LECTURE 15: ALL-PAIRS SHORTEST PATHS (APSP) + REVISION

Harold Soh
harold@comp.nus.edu.sg

ADMINISTRATIVE ISSUES

Quiz1, Quiz 2, and PS1 scores are on Luminus

Please Verify

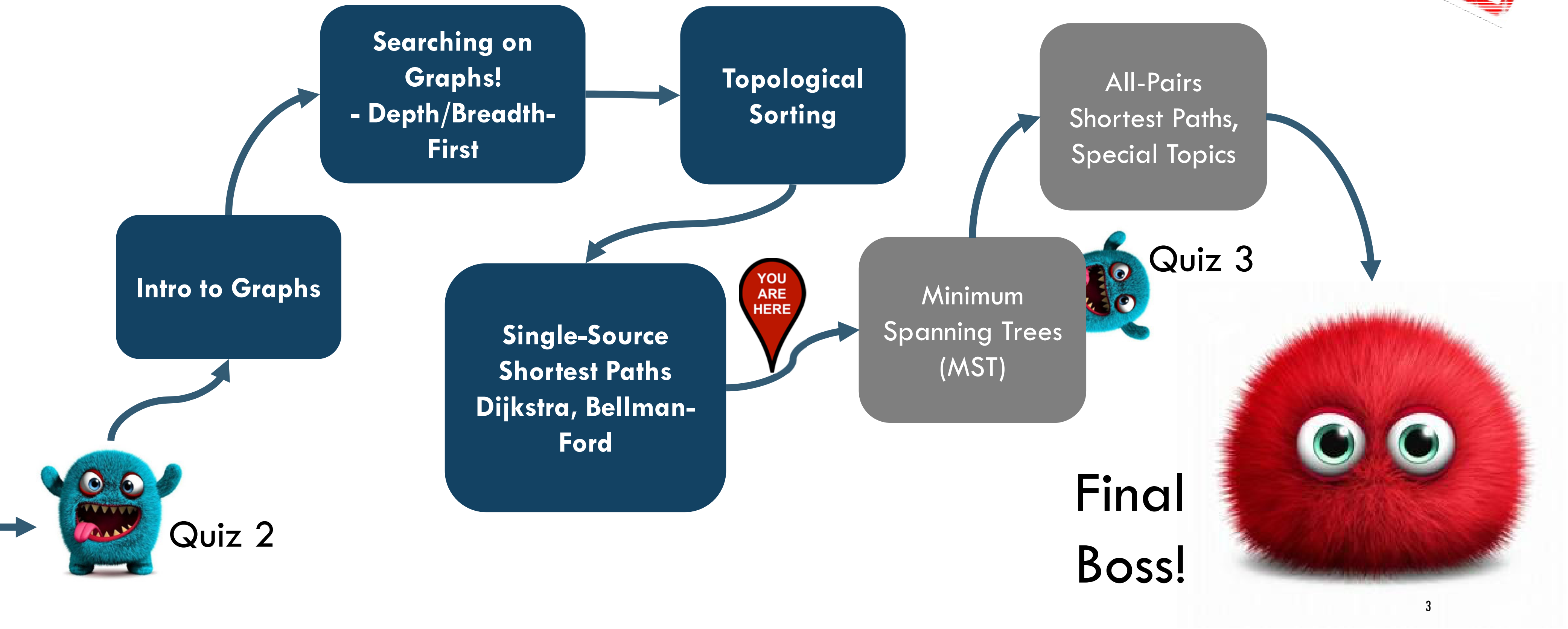
- verify all your scores = 1 participation point
- If something is incorrect, please let us know.

PS2 scores coming soon (just a few more to check).

Quiz 2 to be returned during Friday's tutorial.

