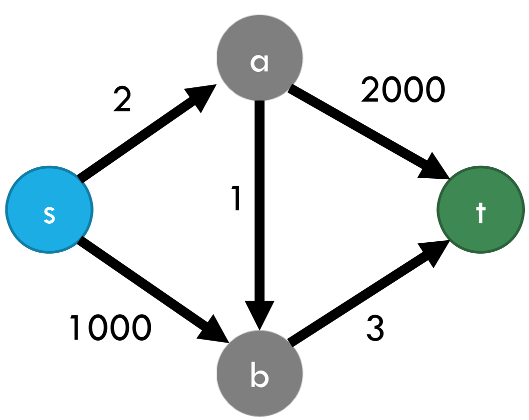
| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (s,a) | (a,b) | (b,t) |
| $d[t]$ | $\infty$ | $\infty$ | 6 |

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**Idea:** At each iteration, relax all the edges!

For the graph

$d[s] = 0$




| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (b,t) | (b,t) | (b,t) |
| | (a,b) | (a,b) | (a,b) |
| | (s,a) | (s,a) | (s,a) |
| | (a,t) | (a,t) | (a,t) |
| | (s,b) | (s,b) | (s,b) |
| $d[t]$ | ? | ? | 6 |

**How many iterations will definitely "cover" the correct order?**

# SIMPLE PATHS

***Lemma 2:*** If $G = (V, E)$ contains **no negative weight cycles**, then the shortest path $p$ from source vertex $s$ to a vertex $v$ is a **simple path.**

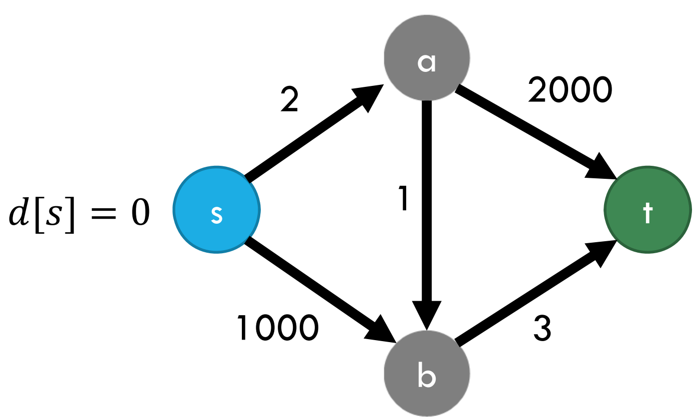
A **simple path** is defined as path $p = \{v_0, v_1, v_2, \dots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\ 0 \leq i \leq (k-1)$ and there is **<u>no</u>** repeated vertex along this path.

This means that the shortest path can have *at most* $\underline{\ \boldsymbol{|V| - 1}\ }$ edges

# PATH RELAXATION PROPERTY → BELLMAN-FORD

**Idea:** At each iteration, relax all the edges! Repeat for $|V| - 1$ iterations.

For the graph




$|V| - 1$ iterations will "cover" all possible shortest paths ***from s to any other node.***

| | Iter 1 | Iter 2 | Iter 3 |
|---|---|---|---|
| Edge | (b,t) | (b,t) | (b,t) |
| | (a,b) | (a,b) | (a,b) |
| | (s,a) | (s,a) | (s,a) |
| | (a,t) | (a,t) | (a,t) |
| | (s,b) | (s,b) | (s,b) |
| $d[t]$ | ? | ? | 6 |

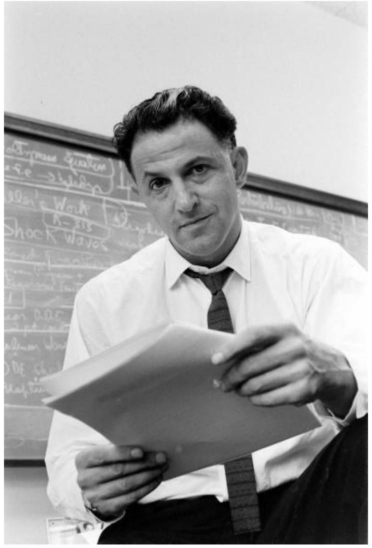
Poll Everywhere

https://bit.ly/2LvG9bq

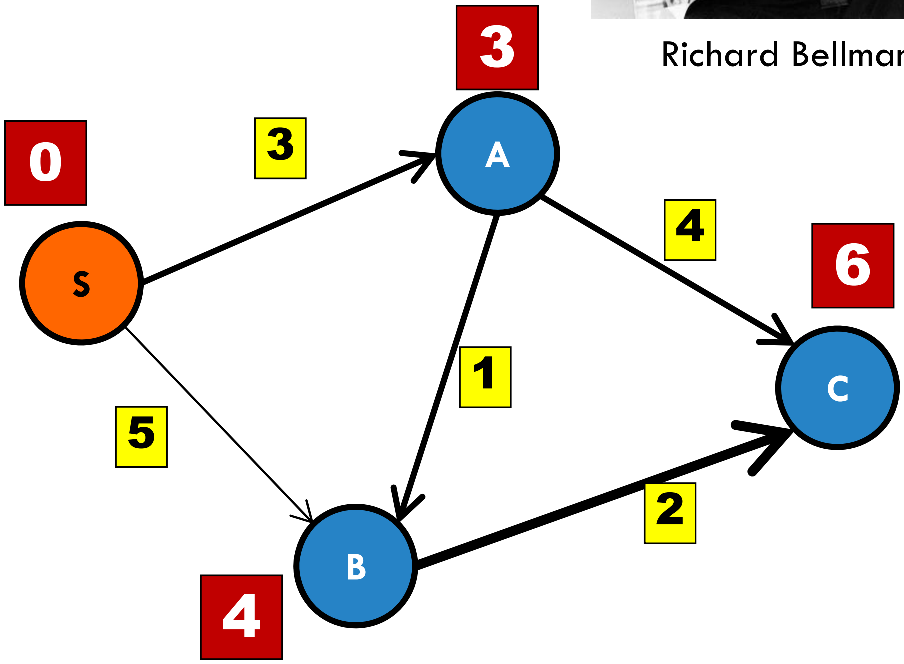
# QUESTIONS?

# BELLMAN-FORD ALGORITHM FOR SINGLE-SOURCE SHORTEST PATHS

```
n = V.length
for i = 1 to n-1
   for Edge e in Graph
       relax(e)
```



Richard Bellman



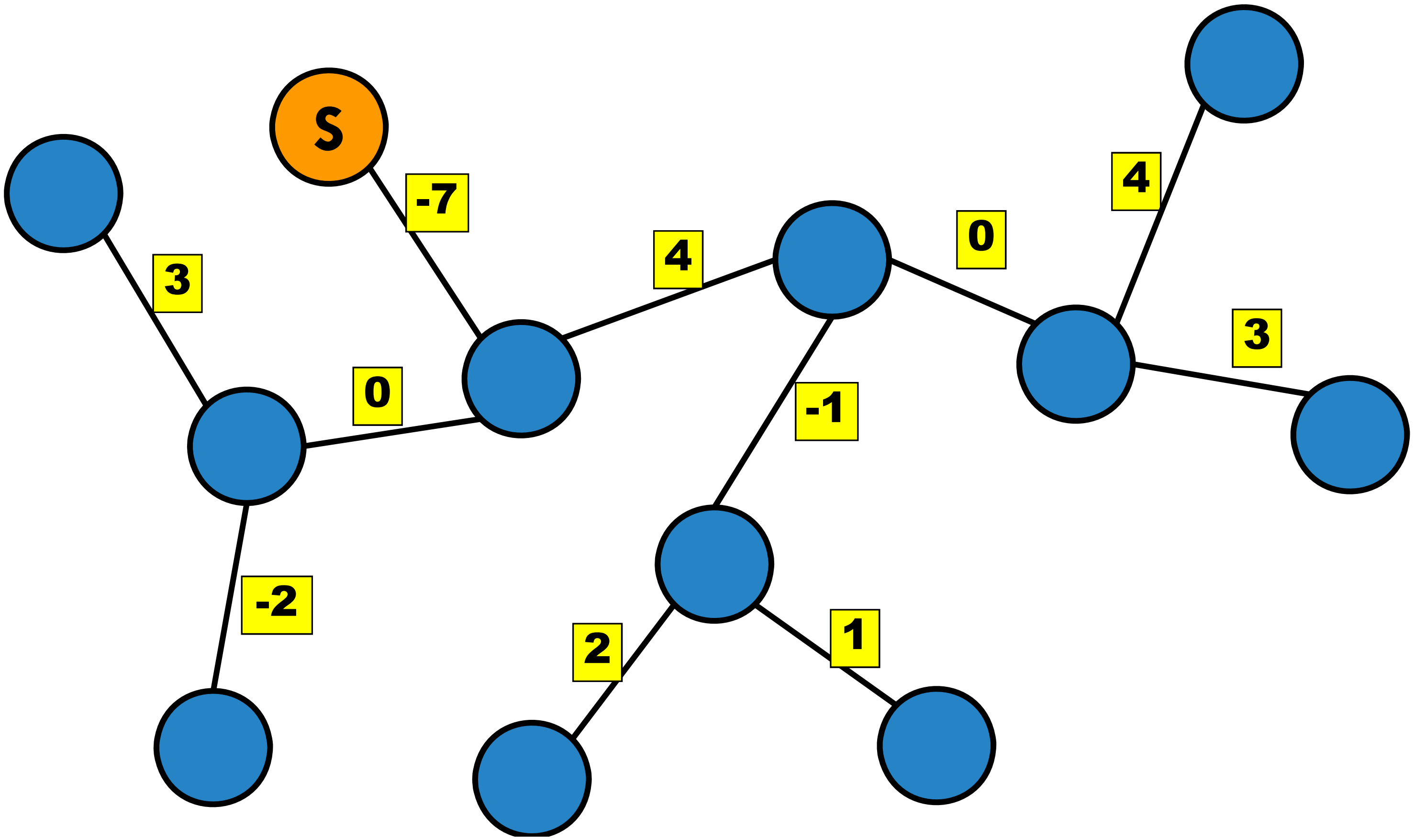

**In what order should I relax the edges?**

# TODAY: SPECIAL CASES

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V + E)$ |
| No Negative Weights | Dijkstra's Algorithm | |
| On Tree | BFS / DFS order | $O(V)$ |
| On DAG | Topological sort order | |

# UNDIRECTED WEIGHTED TREE

every node only has one parent (except the root). $O(V) = O(E)$ edges.

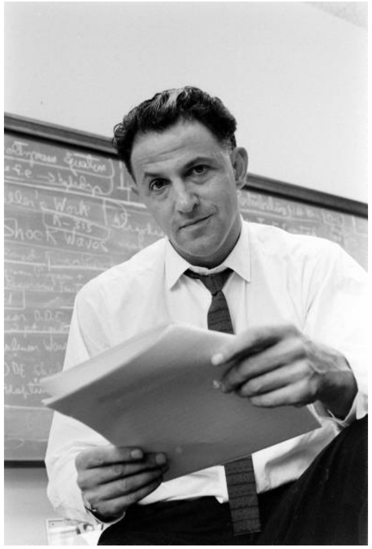


# TODAY: SPECIAL CASES

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | |
| On Tree | BFS / DFS order | $O(V)$ |
| On DAG | Topological sort order | |

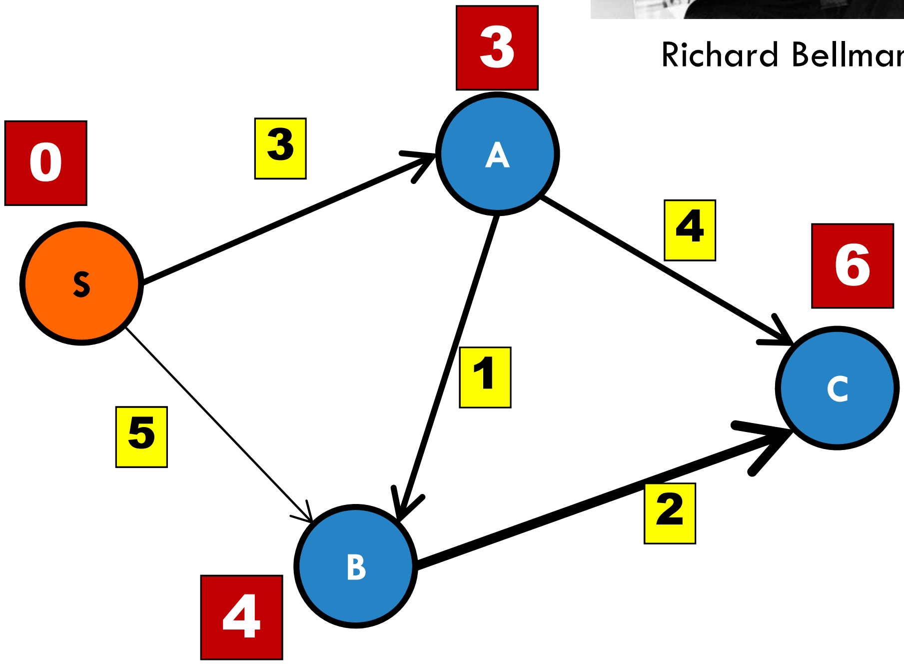
# BELLMAN-FORD ALGORITHM FOR SINGLE-SOURCE SHORTEST PATHS

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
      relax(e)
```



Richard Bellman

**How can I store the shortest path?**



# SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

Maintain estimate for each distance:

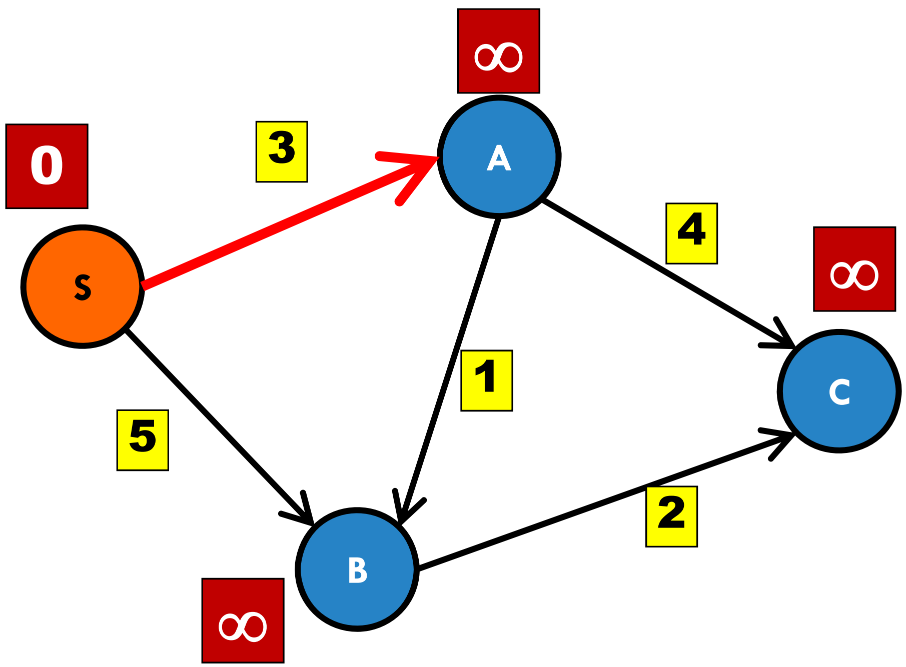
relax(S, A)

**The idea:**

relax($w$,$v$):

Test if the best way to get from $s \rightarrow v$ is to go from $s \rightarrow w$, then $w \rightarrow v$.

Update `dist`



# SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
      dist[v] = dist[u] + weight(u,v);
      edgeTo[v] = u; //update predecessor/parent
}
```

Maintain estimate for each distance:

relax(S, A)

**The idea:**

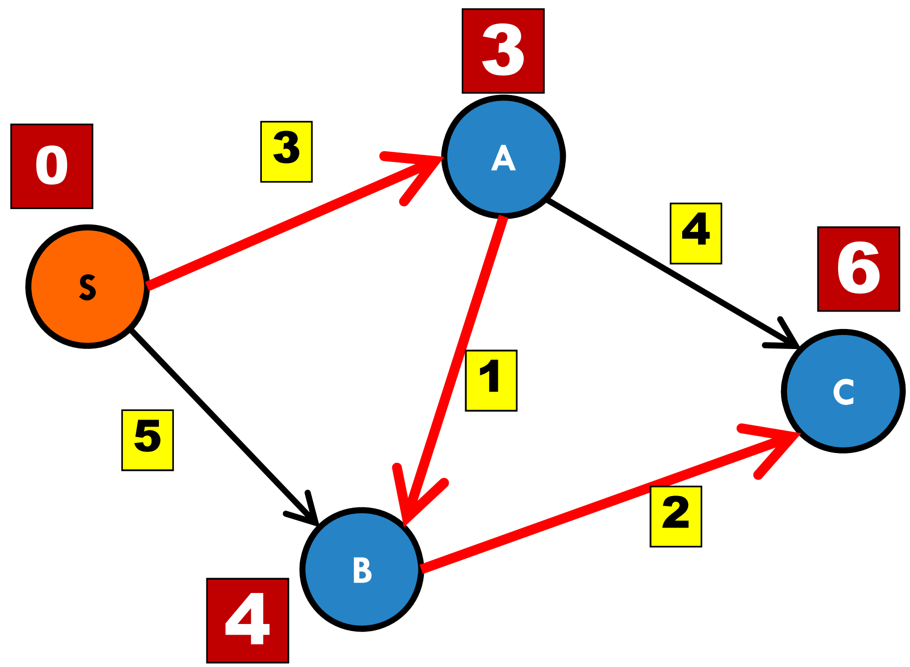
relax($w$,$v$):

- Test if the best way to get from $s \to v$ is to go from $s \to w$, then $w \to v$.

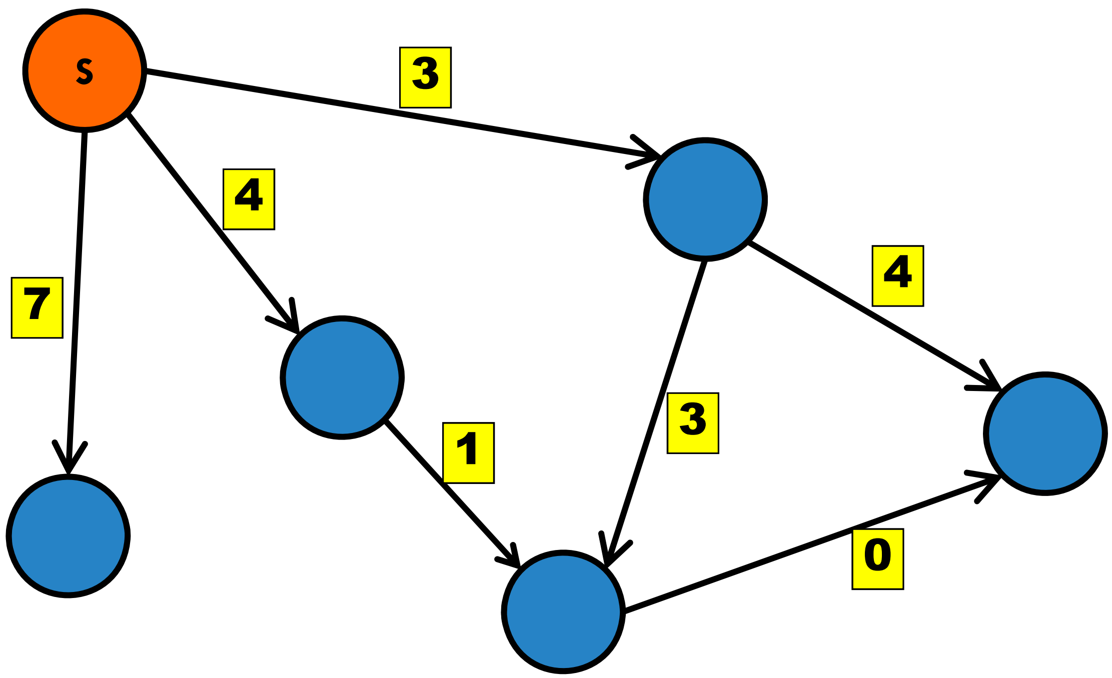
If yes:

- Update dist[v]
- Update edgeTo[v]

This creates a **predecessor subgraph**

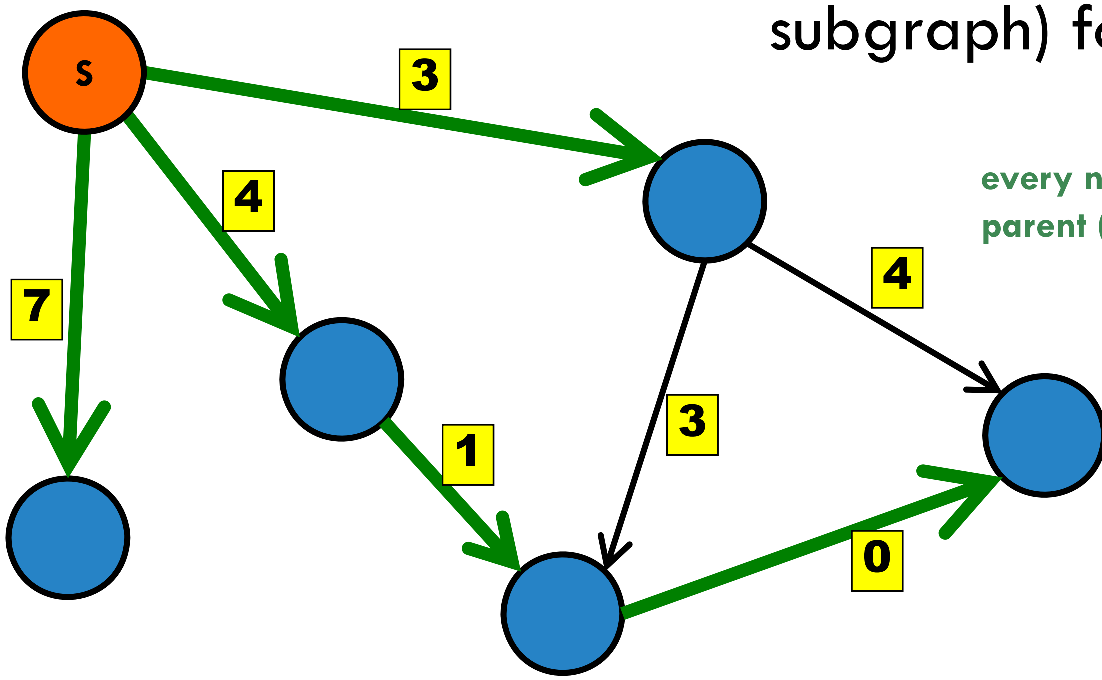


# GENERAL GRAPH: NO-NEGATIVE CYCLES



# GENERAL GRAPH: NO-NEGATIVE CYCLES

Shortest paths (predecessor-subgraph) forms a **tree**.