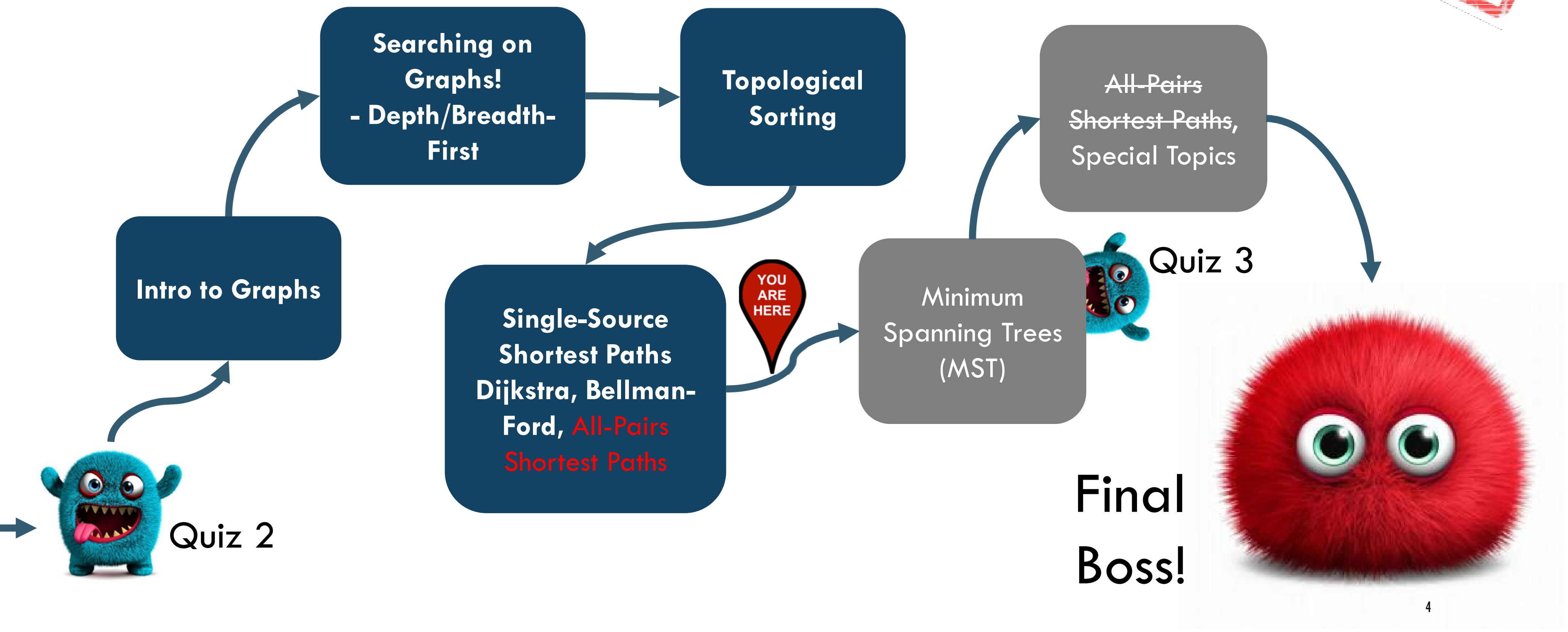
PATH TO MASTERY / COURSE STRUCTURE

DRAFT


Quiz 2
Intro to Graphs
Searching on Graphs!
- Depth/Breadth-First
Topological Sorting
Single-Source Shortest Paths Dijkstra, Bellman-Ford
YOU ARE HERE
Minimum Spanning Trees (MST)
Quiz 3
All-Pairs Shortest Paths, Special Topics
Final Boss!

PATH TO MASTERY / COURSE STRUCTURE

DRAFT


Quiz 2
Intro to Graphs
Searching on Graphs!
- Depth/Breadth-First
Topological Sorting
Single-Source Shortest Paths Dijkstra, Bellman-Ford, All-Pairs Shortest Paths
YOU ARE HERE
Minimum Spanning Trees (MST)
Quiz 3
~~All-Pairs Shortest Paths~~, Special Topics
Final Boss!

TODAY: LEARNING OUTCOMES

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

- State the **all-pairs shortest-paths** (APSP) problem
- Explain **Floyd-Warshall** and apply it to solve the APSP problem.
- Analyze the **computational complexity of Floyd-Warshall.**

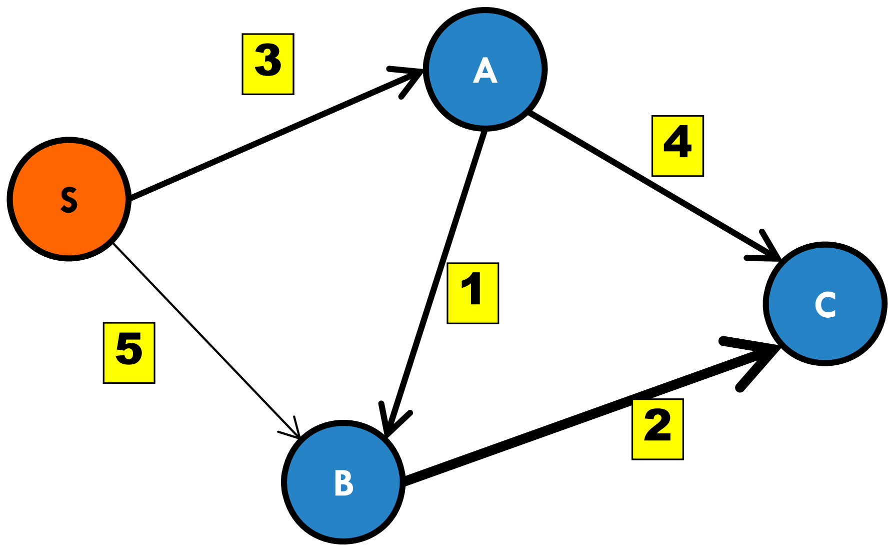
SINGLE-SOURCE SHORTEST PATHS

Input:

- Directed, weighted graph G = (V,E)
- Given source s and target t

Output:

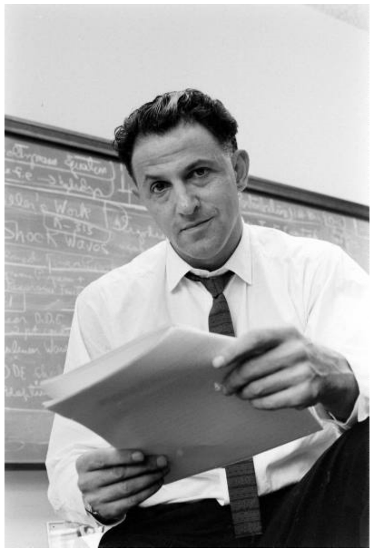
- min-distance(s, t)

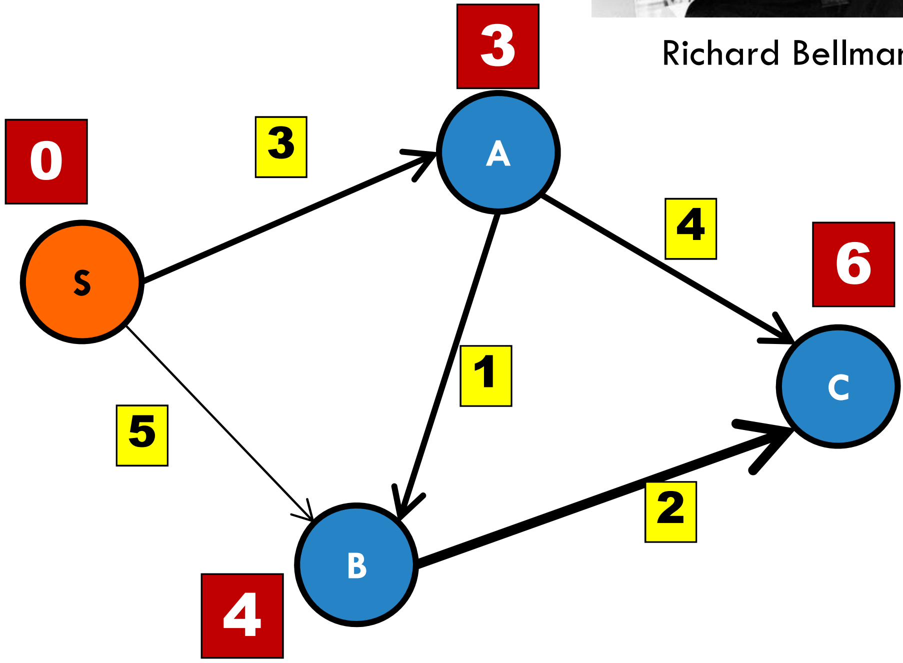


In fact, you can now compute shortest-paths from 1 source to **ALL** other nodes.

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



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	$O(V+E)$

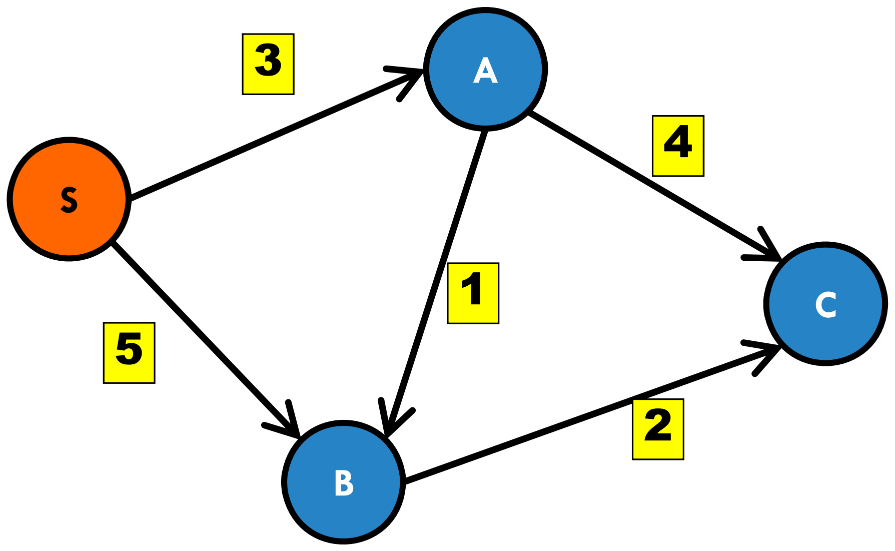
SINGLE-SOURCE SHORTEST PATHS

Input:

- Directed, weighted graph G = (V,E)
- Given source s and target t

Output:

- min-distance(s, t)



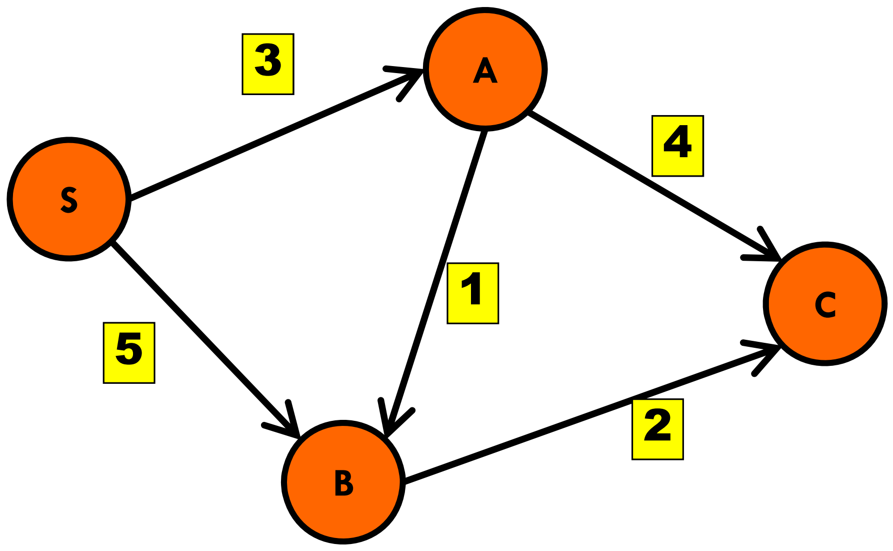
SINGLE-SOURCE SHORTEST PATHS

Input:

- Directed, weighted graph G = (V,E)
- Given source s and target t

Output:

- min-distance(s, t)



But what if we have queries involving many different source nodes?

ALL PAIRS SHORTEST PATHS (APSP) PROBLEM

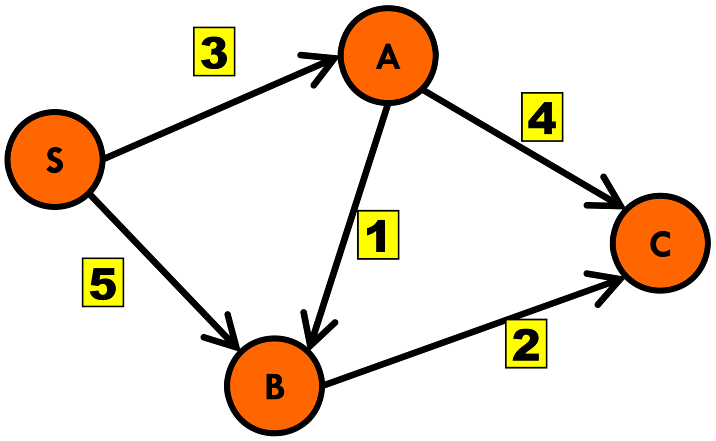
Compute distances between ALL pairs of nodes.

Simple solution (using what you have learnt)?

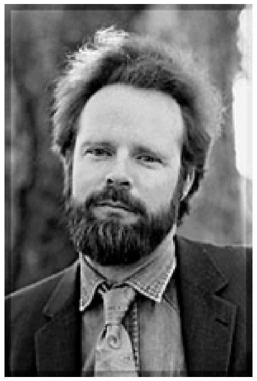
- Run Bellman-Ford / Dijkstra for all nodes

Running time?

- Bellman-Ford: $O(V^2 E)$
- Djikstra: $O((V^2 + VE) \log V)$



FLOYD-WARSHALL (1962)

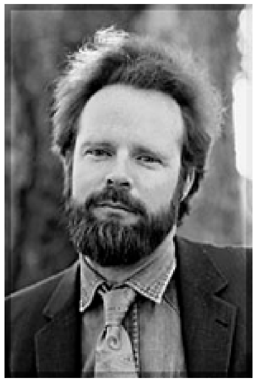


All Pairs Shortest Paths

Computes distances between ALL pairs of nodes.