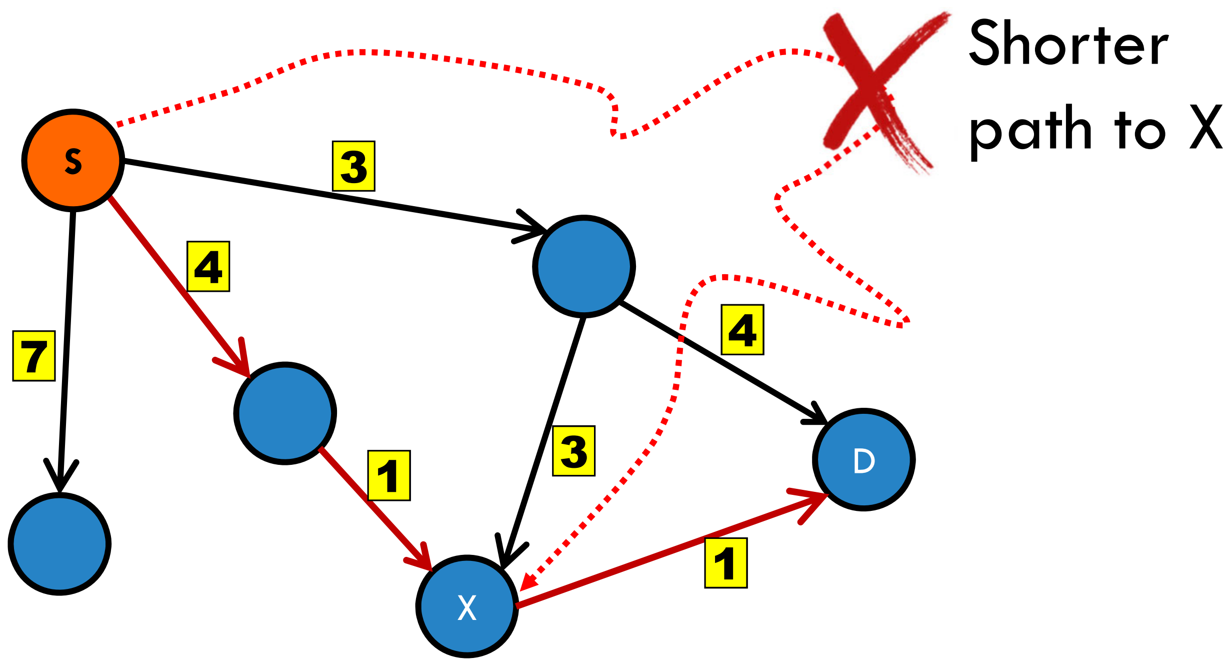

every node only has one parent (except the root).

# SUBPATHS OF SHORTEST PATHS ARE SHORTEST PATHS

**Key property:** If p is the shortest path from S to D,

and if p goes through X,

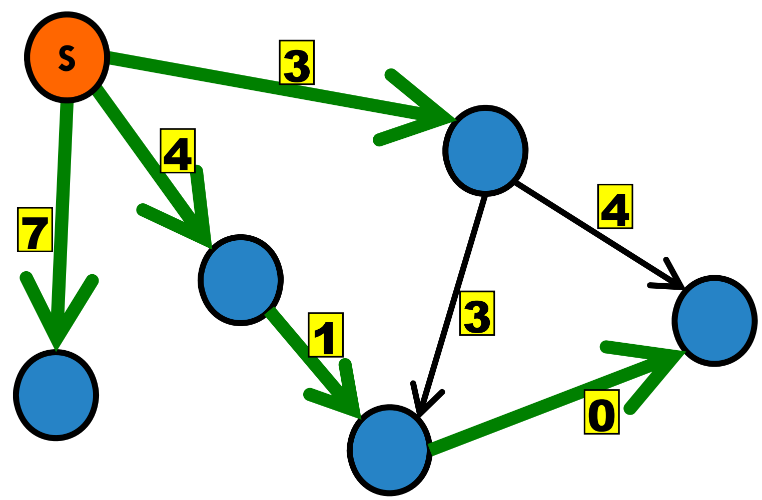
then p is also the shortest path from S to X (and from X to D).



# PREDECESSOR SUBGRAPH PROPERTY

**Lemma 7** Once $d[s, v] = \delta(s, v)$ for all $v \in V$ the **predecessor subgraph** is a **shortest-paths tree** rooted at $s$

**Read as:** Once the distance estimates are all correct (*equal to the true distance*), *then the predecessor-subgraph is tree representing all the shortest-paths from s.*



# GRAPHS WITH NON-NEGATIVE EDGES: DIJKSTRA'S ALGORITHM

**Key idea:**

Relax the edges in the "right" order.

Works on graphs with non-negative edges.

Only relax each edge **once:**

- $O(E)$ cost (for relaxation step).

# EDSGER W. DIJKSTRA

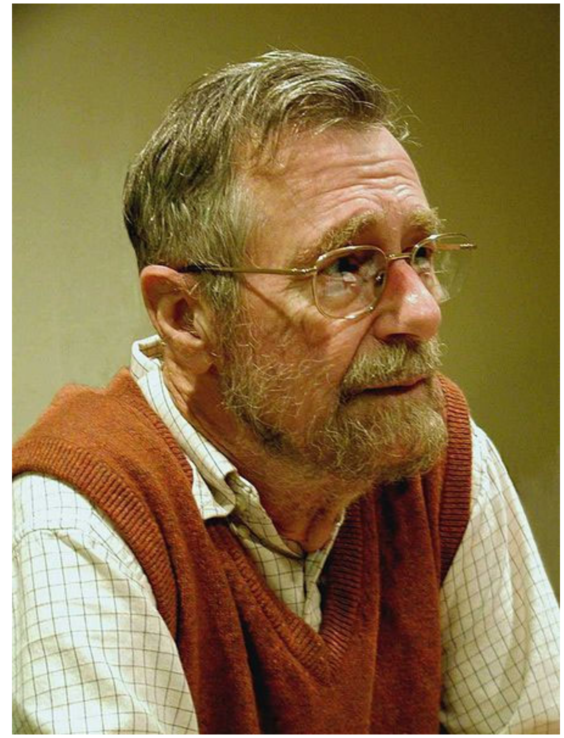
*"Computer science is no more about computers than astronomy is about telescopes."*

*"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."*

*"There should be no such thing as boring mathematics."*

*"Elegance is not a dispensable luxury but a factor that decides between success and failure."*

*"Simplicity is prerequisite for reliability."*



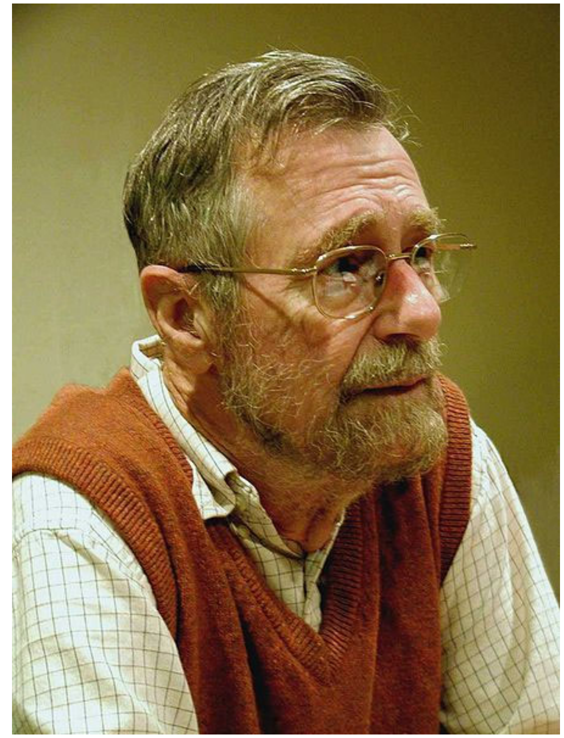
1930-2002

# EDSGER W. DIJKSTRA

*"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."*

*"The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."*

*"Object-oriented programming is an exceptionally bad idea which could only have originated in California."*



1930-2002

# EDSGER W. DIJKSTRA

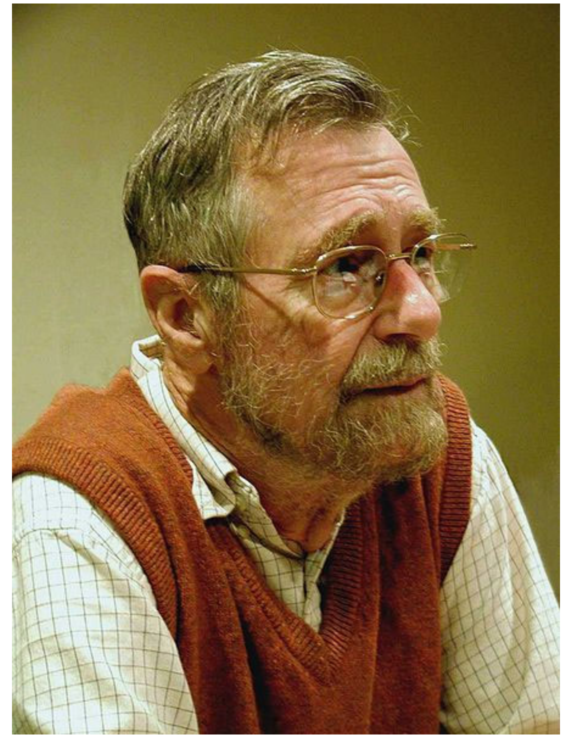
**From Wikipedia:**

His approach to teaching was unconventional …

He invited the students to suggest ideas, which he then explored, or refused to explore because they violated some of his tenets.

He conducted his final examinations orally, over a whole week.

Each student was examined in Dijkstra's office or home, and an exam lasted several hours.



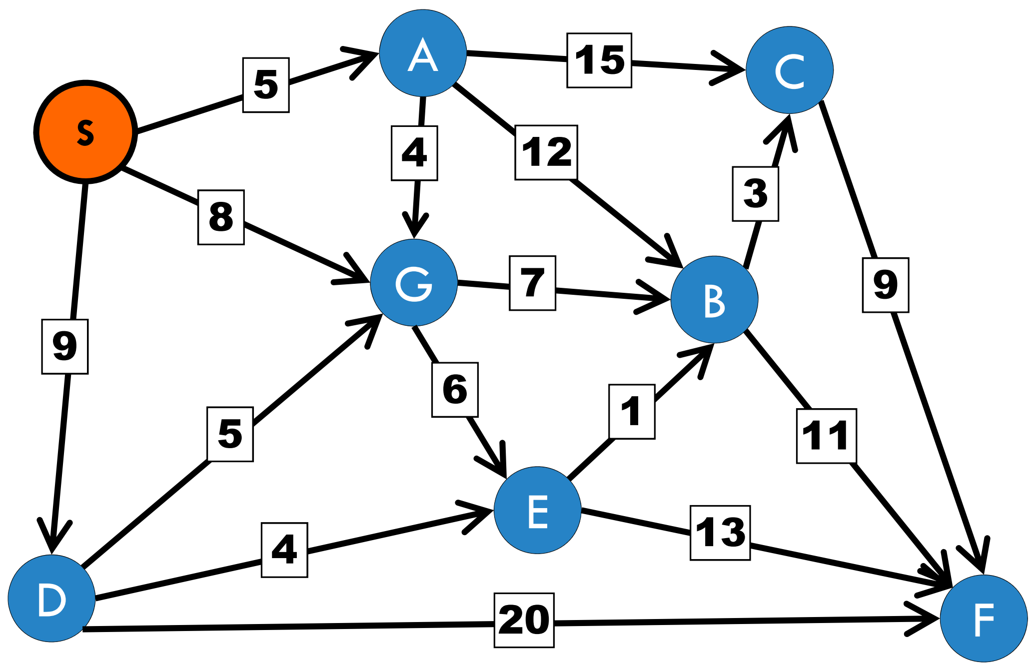
1930-2002

# DIJKSTRA'S ALGORITHM

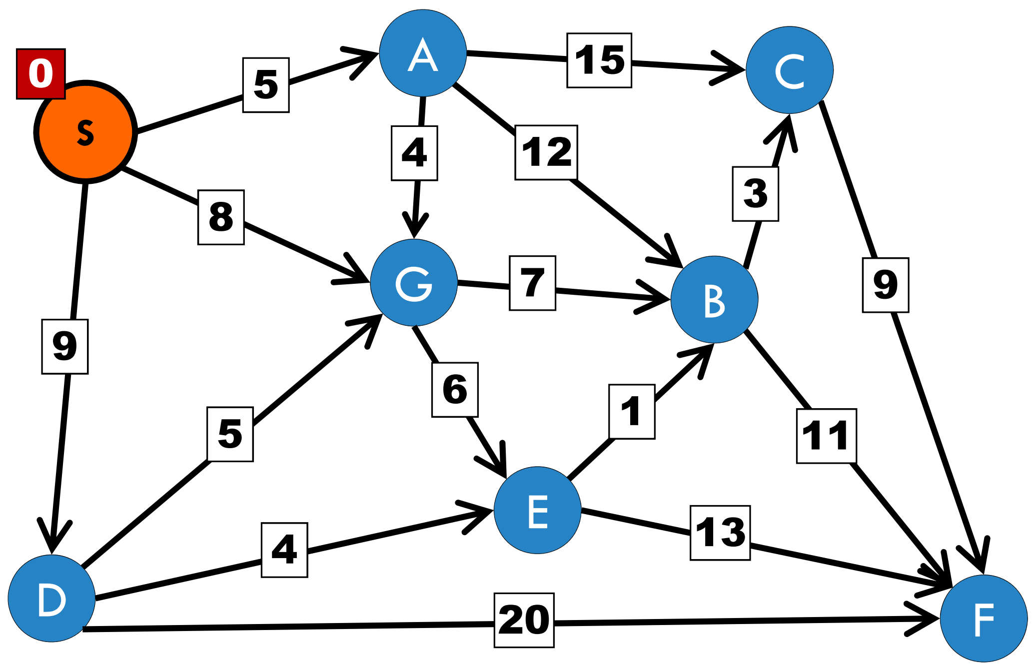
**Basic idea:**

- Maintain distance **estimate** for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Consider vertex with minimum **estimate**.
  - Add vertex to **shortest-path-tree**.
  - Relax all outgoing edges.

# DIJKSTRA'S ALGORITHM



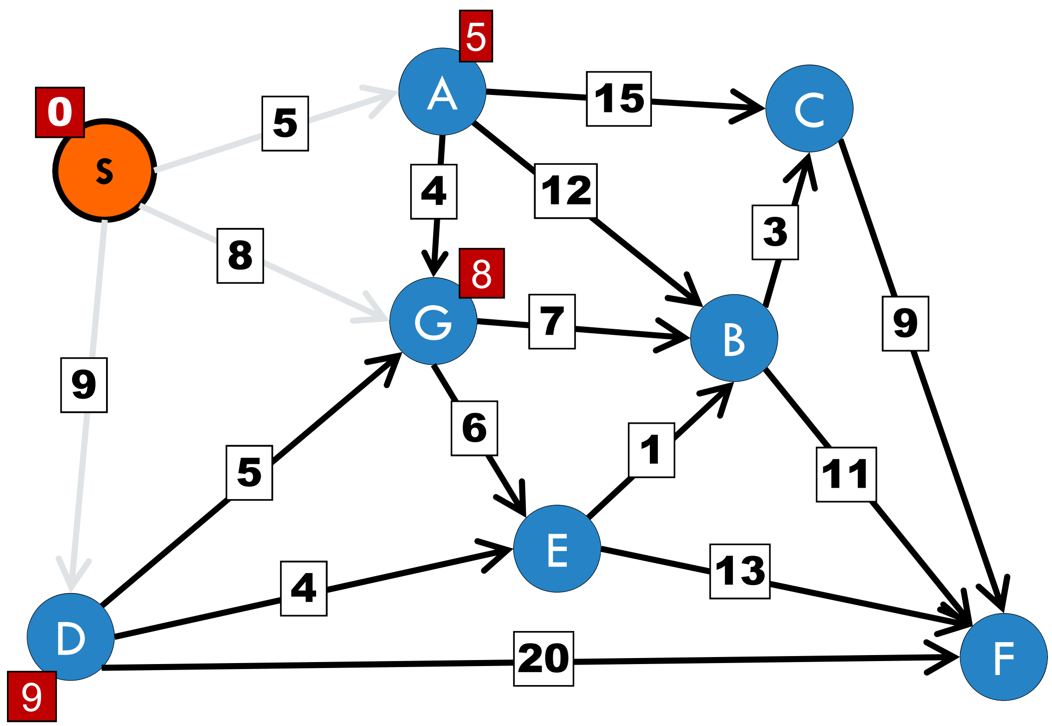
# DIJKSTRA'S ALGORITHM




| Vertex | Dist. |
|---|---|
| S | 0 |
| | |
| | |
| | |
| | |
| | |
| | |

Step 1: Add source

# DIJKSTRA'S ALGORITHM

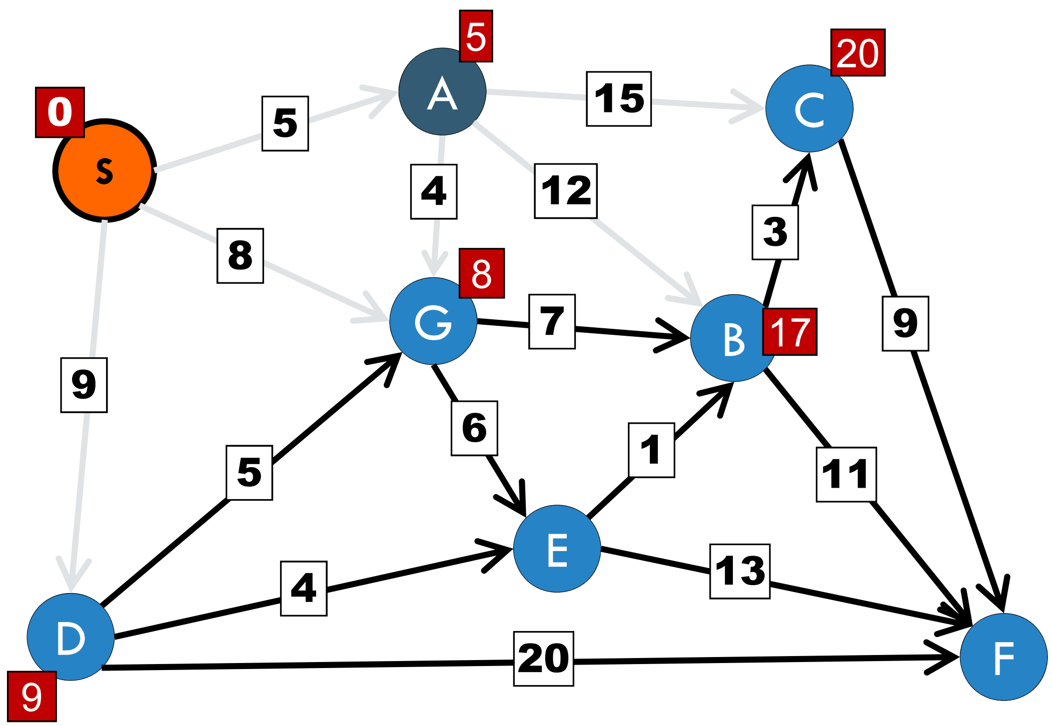



| Vertex | Dist. |
|---|---|
| A | 5 |
| G | 8 |
| D | 9 |
| | |
| | |
| | |
| | |

Step 1: Add source

Step 2: Remove S and relax.