Slide from the Lecture on Heaps!

REMEMBER ROBERT FLOYD?
FROM HEAPS: CLEVER CREATION IN $O(n)$ TIME



Invented by Robert Floyd in 1964

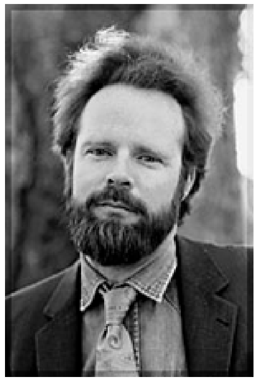
- invented invariants (among other things)
- we'll hear about him again in when we meet graphs!

The idea:

- View the input array as a binary tree
- "Bottom up" fixing of the tree to satisfy MaxHeap property

https://en.wikipedia.org/wiki/Robert_W._Floyd

FLOYD-WARSHALL (1962)



All Pairs Shortest Paths

Computes distances between ALL pairs of nodes.

The original algorithm was invented by Bernard Roy in 1959.

STIGLER'S LAW OF EPONYMY

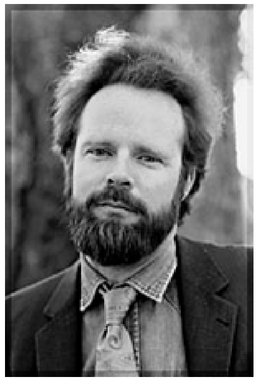
No scientific discovery is named after its original discoverer.

proposed by University of Chicago statistics professor Stephen Stigler.

Stigler named Columbia University sociology professor **Robert K. Merton** as the original discoverer of “Stigler's law”

Stigler’s law follows Stigler’s law.

FLOYD-WARSHALL (1962)



All Pairs Shortest Paths

Computes distances between ALL pairs of nodes.

Key Idea: Shortest paths have "optimal sub-structure":

If P is the shortest path ($u \rightarrow v \rightarrow w$), then P contains the shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

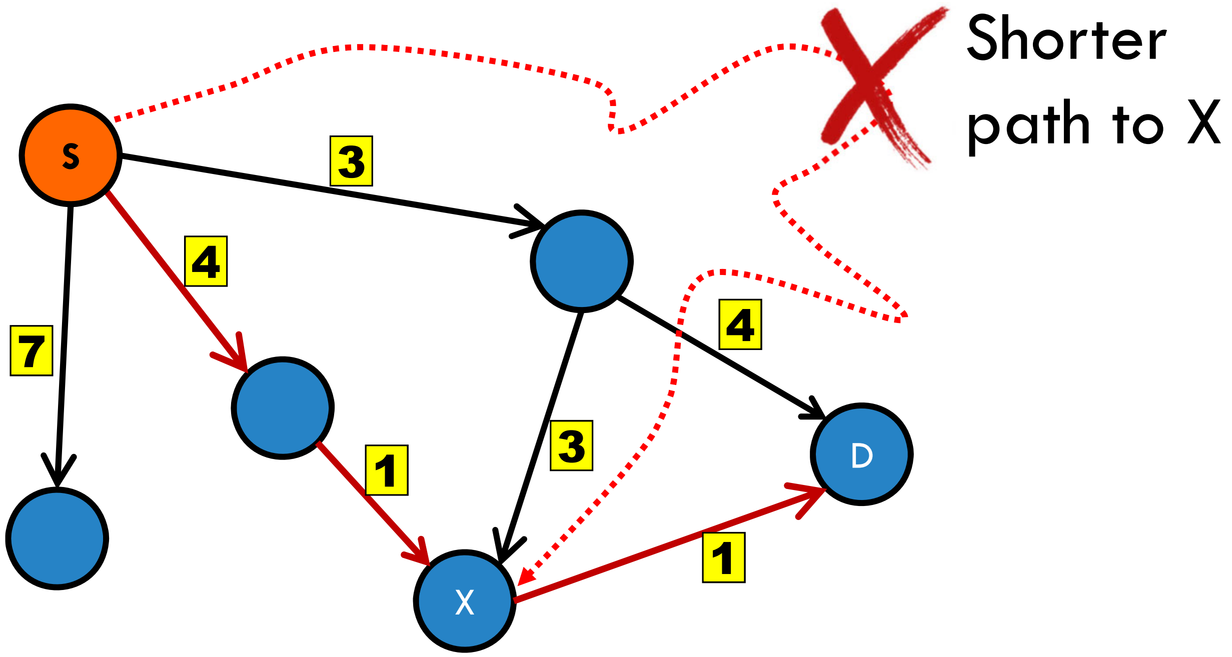
Many shortest path calculations depends on the same sub-pieces.

RECALL: SUBPATHS OF SHORTEST PATHS ARE SHORTEST PATHS

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

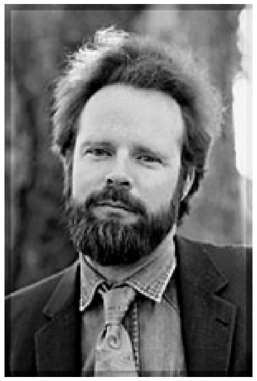
and if p goes through X,

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



FLOYD-WARSHALL (1962)

The original algorithm was invented by Bernard Roy in 1959.



All Pairs Shortest Paths

Computes distances between ALL pairs of nodes.

Key Idea: Shortest paths have "optimal sub-structure":

If P is the shortest path ($u \rightarrow v \rightarrow w$), then P contains the shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

Many shortest path calculations depends on the same sub-pieces.

Is a **Dynamic Programming** solution

DYNAMIC PROGRAMMING

Actually isn't really about "PROGRAMMING"

"Programming" refers to a "tabular method"

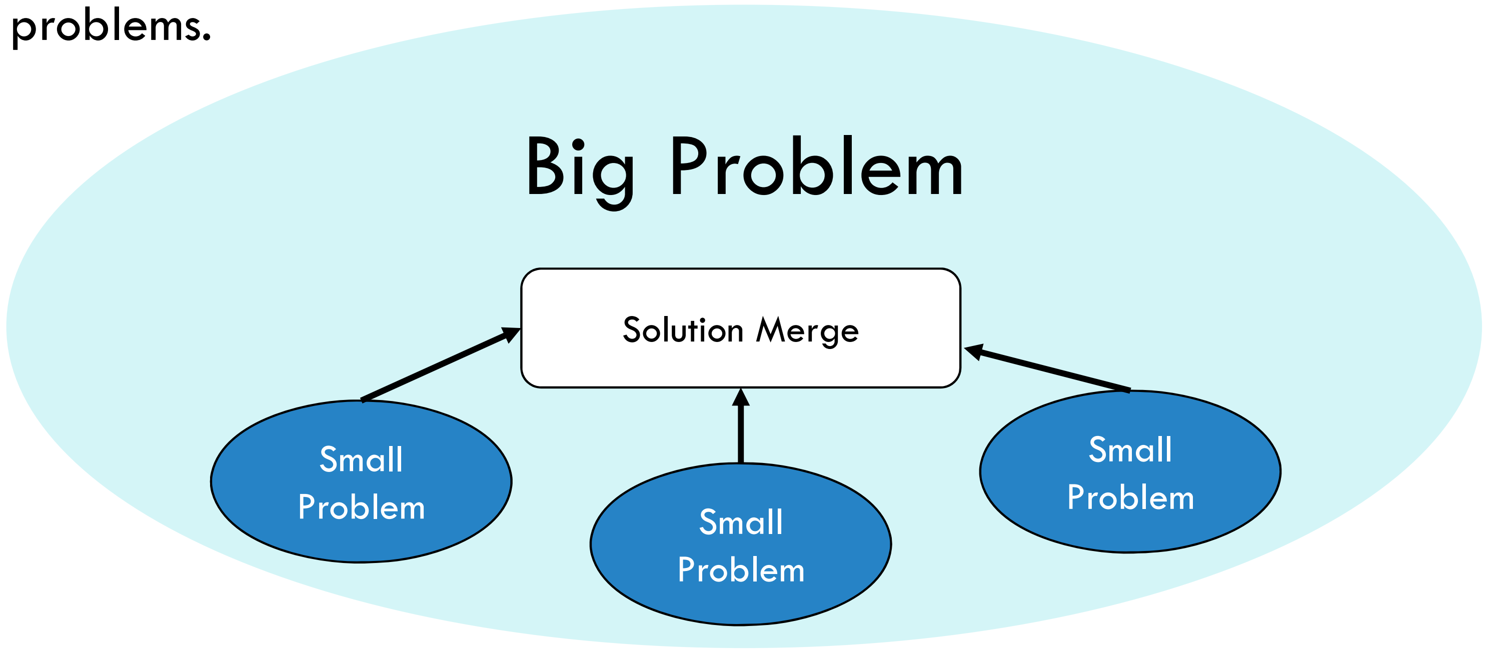
4 basic steps:

- Figure out the subproblems.
- Relate the subproblem solutions.
- Recurse and memoize ("memorize")
- Solve the original problem via subproblems.