# DIJKSTRA'S ALGORITHM



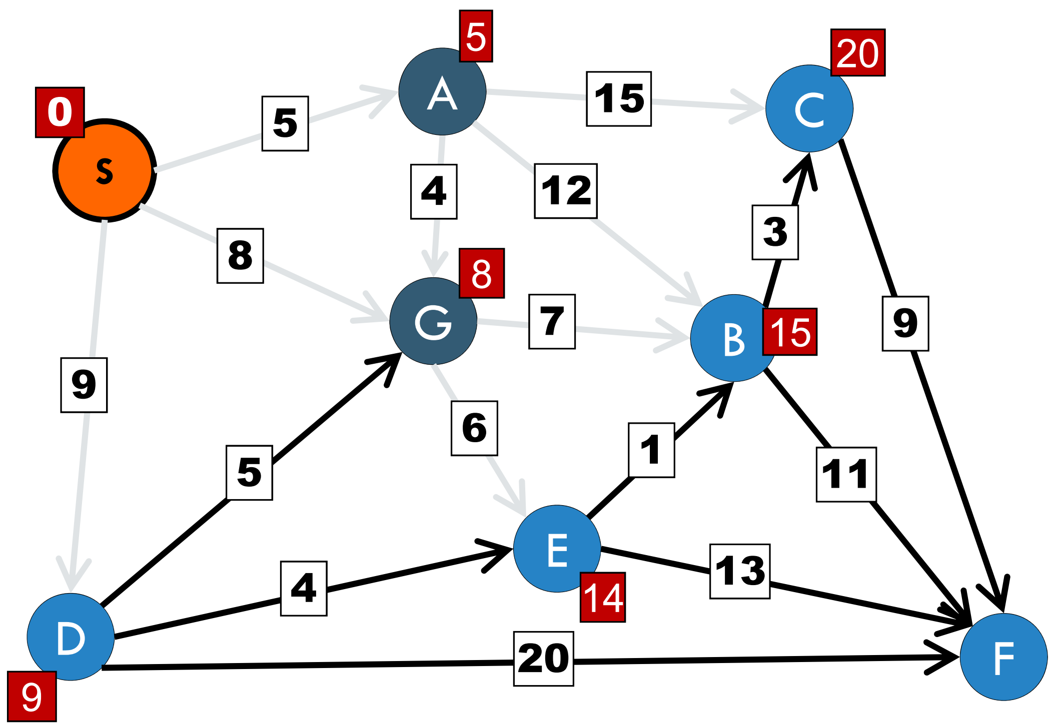

| Vertex | Dist. |
|---|---|
| G | 8 |
| D | 9 |
| B | 17 |
| C | 20 |
| | |
| | |

Step 1: Add source

Step 2: Remove S and relax.

Step 3: Remove A and relax.

# DIJKSTRA'S ALGORITHM




| Vertex | Dist. |
|---|---|
| D | 9 |
| E | 14 |
| B | 15 |
| C | 20 |
| | |
| | |

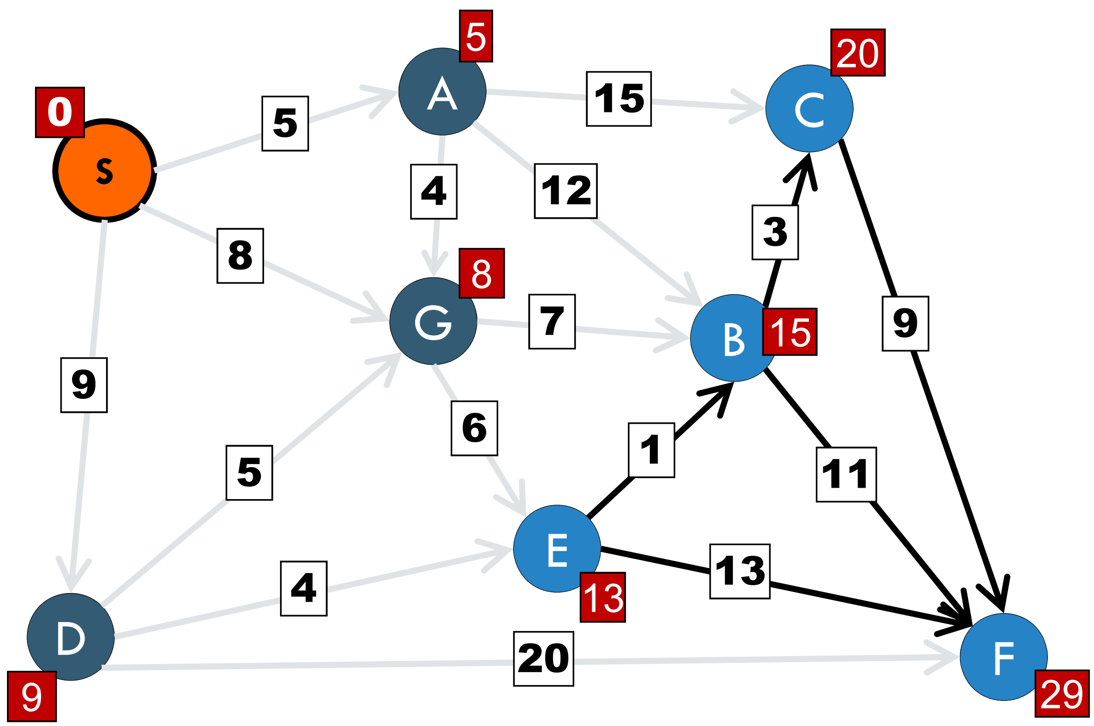
Step 1: Add source

Step 2: Remove S and relax.

Step 3: Remove A and relax.

Step 4: Remove G and relax.

# DIJKSTRA'S ALGORITHM




| Vertex | Dist. |
|---|---|
| **E** | **13** |
| B | 15 |
| C | 20 |
| **F** | **29** |
| | |

Step 1: Add source

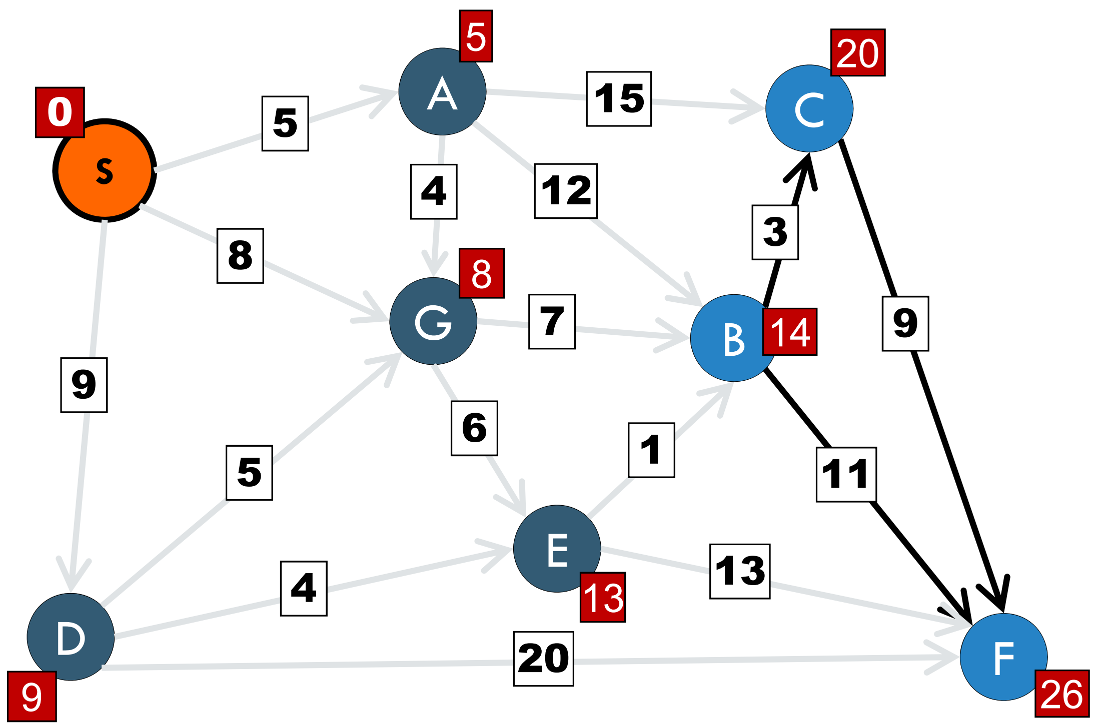
Step 2: Remove S and relax.

Step 3: Remove A and relax.

Step 4: Remove G and relax.

Step 5: Remove D and relax.

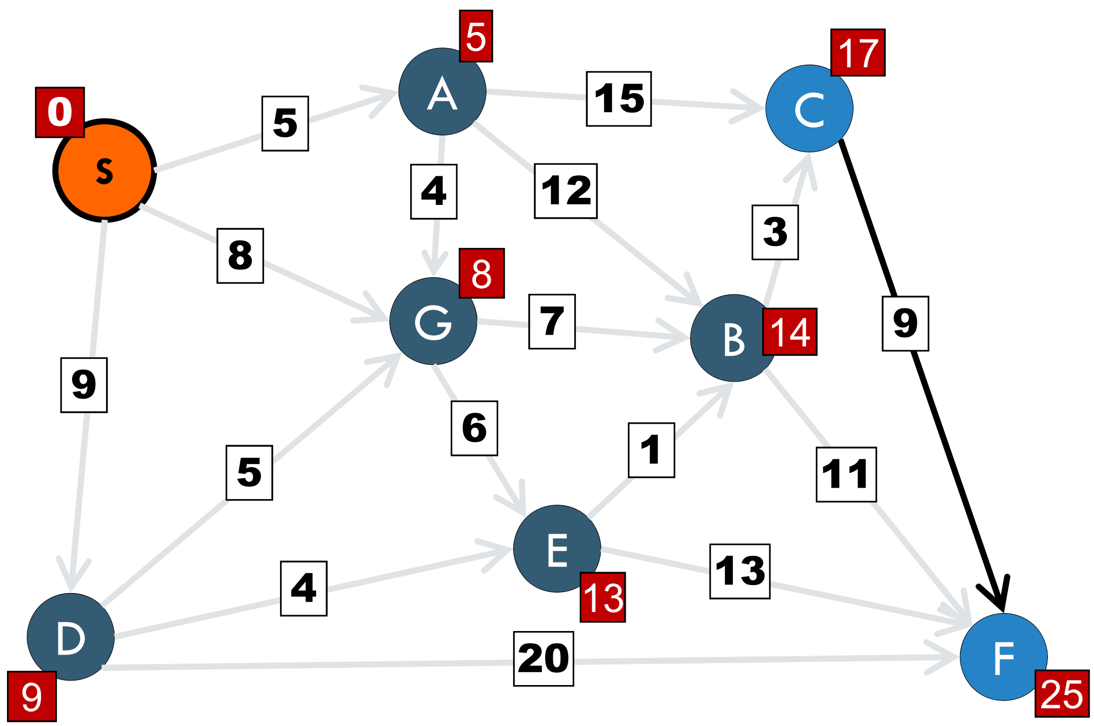
# DIJKSTRA'S ALGORITHM




| Vertex | Dist. |
|---|---|
| **B** | **14** |
| C | 20 |
| **F** | **26** |
| | |

Step 6: Remove E and relax.

# DIJKSTRA'S ALGORITHM



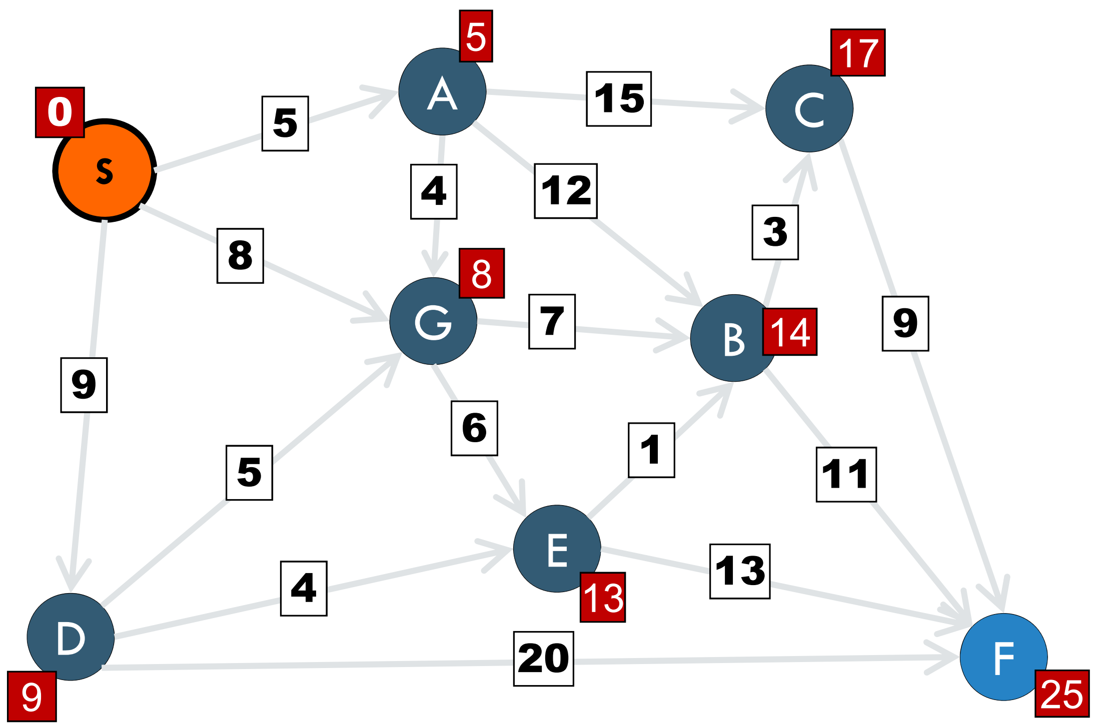

| Vertex | Dist. |
|---|---|
| C | 20 |
| F | 25 |
| | |

Step 6: Remove E and relax.

Step 7: Remove B and relax.

# DIJKSTRA'S ALGORITHM

| Vertex | Dist. |
|---|---|
| F | 25 |
| | |



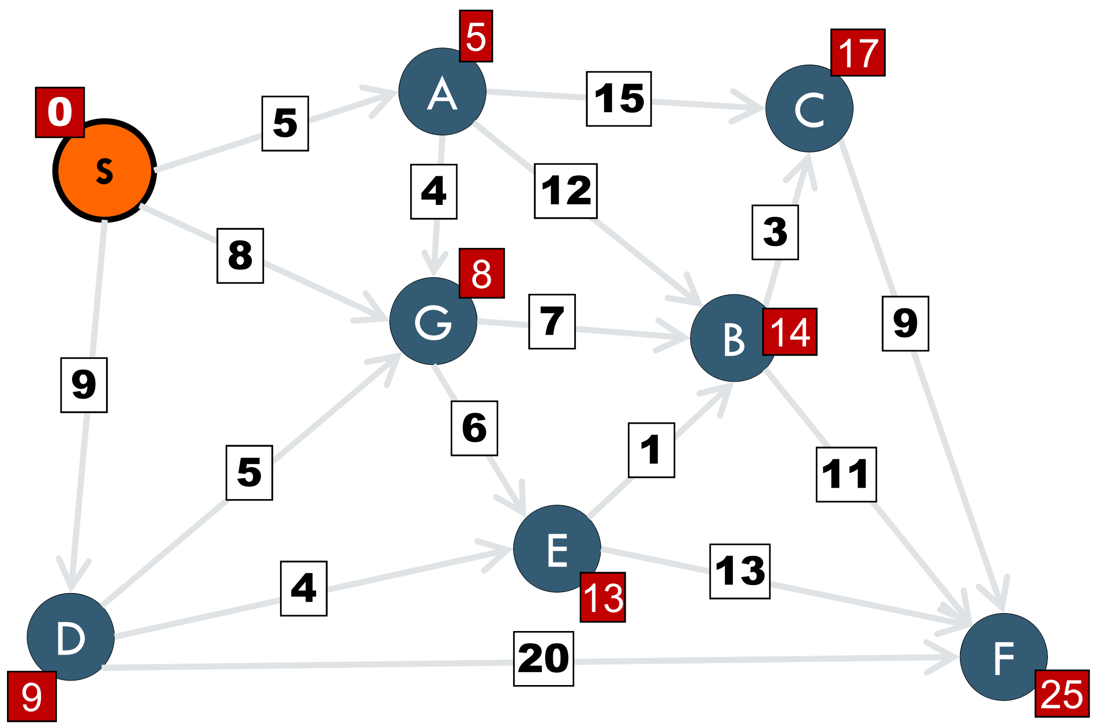

Step 6: Remove E and relax.

Step 7: Remove B and relax.

Step 8: Remove C and relax.

# DIJKSTRA'S ALGORITHM

| Vertex | Dist. |
|---|---|
| | |




Step 6: Remove E and relax.

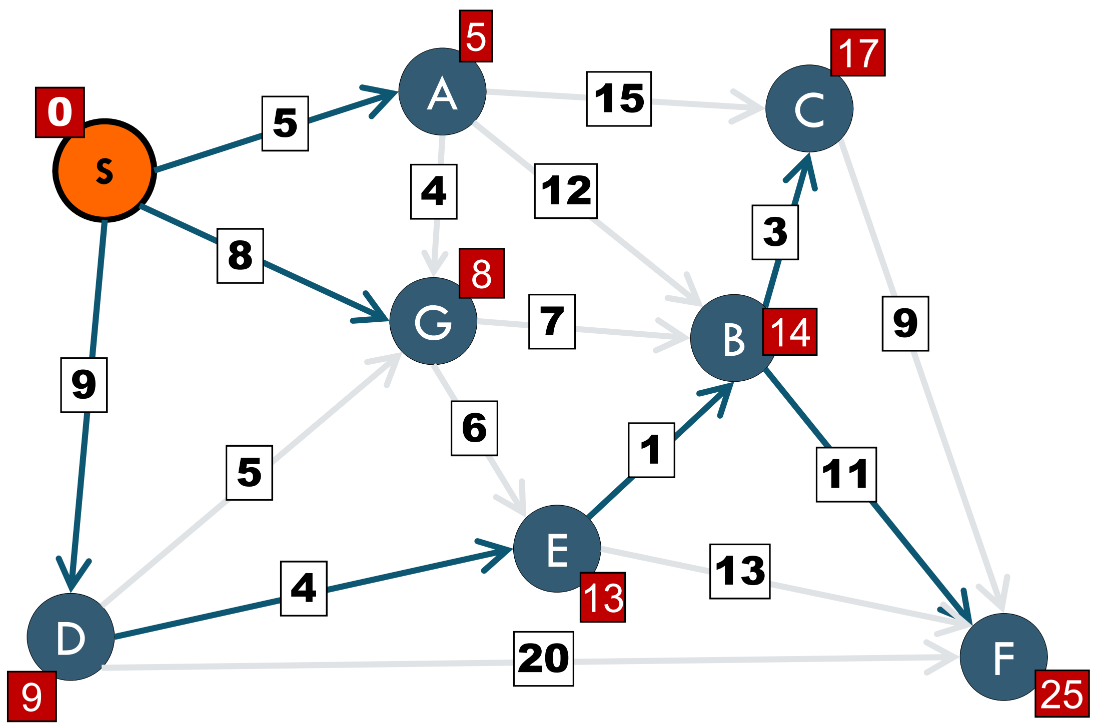
Step 7: Remove B and relax.

Step 8: Remove C and relax.

Step 9: Remove F and relax.

# DIJKSTRA'S ALGORITHM

| Vertex | Dist. |
| --- | --- |
| | |




Step 6: Remove E and relax.

Step 7: Remove B and relax.

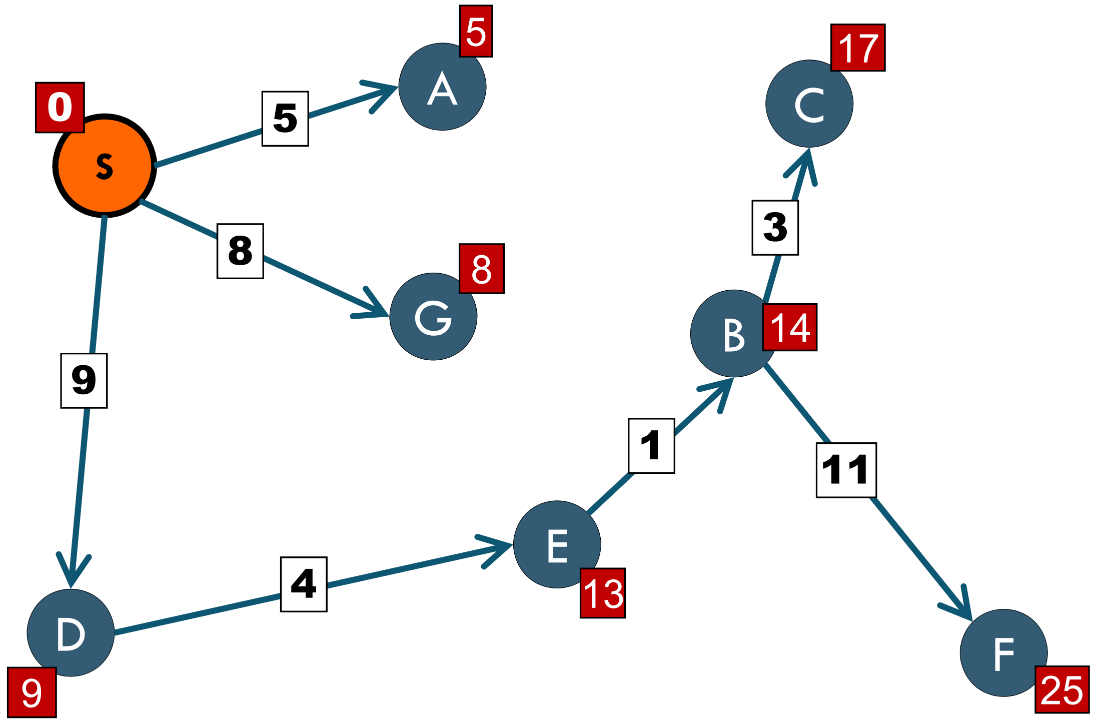
Step 8: Remove C and relax.

Step 9: Remove F and relax.

Done!

# DIJKSTRA'S ALGORITHM

| Vertex | Dist. |
| --- | --- |
| | |




Step 10: Enjoy your Shortest-Path Tree. ☺

# IMPLEMENTING DIJKSTRA'S ALG

```
Dijkstra(G, s)
    create vertex set Q
    for each v in G
        dist[v] = INFTY
        prev[v] = UNDEF
    dist[s] = 0
    while Q is not empty
        u = vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
            relax(u,v)
    return dist, prev
```

What data structure /ADT should I use for the vertex set Q?

A. Stack
B. Queue
C. List
D. Priority Queue

# IMPLEMENTING DIJKSTRA'S ALG

```
Dijkstra(G, s)
    create vertex set Q
    for each v in G
        dist[v] = INFTY
        prev[v] = UNDEF
    dist[s] = 0
    while Q is not empty
        u = vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
            relax(u,v)
    return dist, prev
```

What data structure /ADT should I use for the vertex set Q?

A. Stack
B. Queue
C. List
D. **Priority Queue**

**What operation do you do need to repeatedly perform?**

# IMPLEMENTING DIJKSTRA'S ALG

```
Dijkstra(G, s)
    create vertex set Q
    for each v in G
        dist[v] = INFTY
        prev[v] = UNDEF
    dist[s] = 0
    while Q is not empty
        u = vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
            relax(u,v)
    return dist, prev
```

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v)
        prev[v] = u
}
```

**Need to update key:**

`decreaseKey(key, priority)`

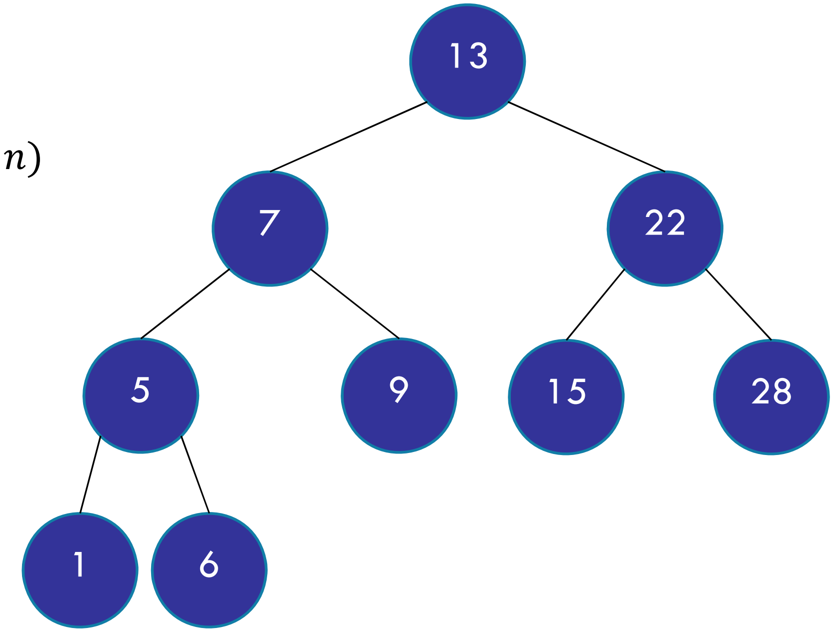
# PRIORITY QUEUE: AVL DATA STRUCTURE

AVL Tree Operations

- `deleteMin():` $O(\log n)$
- `insert(key, priority)` $O(\log n)$
- `contains(key)` $O(\log n)$
- `decreaseKey(key, priority)` $O(\log n)$

**Question:** How can we quickly find a node: dist[u] ?

Use a **hash table** to map vertices to the appropriate node in the BST.



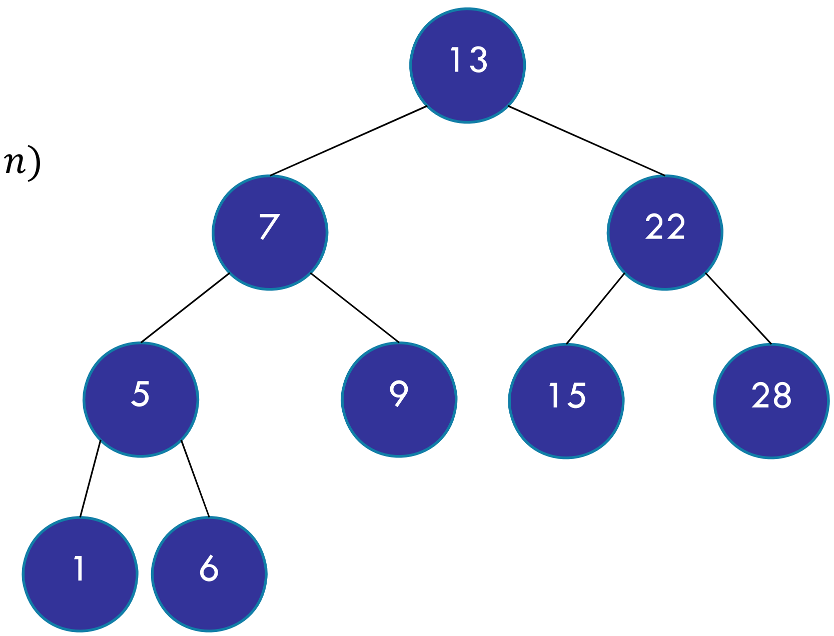
# PRIORITY QUEUE: AVL DATA STRUCTURE

AVL Tree Operations

- `deleteMin():` $O(\log n)$
- `insert(key, priority)` $O(\log n)$
- `contains(key)` $O(1)$
- `decreaseKey(key, priority)` $O(\log n)$

**Question:** How can we quickly find a node: dist[u] ?

Use a **hash table** to map vertices to the appropriate node in the BST.



# DIJKSTRA'S ALG: PERFORMANCE

| PQ Implementation | insert | deleteMin | decreaseKey | Total |
|---|---|---|---|---|
| Array | 1 | $V$ | 1 | $O(V^2)$ |
| AVL Tree | $\log V$ | $\log V$ | $\log V$ | $O((V+E) log\ V)$ |
| d-way Heap | $d\log_d V$ | $d\log_d V$ | $\log_d V$ | $O(E \log_{\frac{E}{V}} V)$ |
| Fibonacci Heap | 1 | $\log V$ | 1 | $O(E + V\ log\ V)$ |

# DIJKSTRA'S ALGORITHM: ANALYSIS

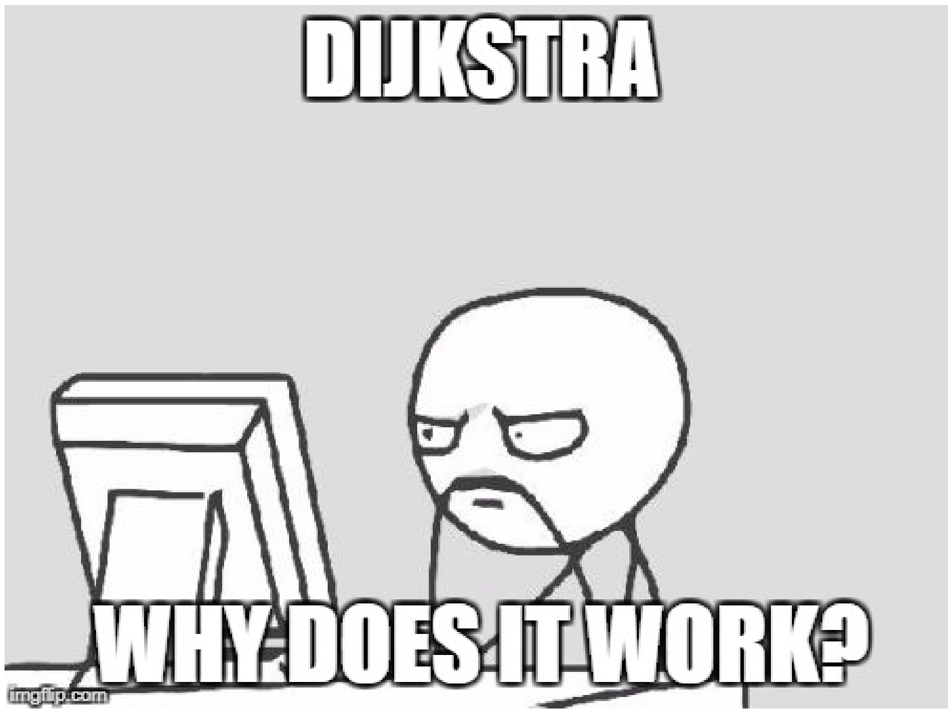
For directed graphs,
$\sum_v \deg^+(v) = |E|$
Where deg+ indicates out-degree