DYNAMIC PROGRAMMING BASICS

Optimal sub-structure:

- Optimal solution can be constructed from optimal solutions to smaller sub-problems.



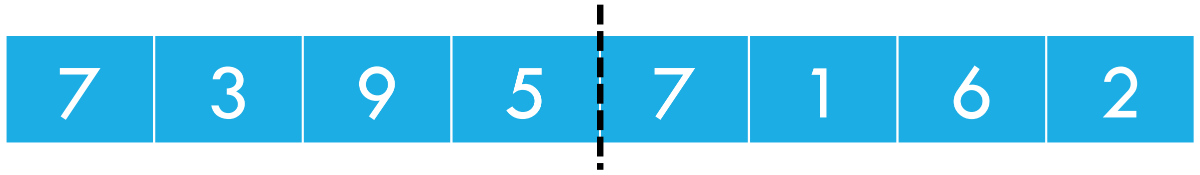
OPTIMAL SUB-STRUCTURE

Property of many problems we study:

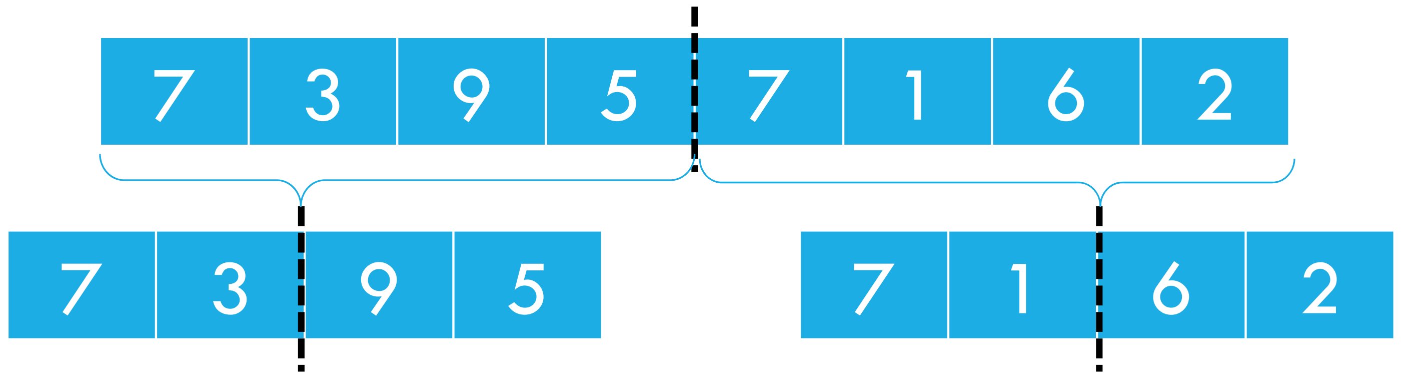
- **GREEDY** algorithms
  - Dijkstra's Algorithm
  - Minimum Spanning Tree algorithms

- **Divide-and-conquer** algorithms
  - MergeSort

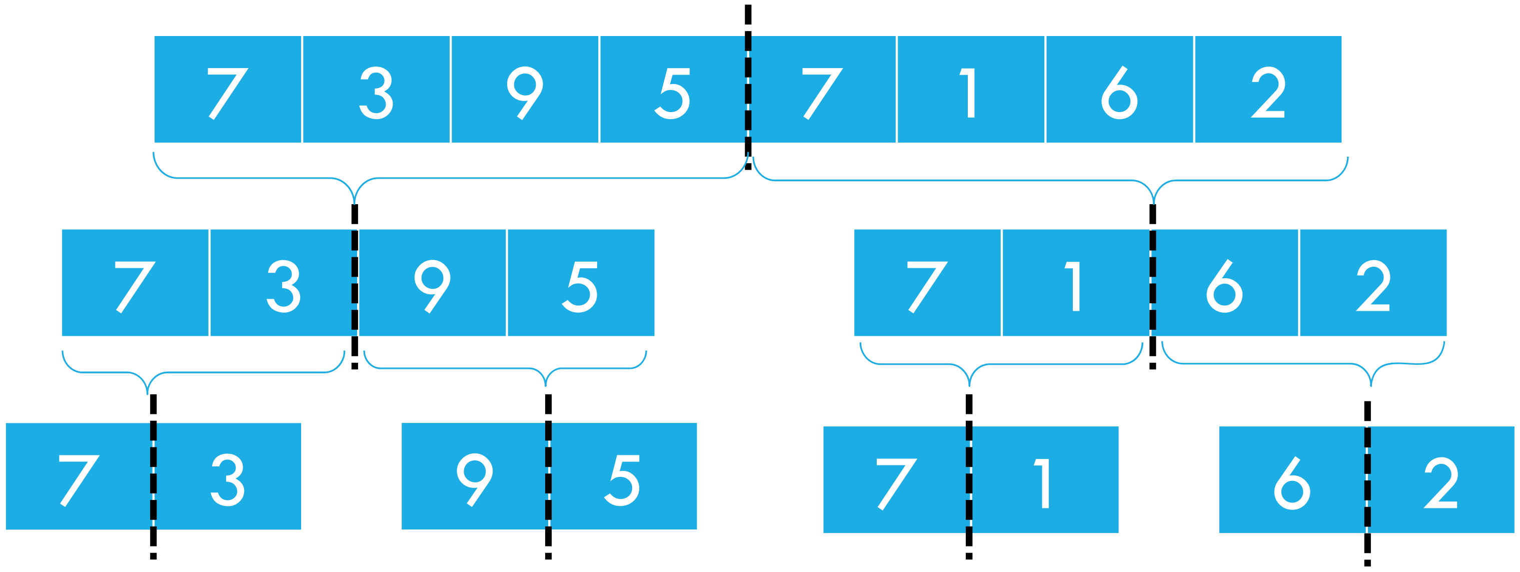
MERGESORT: RECURSE “DOWNWARDS”