```
Dijkstra(G, s)
    create vertex set Q
    for each v in G
        dist[v] = INFTY
        prev[v] = UNDEF
    dist[s] = 0
    while Q is not empty
        u = vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
            relax(u,v)
    return dist, prev
```

**Analysis:**

- **insert / deleteMin:** $|V|$ times
  - Each node is added to the priority queue **once.**
- **relax / decreaseKey:** $|E|$ times
  - Each edge is relaxed once.
- Priority queue operations: $O(\log V)$
- **Total:** $O((V+E)\log V)$

# DIJKSTRA'S ALGORITHM



# PROOF IS BY … INDUCTION

Every “finished” (dequeued) vertex has a correct estimate.

- Initially: only “finished” vertex is start.

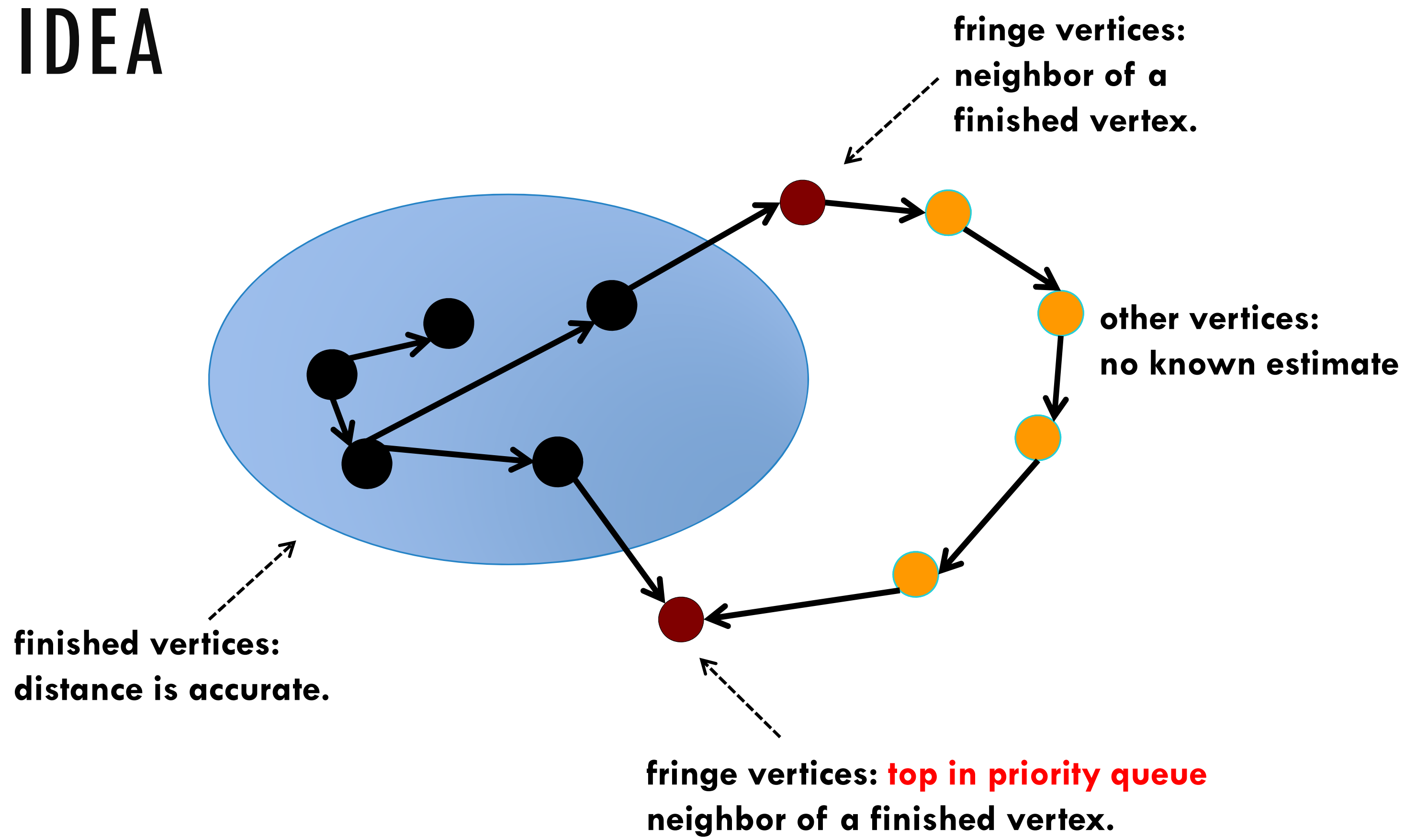
**Proof Strategy:** (like recursion)

***Base case:*** Show the statement is true for some base case

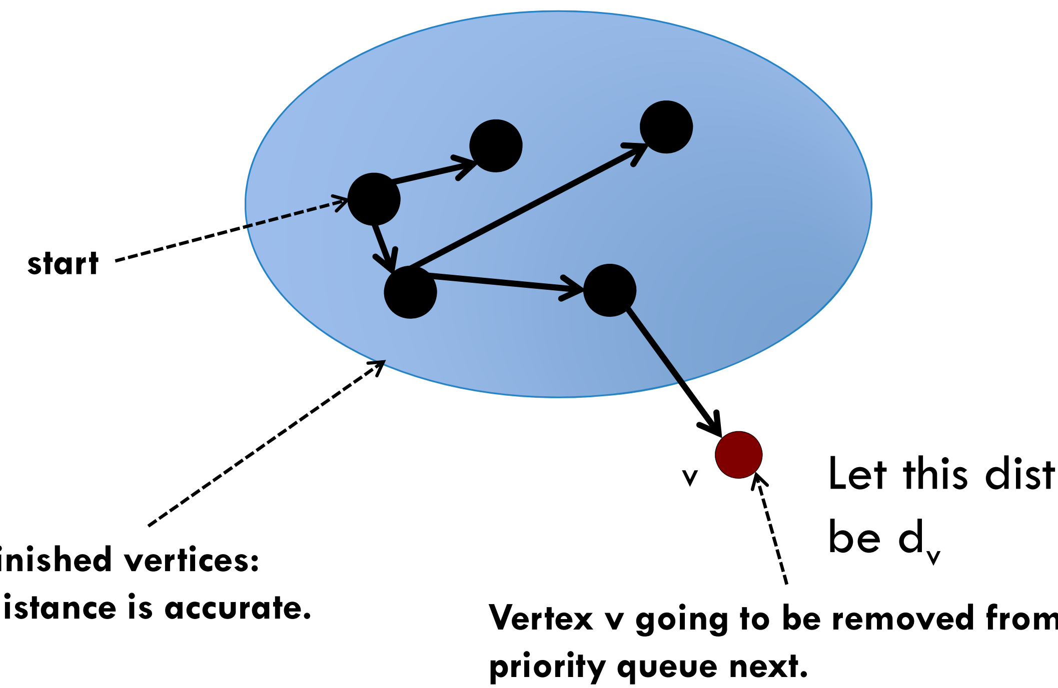
***Inductive hypothesis:*** Assume the statement is true for some k

***Inductive step:*** Show the statement holds for k+1

# THE IDEA


fringe vertices:
neighbor of a
finished vertex.
other vertices:
no known estimate
finished vertices:
distance is accurate.
fringe vertices: top in priority queue
neighbor of a finished vertex.

# PROOF (SKETCH) BY INDUCTION




Let this distance of v be $d_v$

# PROOF (SKETCH) BY INDUCTION

- **Base Step:** only "finished" vertex is the source.
- **Inductive hypothesis:**
  - Assume dist[v] $= \delta(s, v)$ at step k
    - for all finished vertices
- **Inductive step:**
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

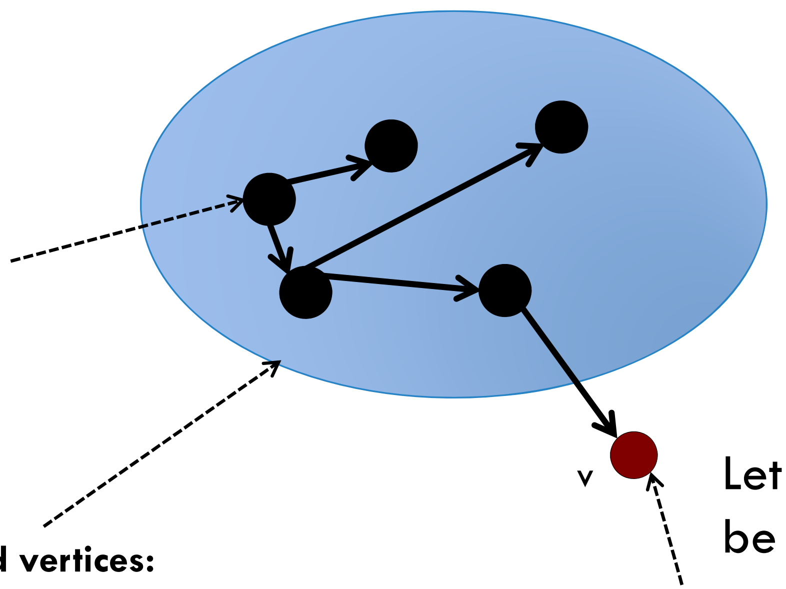
**Proof Strategy:** (like recursion)

***Base case:*** Show the statement is true for some base case

***Inductive hypothesis:*** Assume the statement is true for some k

***Inductive step:*** Show the statement holds for k+1

# PROOF (SKETCH) BY INDUCTION



start

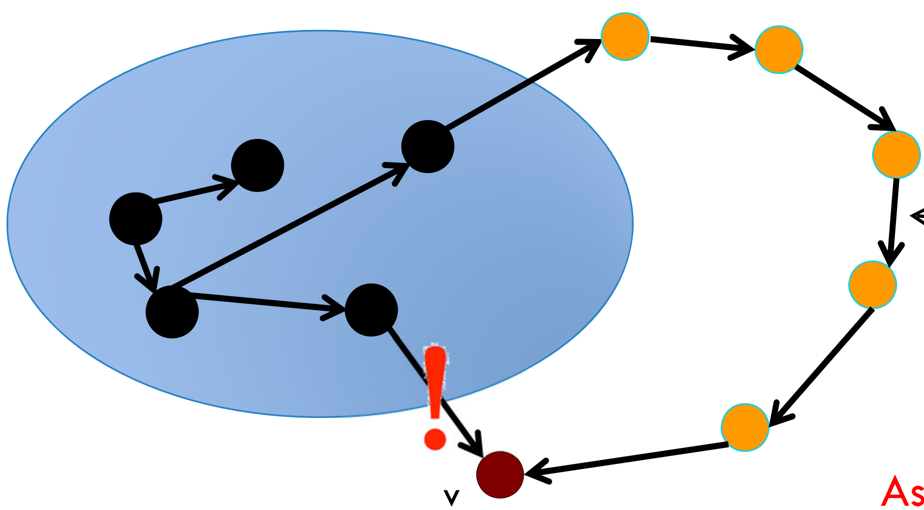
finished vertices:
distance is accurate.

v

Let this distance of v be $d_v$

Vertex v going to be removed from priority queue next.

# PROOF (SKETCH) BY INDUCTION



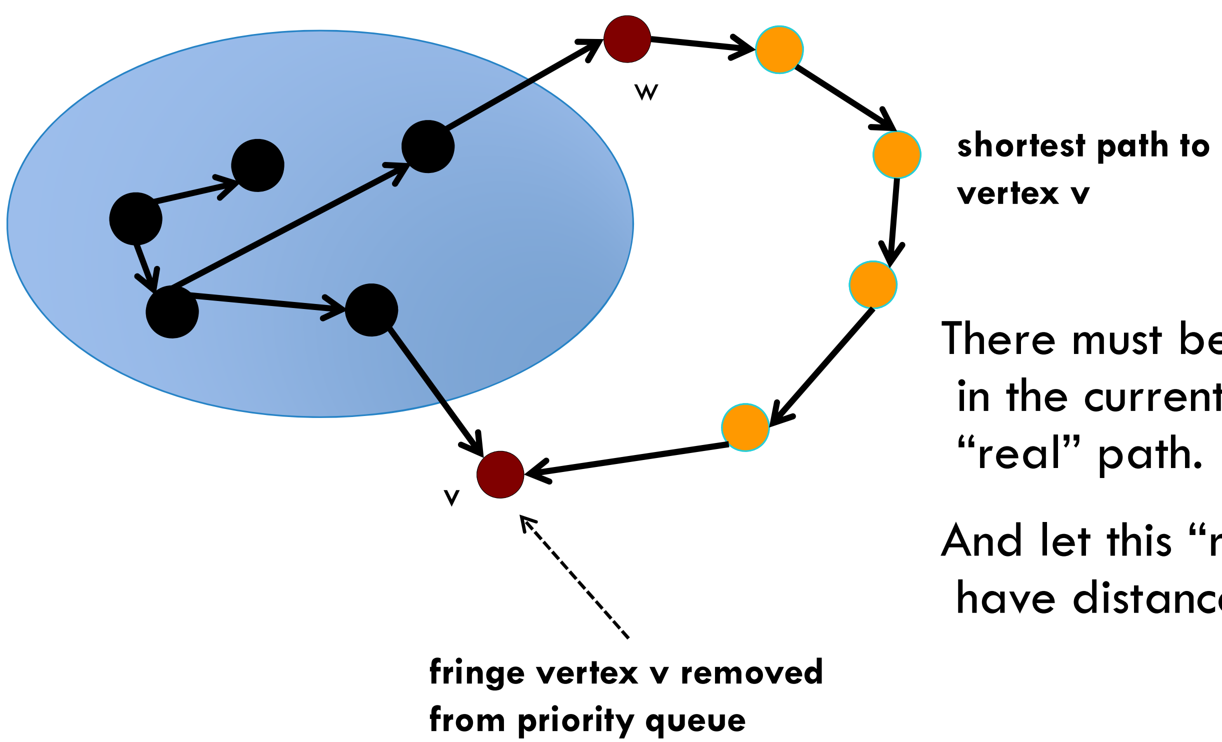

**Assume this is the "real" shortest path to v**

**fringe vertex v removed from priority queue**

Assume the current estimate is **not** the shortest path.

(we're going to show a contradiction)

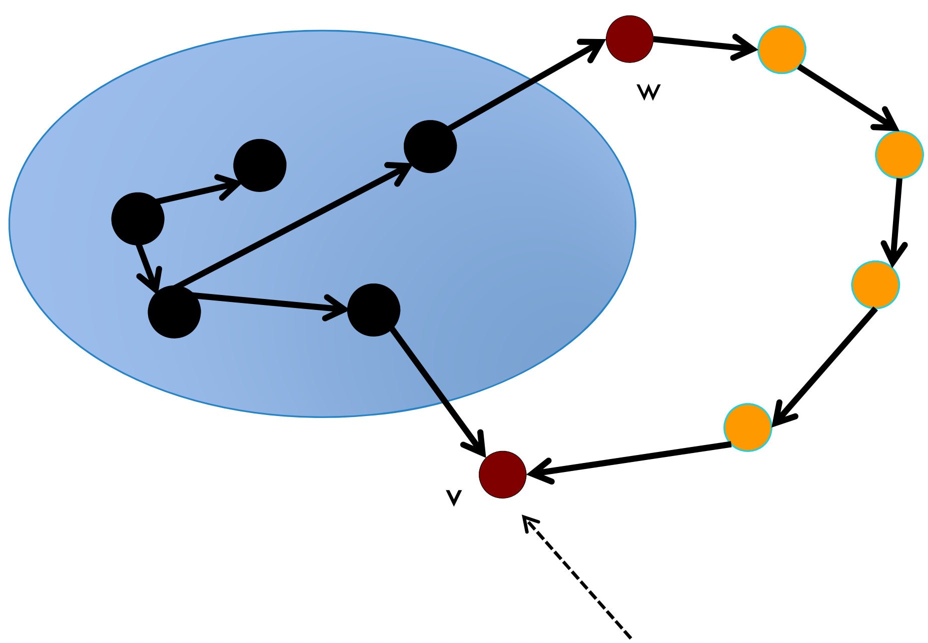
# PROOF (SKETCH) BY INDUCTION




There must be a vertex w in the current PQ on this "real" path.

And let this "real" path have distance $r_v$

# PROOF (SKETCH) BY INDUCTION




**shortest path to vertex v**

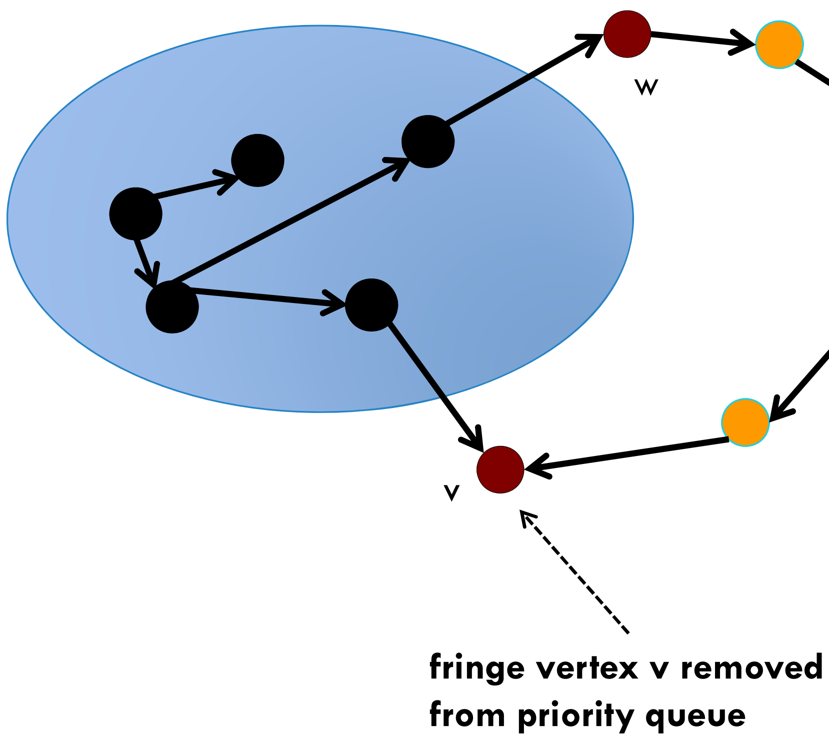
**fringe vertex v removed from priority queue**

If P is shortest path to v, then prefix of P is shortest path to w.

Then `distTo[w]` is accurate. And

`distTo[w]` $< r_v < d_v =$ `distTo[v]`

# PROOF (SKETCH) BY INDUCTION



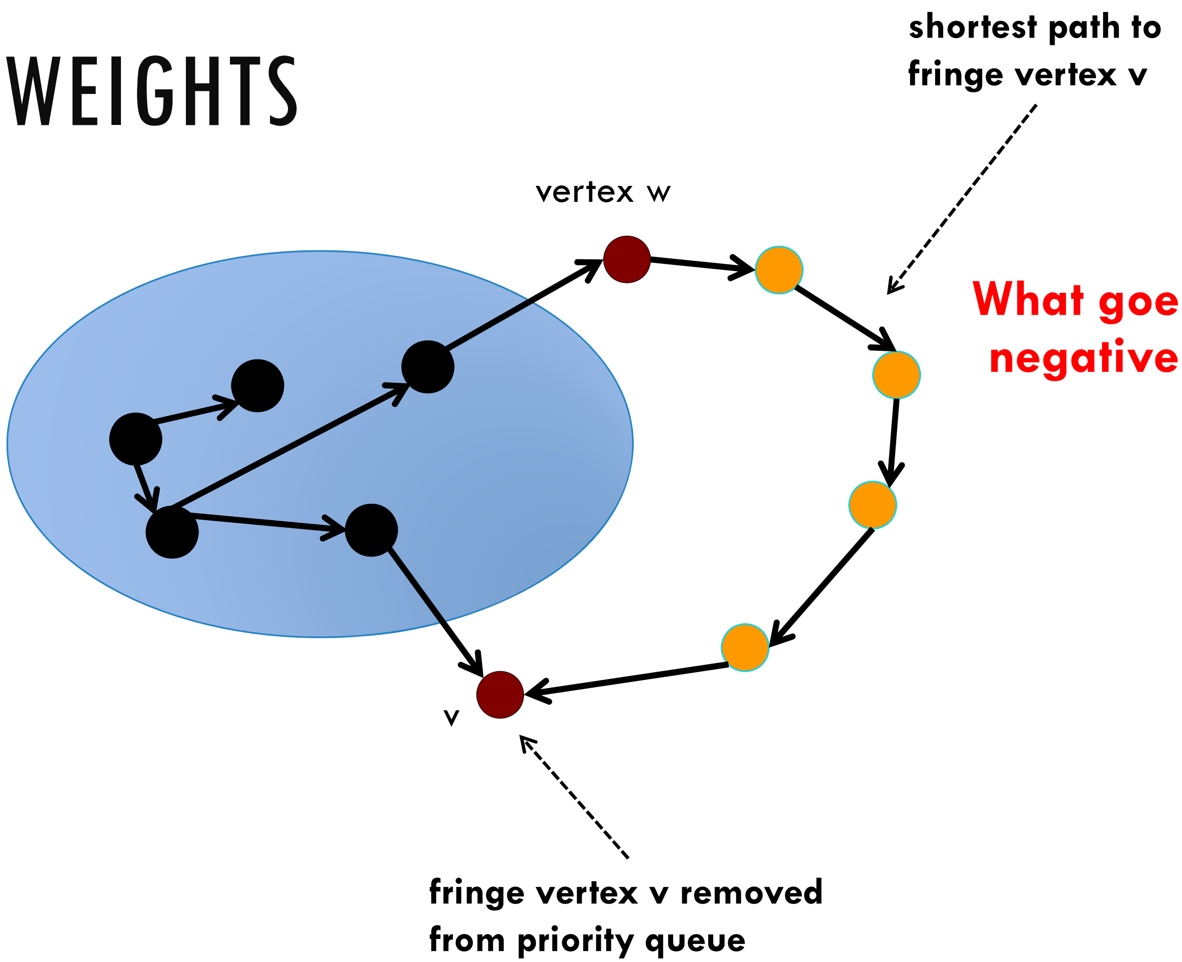

But:
 `distTo[w] >= distTo[v]`
 according to PQ!

**Contradiction!** Because v is the min in PQ!

So, the claim is true.

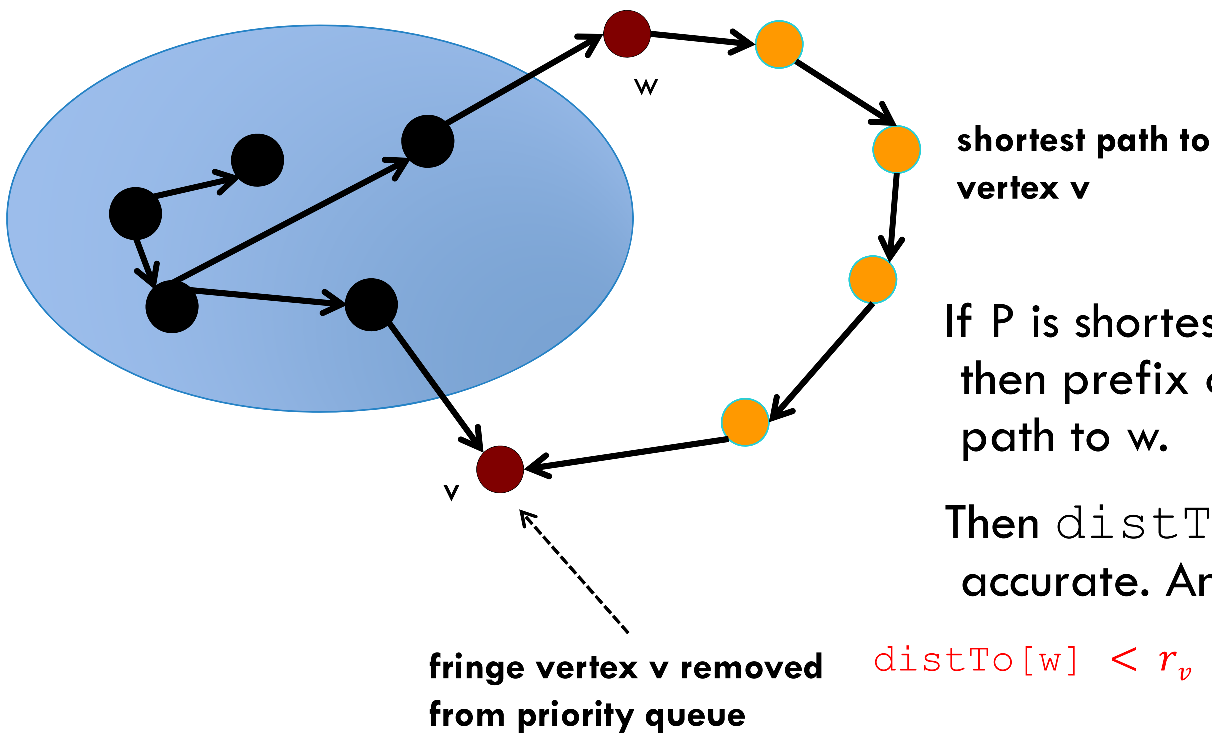
And inductive step holds. ■

# NEGATIVE WEIGHTS




**What goes wrong with negative weights?**

# PROOF (SKETCH) BY INDUCTION

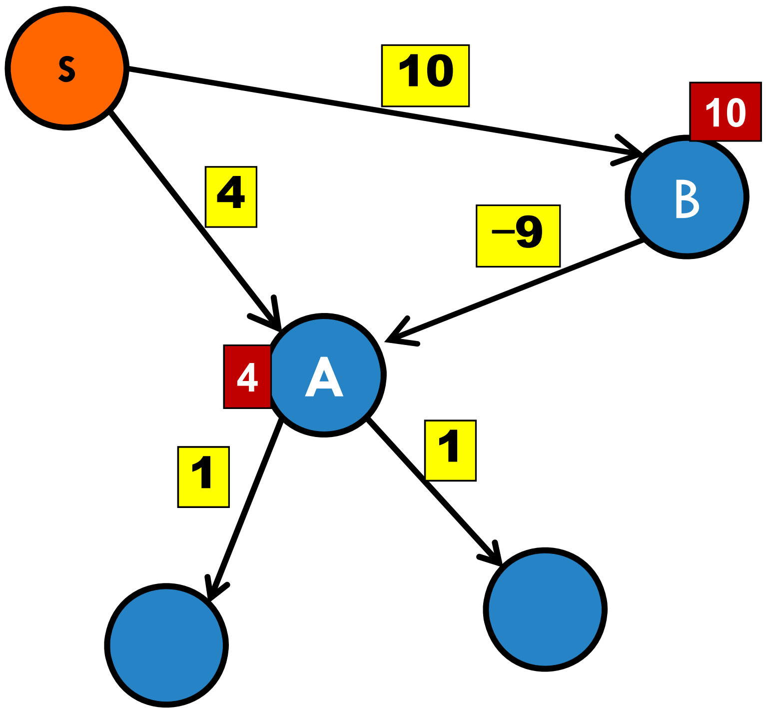



If P is shortest path to v, then prefix of P is shortest path to w.

Then `distTo[w]` is accurate. And

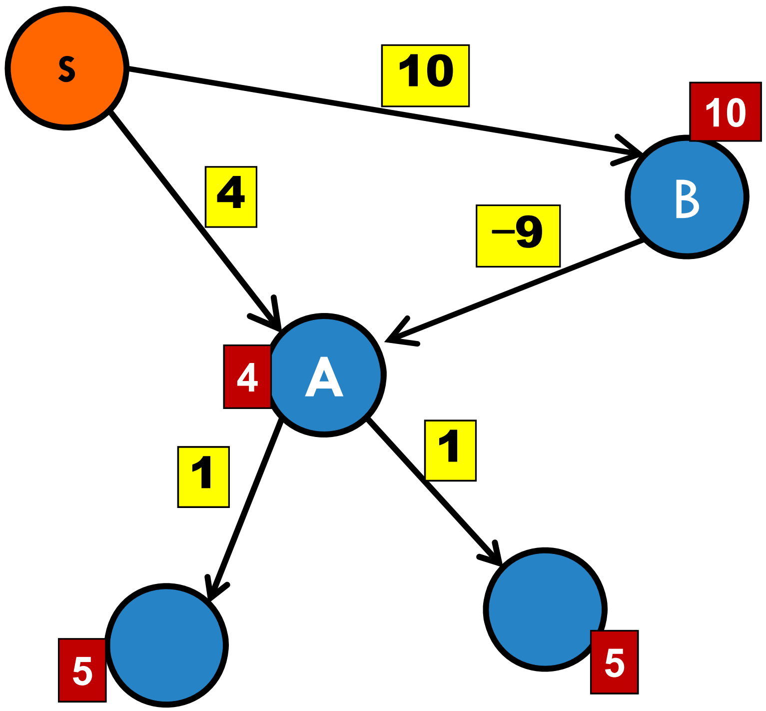
`distTo[w]` $< r_v < d_v =$ `distTo[v]`

# NEGATIVE WEIGHTS




**Step 1:** Remove A.
Relax A.
Mark A done.

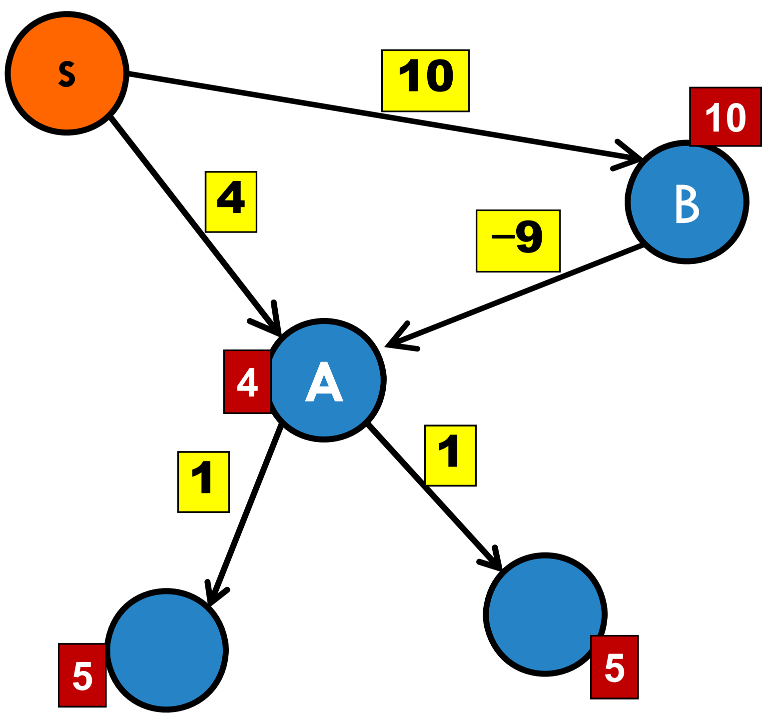
# NEGATIVE WEIGHTS




**Step 1:** Remove A.
Relax A.
Mark A done.

…

# NEGATIVE WEIGHTS




**Step 1:** Remove A.
Relax A.
Mark A done.

…

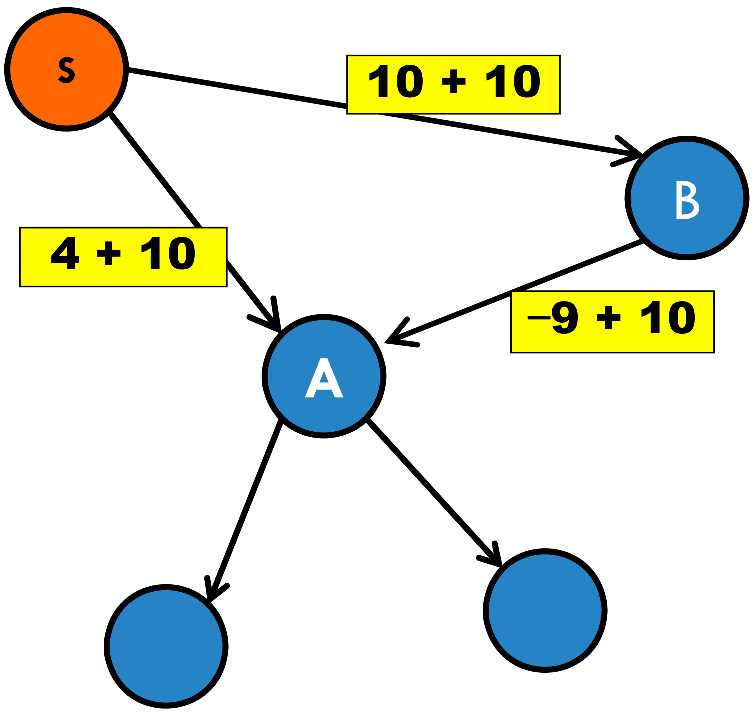
**Step 4:** Remove B.
Relax B.
Mark B done.

**Oops:** We need to update A.

# NEGATIVE WEIGHTS

Poll Everywhere

https://bit.ly/2LvG9bq



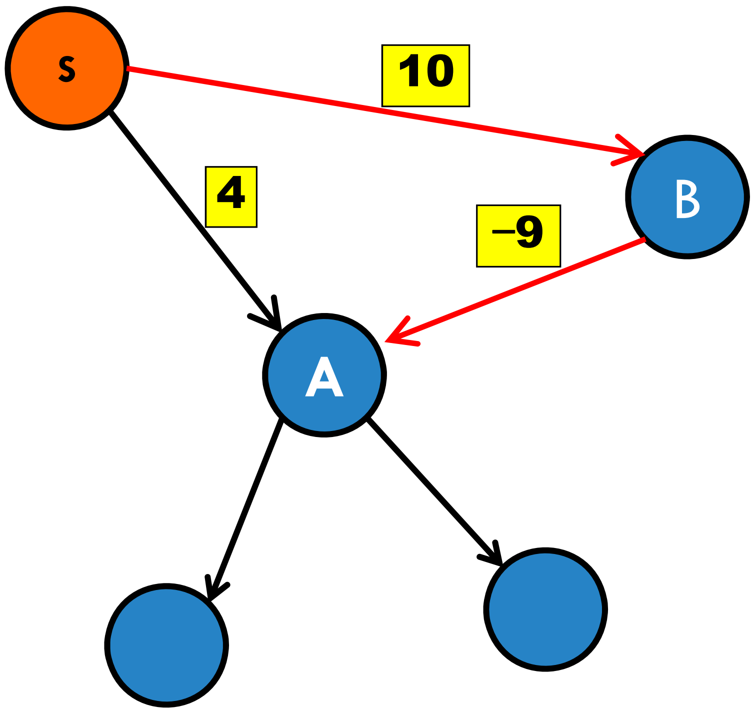

Can we re-weight the edges with some constant (10)?

A. Hell yeah!

B. Nope.. wouldn't work.

C. 

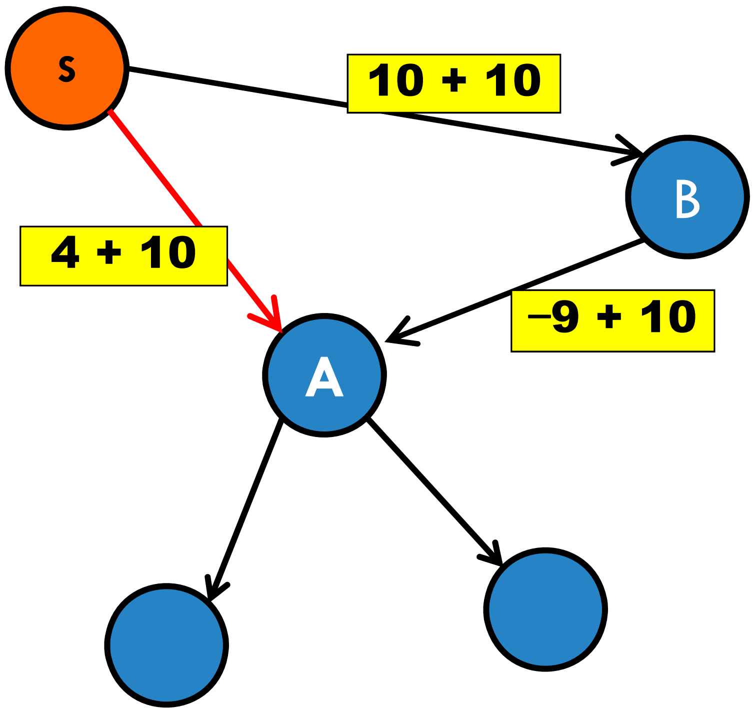
# REWEIGHTING?




Path S-B-A: 1

Path S-A: 4

# REWEIGHTING?




Path S-B-A: 21

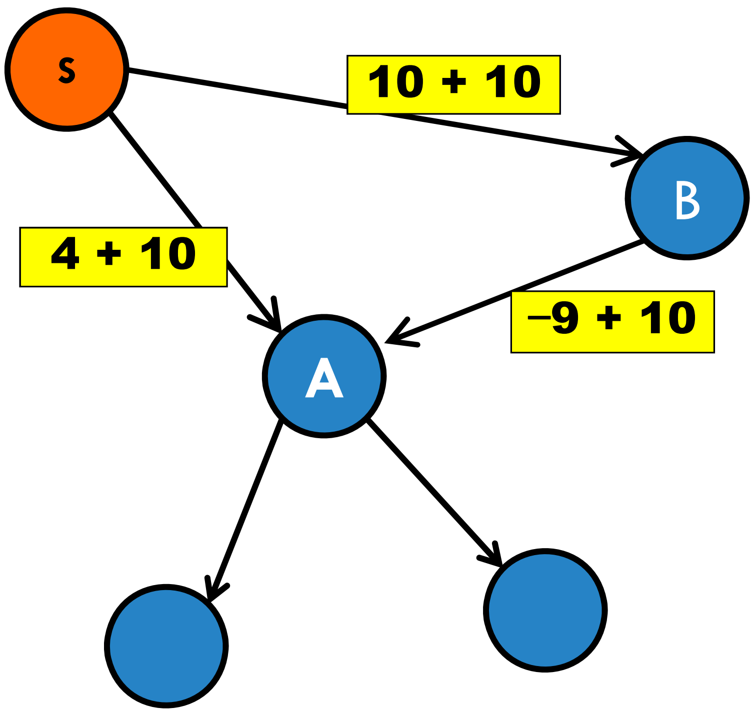
Path S-A: 14

**The shortest path is no longer preserved!**

# NEGATIVE WEIGHTS

Poll Everywhere

https://bit.ly/2LvG9bq




Can we re-weight the edges with some constant?

A. Hell yeah!

B. Nope.. wouldn’t work.

C. 

# COMPARISON TO BFS AND DFS

Maintain a set of explored vertices.

Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

- **BFS:** vertex that was discovered **least** recently.
- **DFS:** vertex that was discovered **most** recently.
- **Dijkstra's:** vertex that is **closest** to source.

# COMPARISON TO BFS AND DFS

Maintain a set of explored vertices.

Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

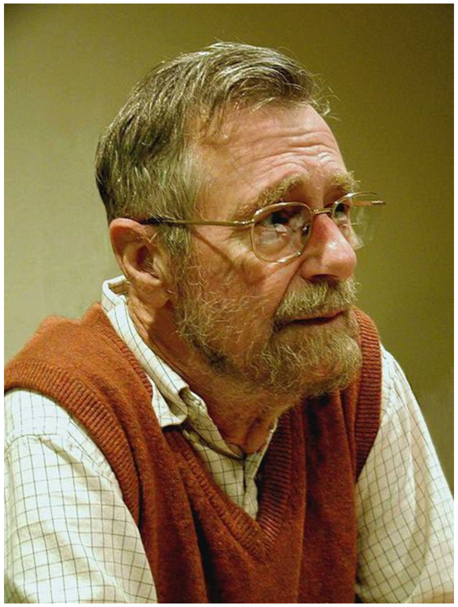
- **BFS:** Use **queue.**
- **DFS:** Use **stack.**
- **Dijkstra's:** Use **priority queue.**

**The "same" algorithm again!**

# DIJKSTRA'S ALGORITHM: SUMMARY

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.
- $O((V + E) \log V)$ time (with AVL tree Priority Queue)
- No negative weight edges!



# TODAY: SPECIAL CASES

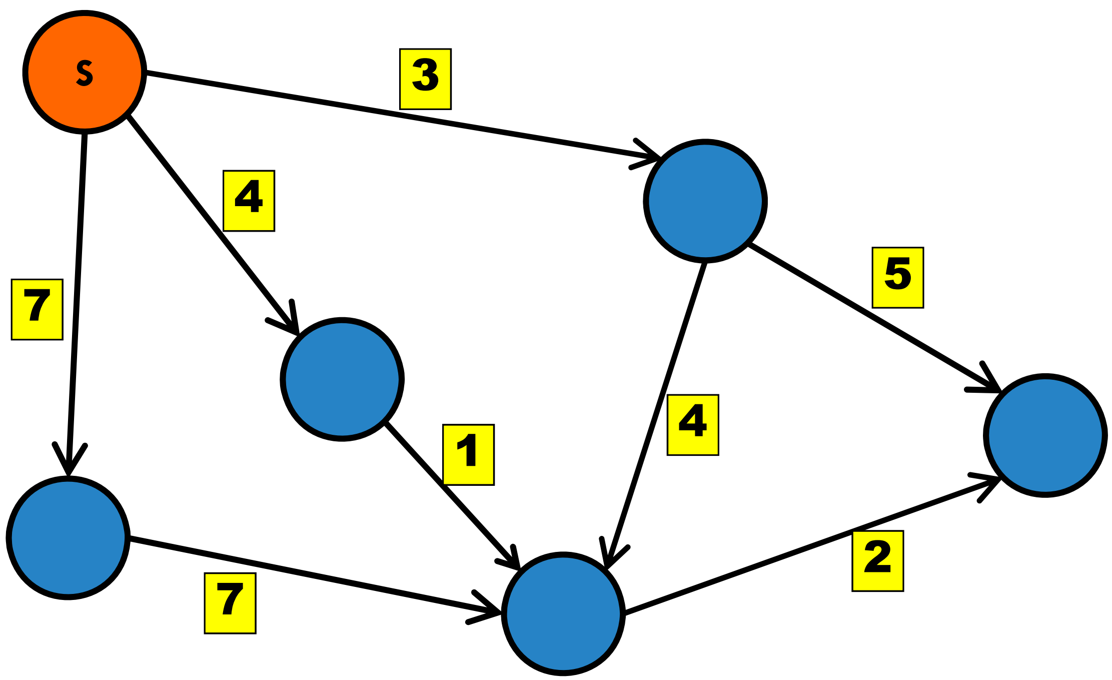
| Condition | Algorithm | Time Complexity |
| --- | --- | --- |
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | $O((V+E)log\, V)$ |
| On Tree | BFS / DFS order | $O(V)$ |
| On DAG | Topological sort order | |

# QUESTIONS?

Poll Everywhere

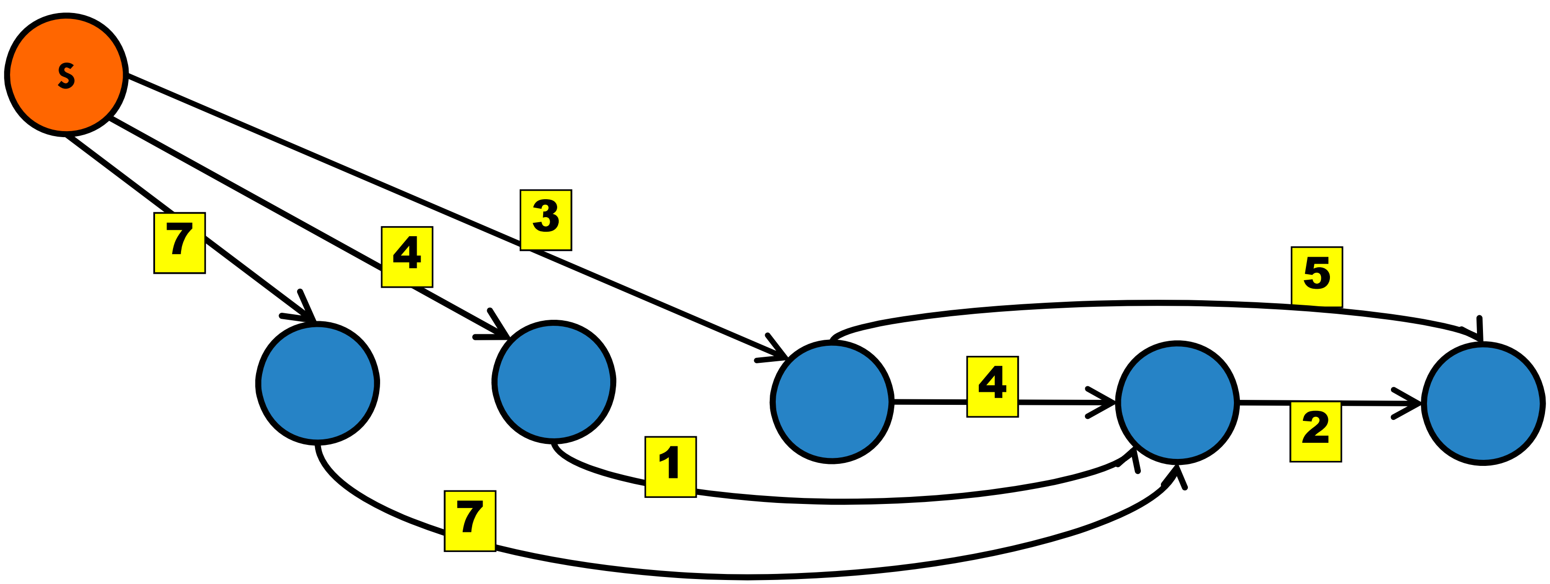
https://bit.ly/2LvG9bq

# DIRECTED ACYCLIC GRAPH (DAG)



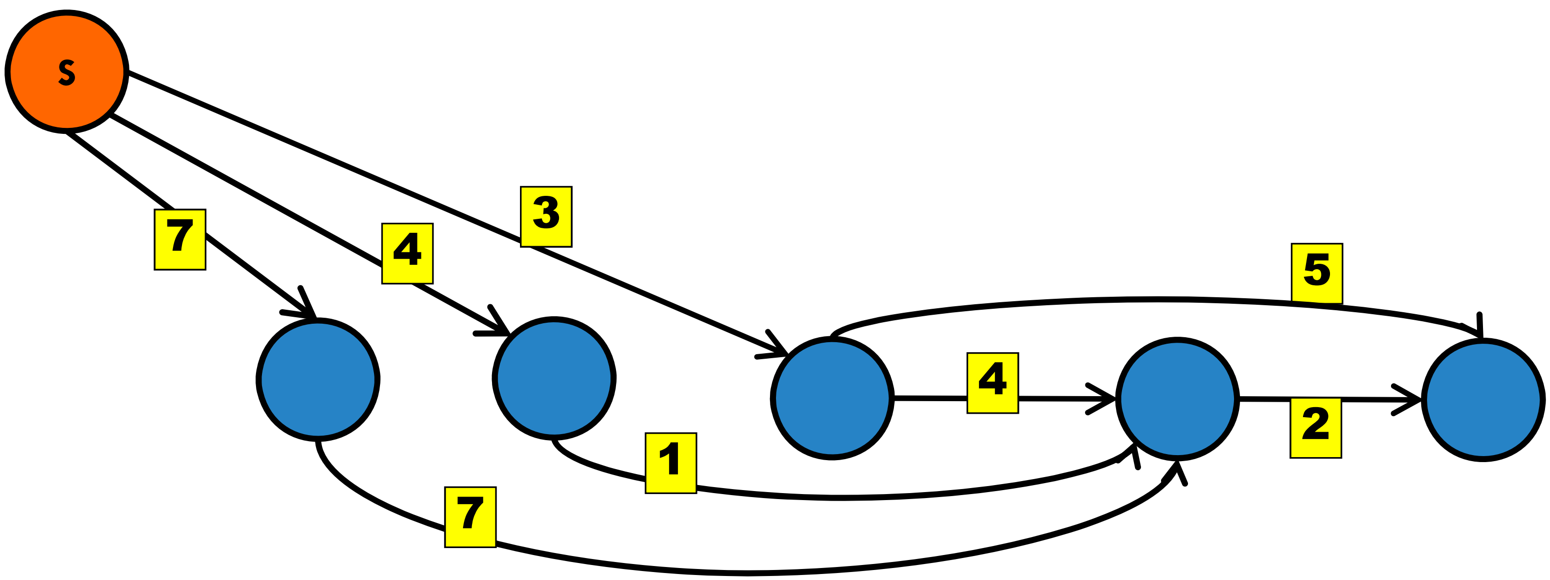
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



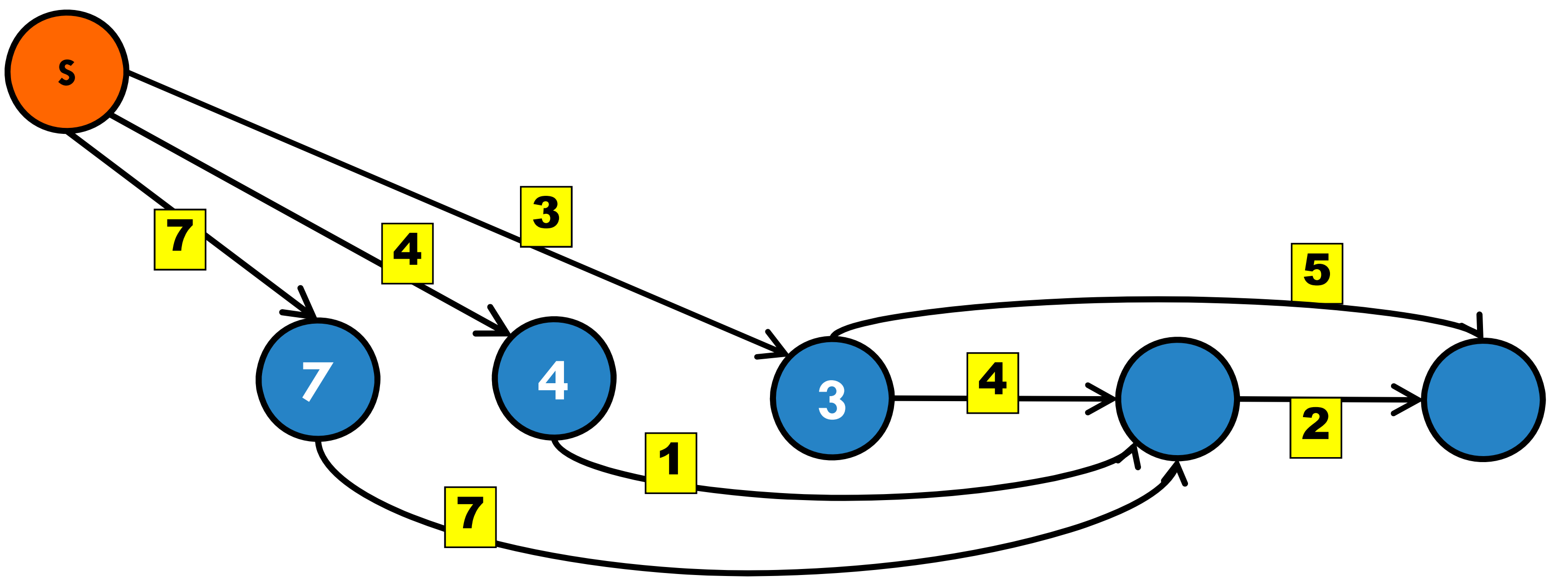
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



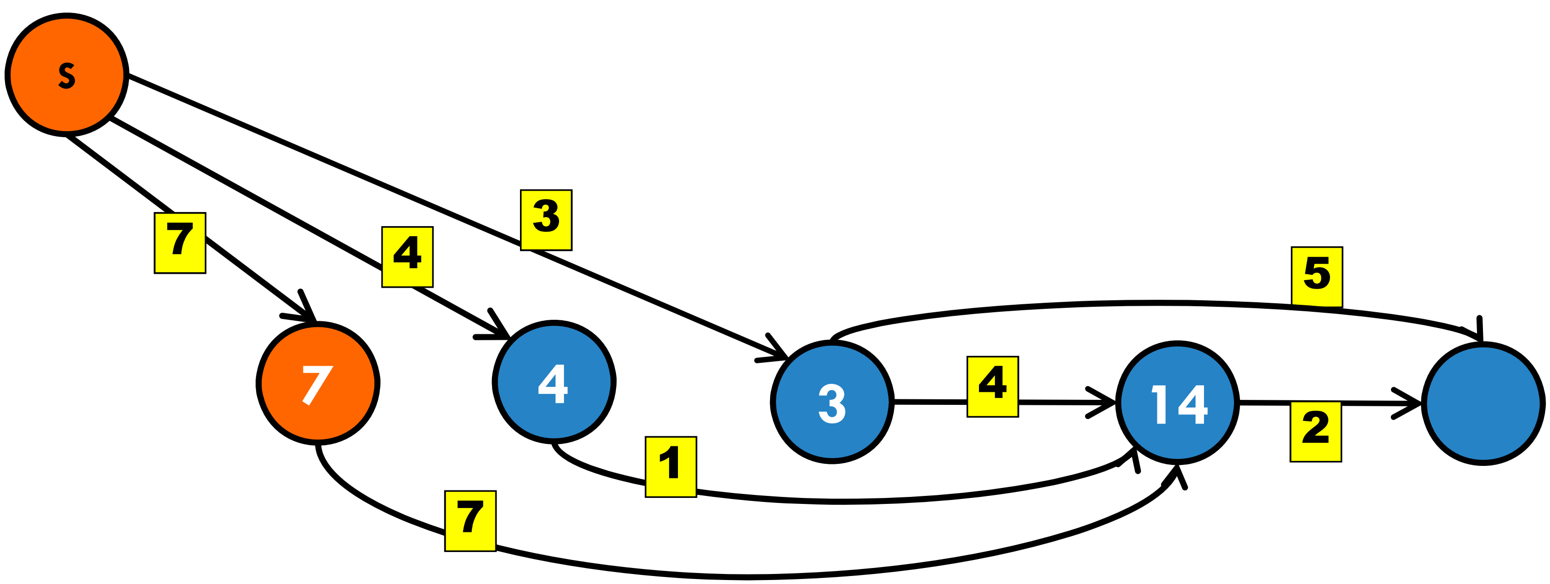
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



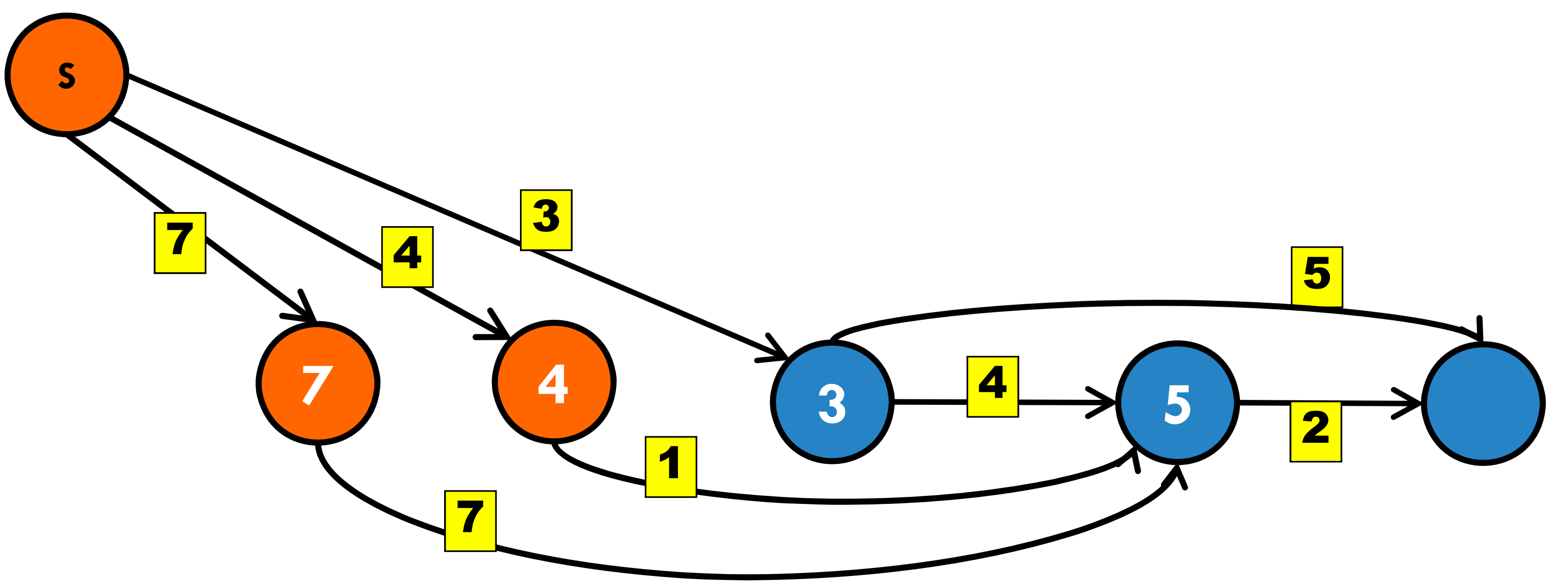
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



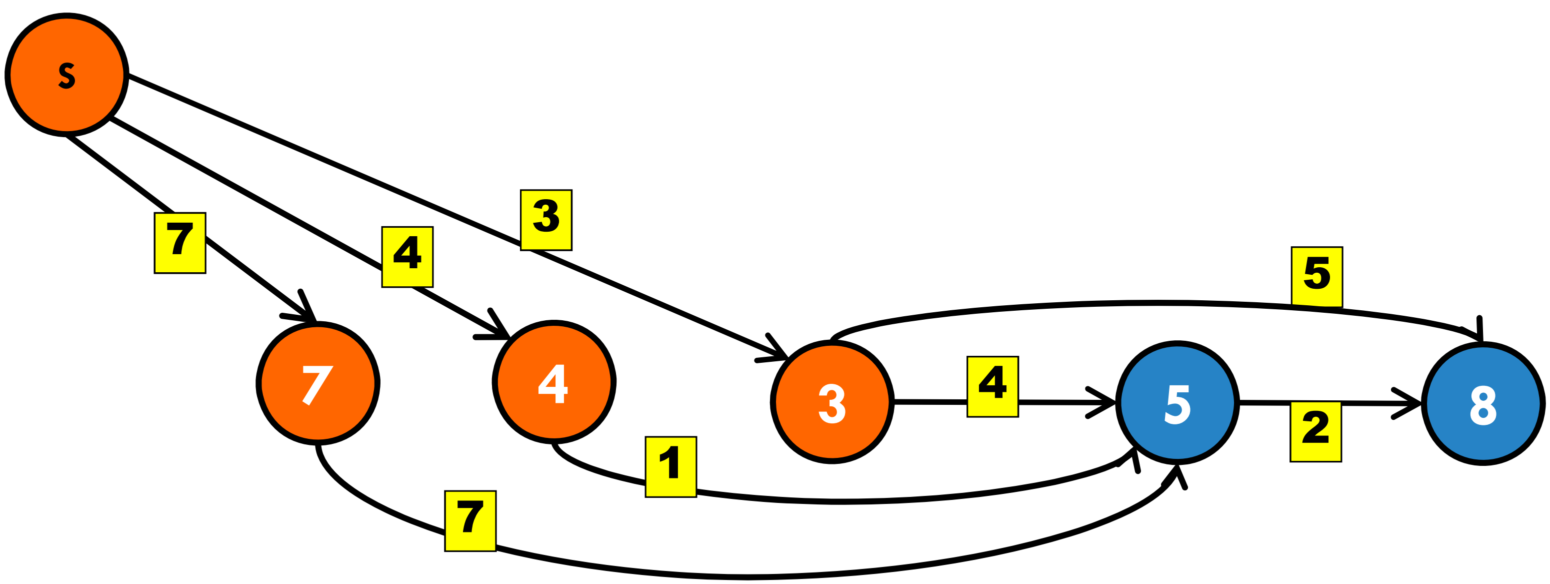
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



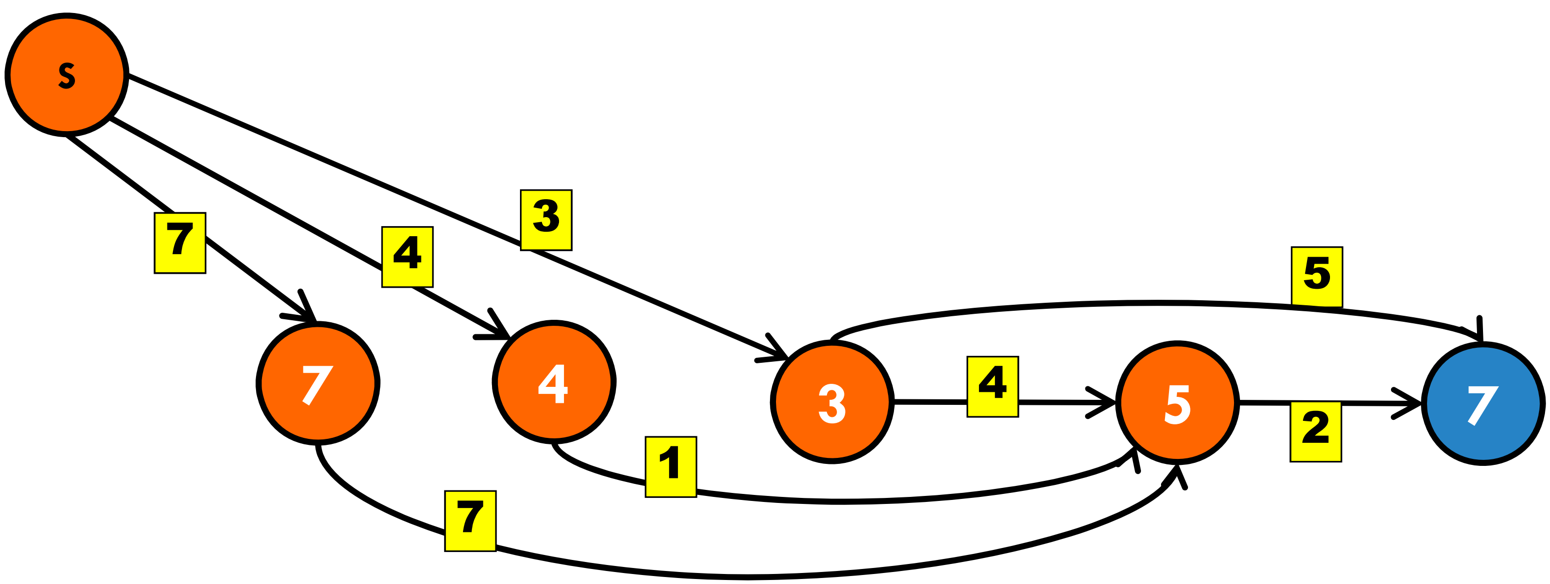
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



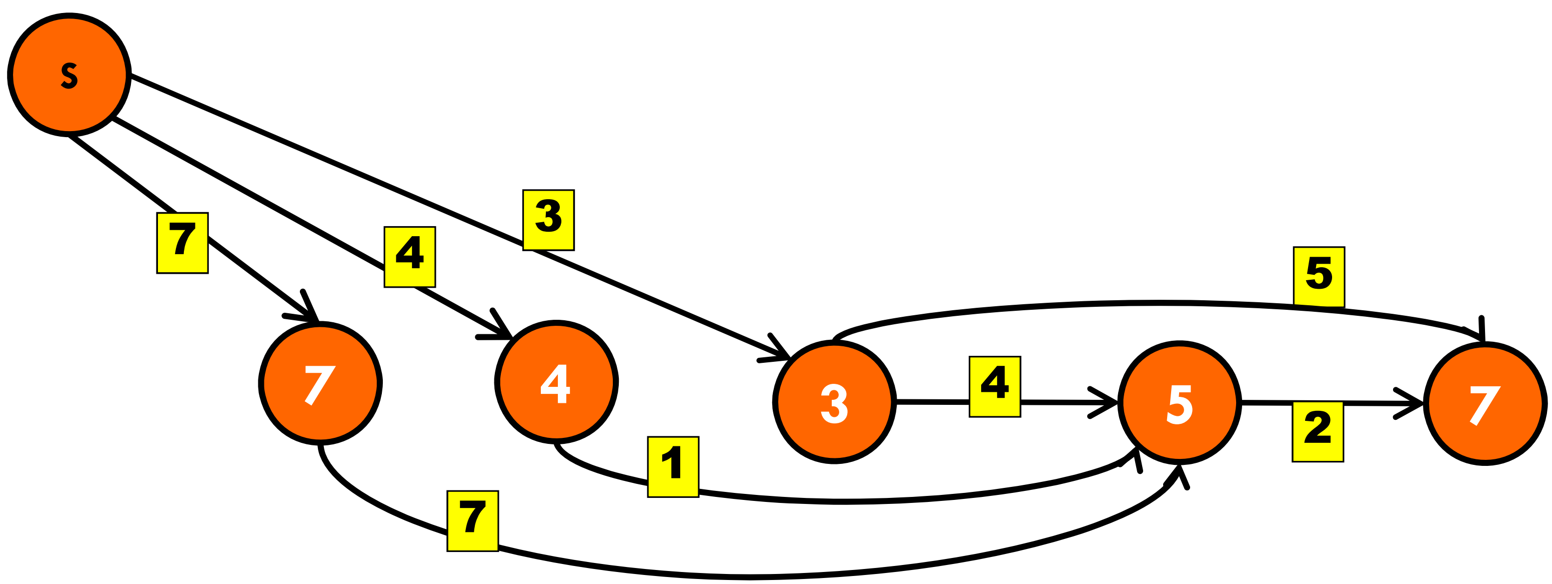
# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.

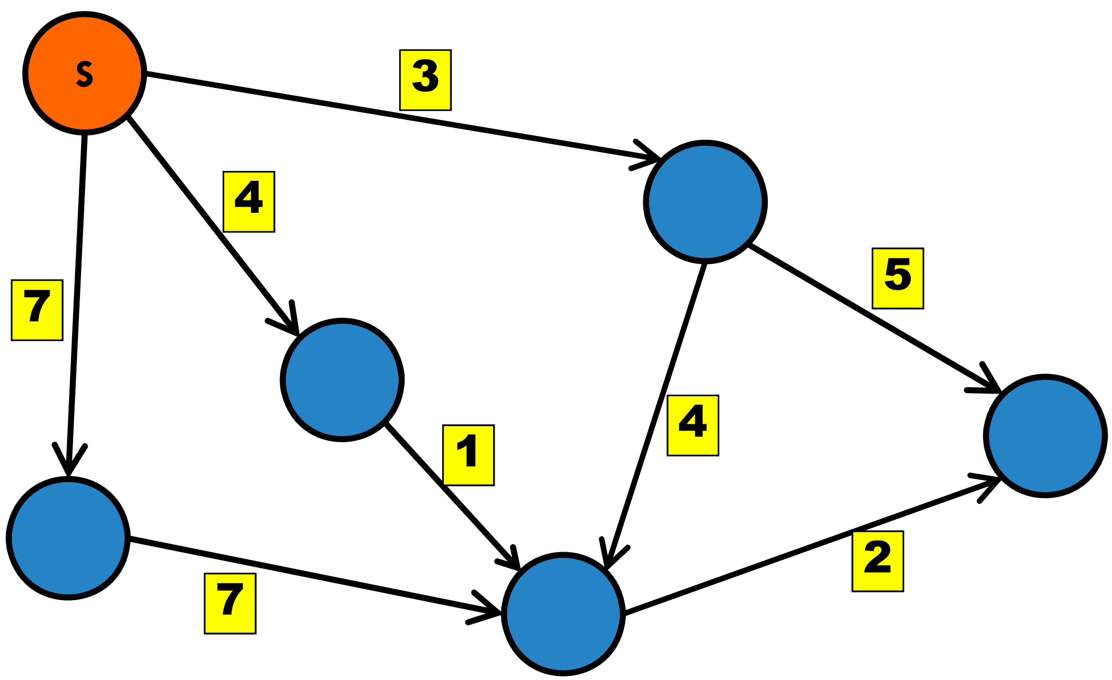


# DIRECTED ACYCLIC GRAPH (DAG)

1. Topological sort
2. Relax in order.



# WHY TOPOLOGICAL ORDER?



# PATH RELAXATION PROPERTY

**Lemma 5.** If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to $v_k$ and we relax the edges of $p$ in the order

$$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$$

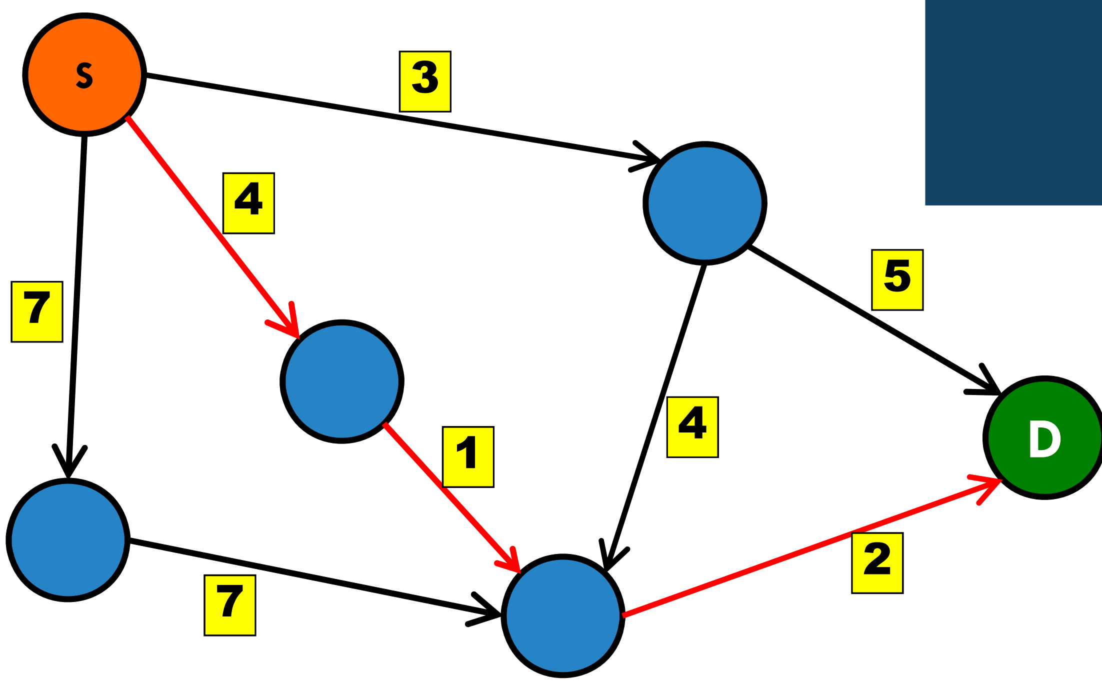
Then $d[v_k] = \delta[v_k]$.

This property holds ***regardless of any other relaxation steps that occur*** (even **intermixed**)

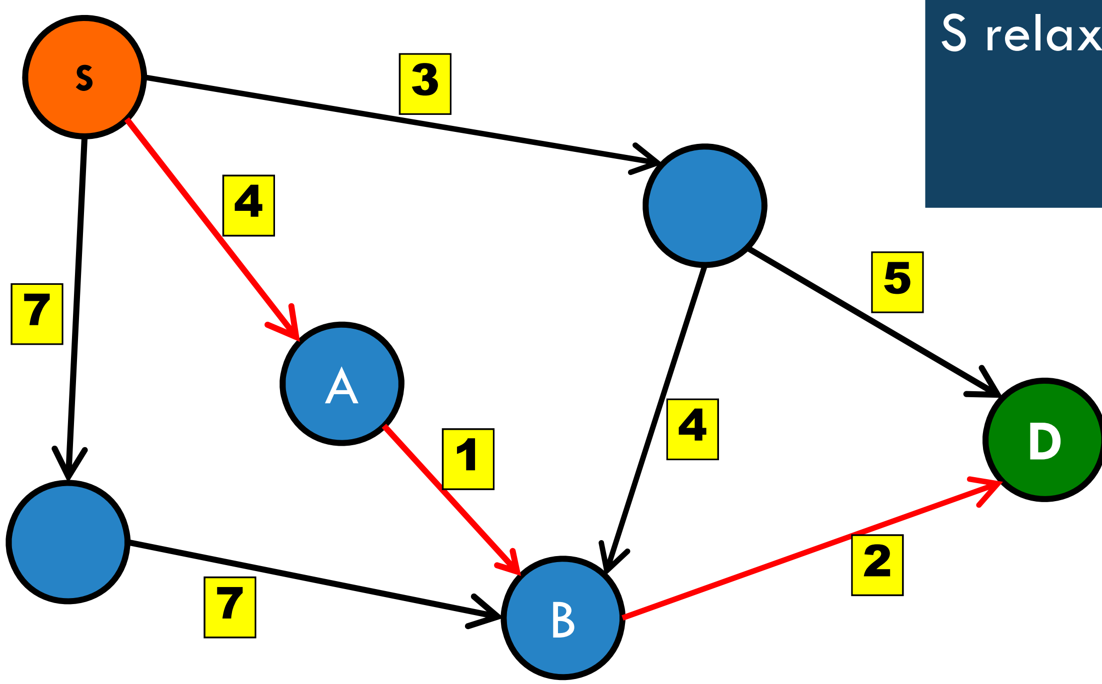
- E.g., $(v_0, v_1), (v_i, v_j), (v_1, v_2), \dots, (v_{k-1}, v_k)$ will *still* result in $d[v_k] = \delta[v_k]$.

# WHY TOPOLOGICAL ORDER?

Fix S-D shortest path.



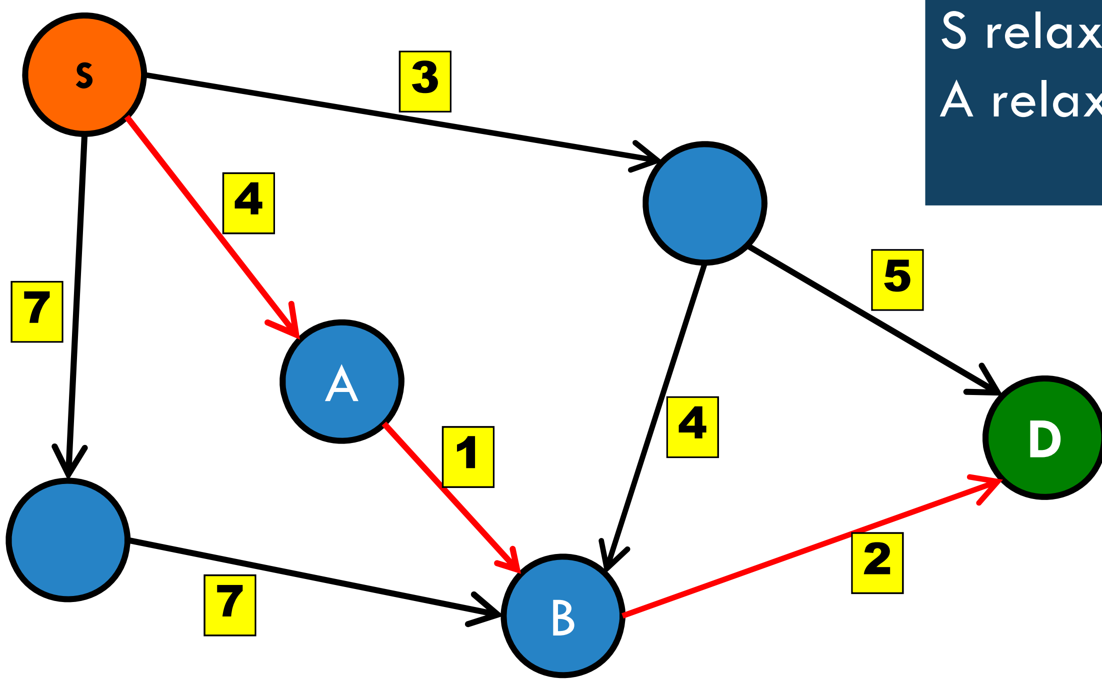
# WHY TOPOLOGICAL ORDER?




Fix S-D shortest path.

S relaxed before A.

# WHY TOPOLOGICAL ORDER?

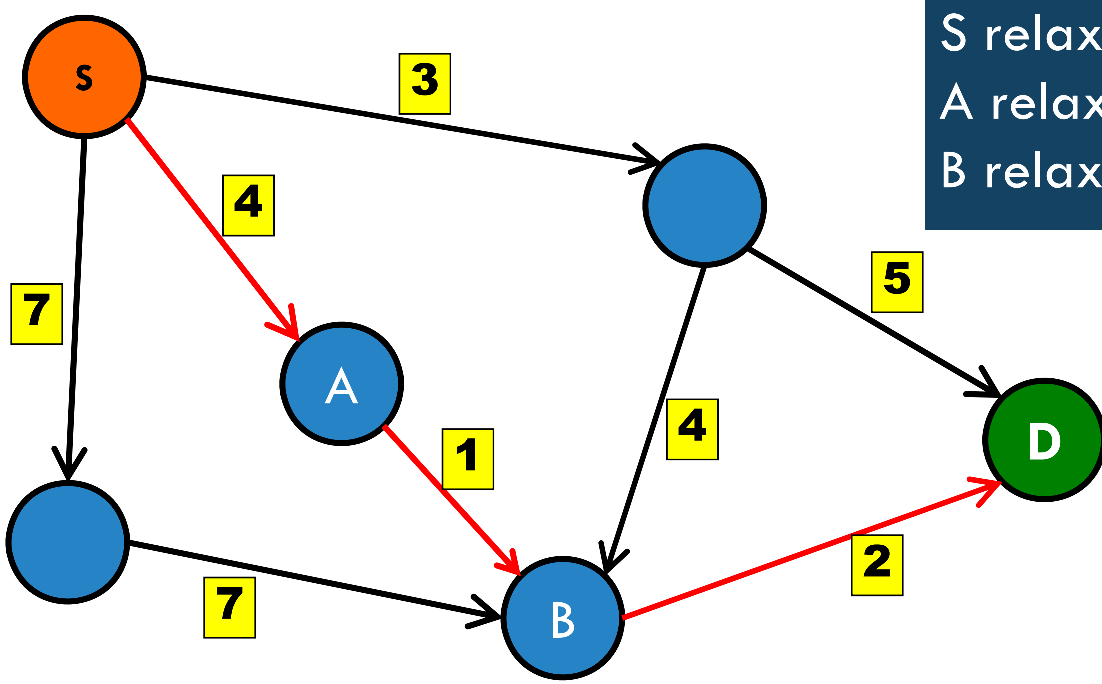



Fix S-D shortest path.

S relaxed before A.

A relaxed before B.

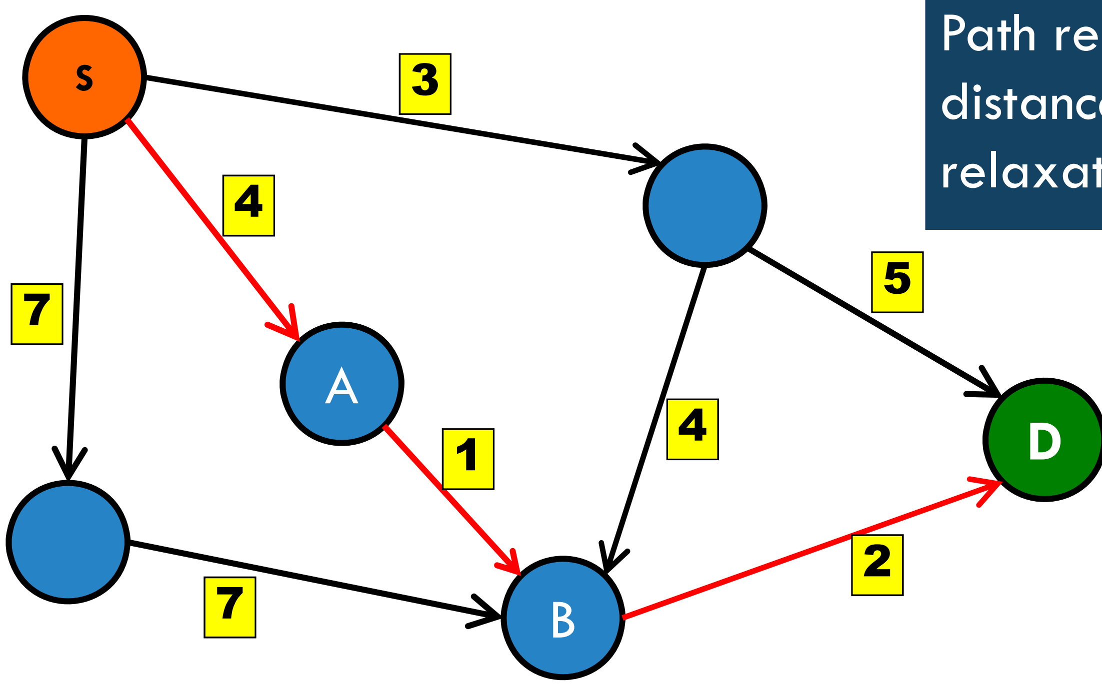
# WHY TOPOLOGICAL ORDER?




Fix S-D shortest path.

S relaxed before A.
A relaxed before B.
B relaxed before D.

# WHY TOPOLOGICAL ORDER?




Fix S-D shortest path.

Path relaxed in-order, so distance is correct after relaxation.

# TODAY: SPECIAL CASES

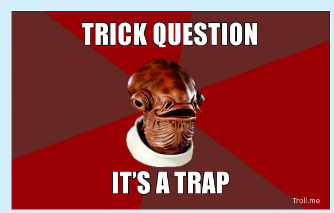
| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra’s Algorithm | $O((V+E)log\,V)$ |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Topological Sort | $O(V+E)$ |

# WHAT IF WE WANT THE MIN DISTANCE CYCLE?

Given a unweighted graph with non-negative edges

Find the shortest possible route that visits each vertex exactly once and returns to the origin vertex.

How fast can I solve this problem?

A. O(VE) ... just use Bellman-Ford
B. O(V + E) ... use BFS/DFS
C. $O((V+E)log\,V)$ use Dijkstra's algorithm.
D. 

# WHAT IF WE WANT THE MIN DISTANCE CYCLE?

Given a unweighted graph with non-negative edges

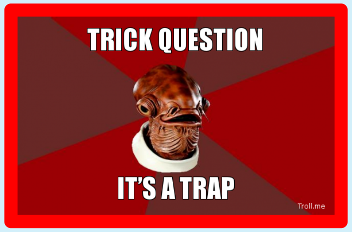
Find the shortest possible route that visits each vertex exactly once and returns to the origin vertex.

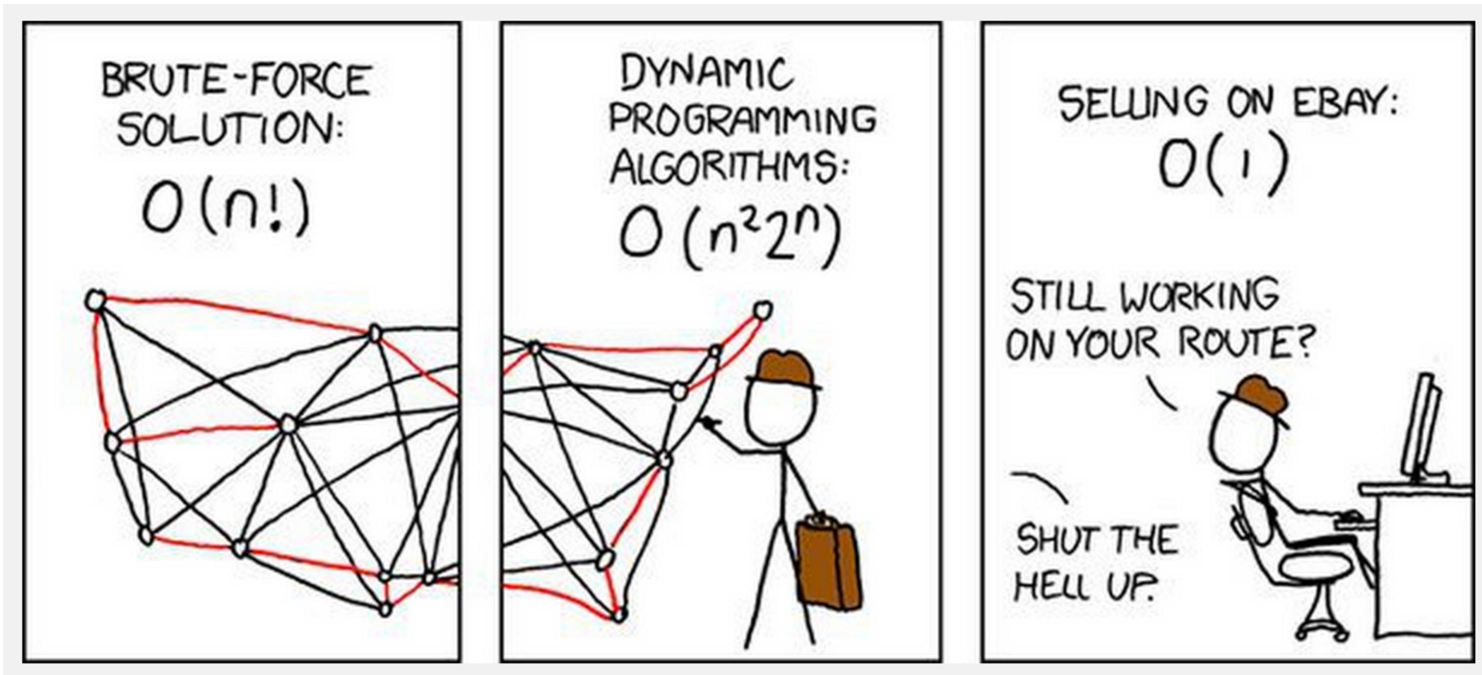
**This is the famous traveling salesman problem. It is NP-Hard!**

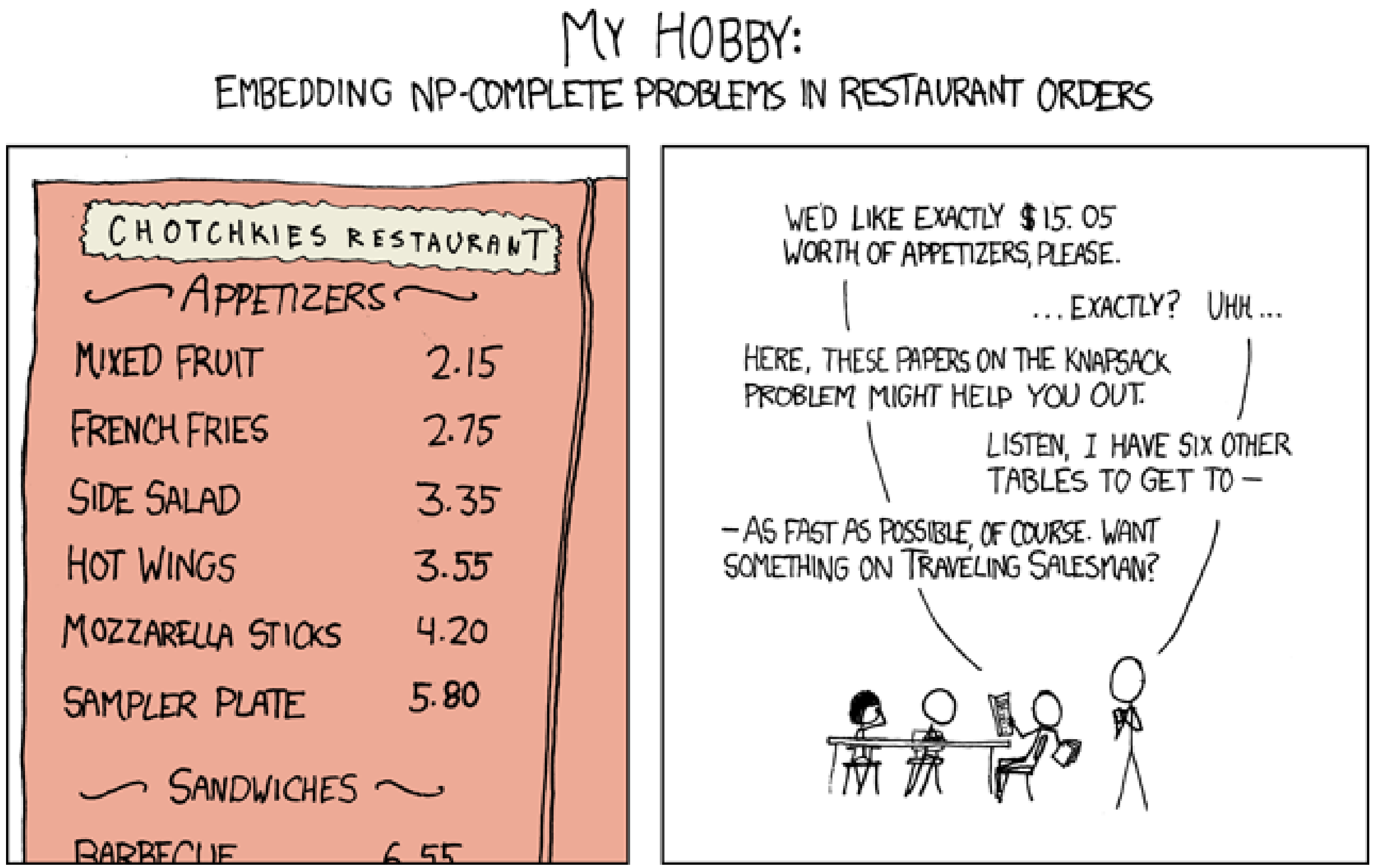
**"Non-deterministic Polynomial acceptable problems"**

**Dynamic programming solution is:** $\boldsymbol{O(V^2\ 2^V)}$

How fast can I solve this problem?

A. O(VE) ... just use Bellman-Ford
B. O(V + E) ... use BFS/DFS
C. $O((V+E)log\,V)$ use Dijkstra's algorithm.
**D.** 


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

# SUMMARY

By the end of this session, students should be able to:

- State the **special graphs** that we can apply **faster shortest path methods.**
- Describe **each special graph** and **the algorithm used**.
- Explain **Dijkstra's algorithm** and relate it to BFS/DFS.
- Analyze the **computational complexity of Dijkstra's algorithm**.

# TODAY: SPECIAL CASES

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | $O((V+E)log\, V)$ |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Topological Sort | $O(V+E)$ |

Poll Everywhere

https://bit.ly/2LvG9bq

# QUESTIONS?