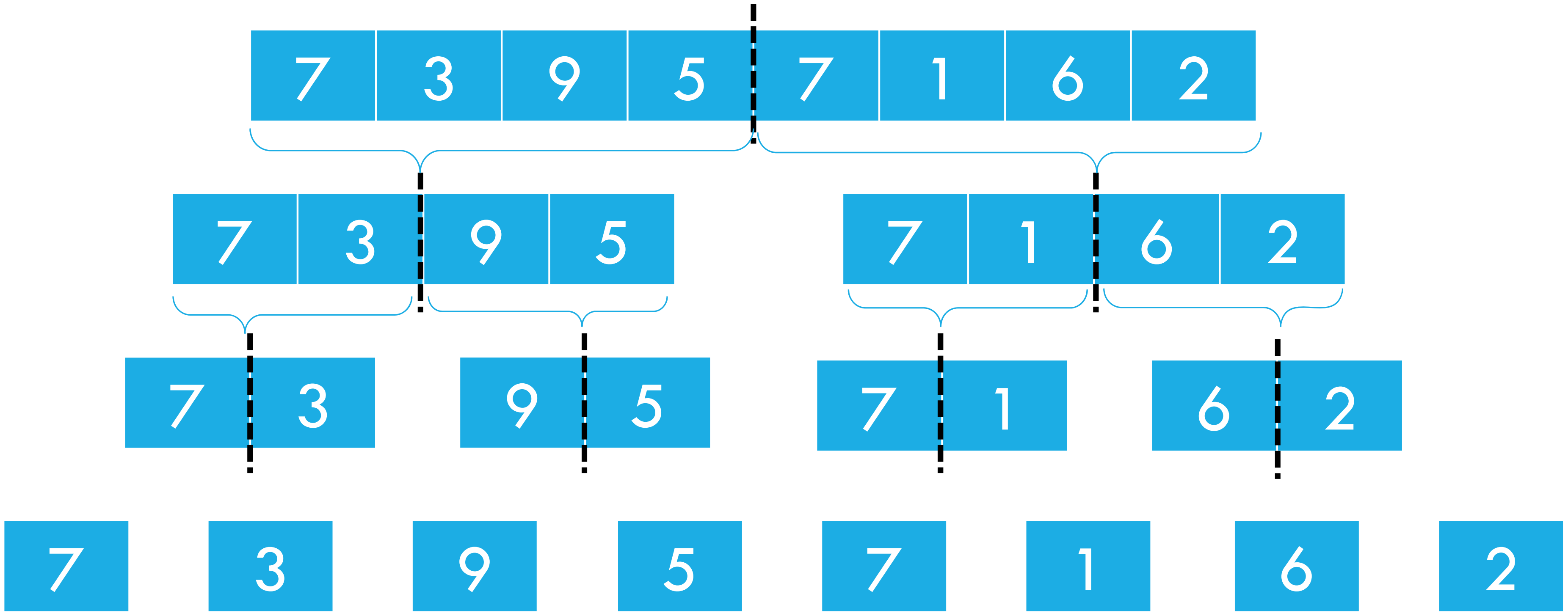
MERGESORT: RECURSE “DOWNWARDS”



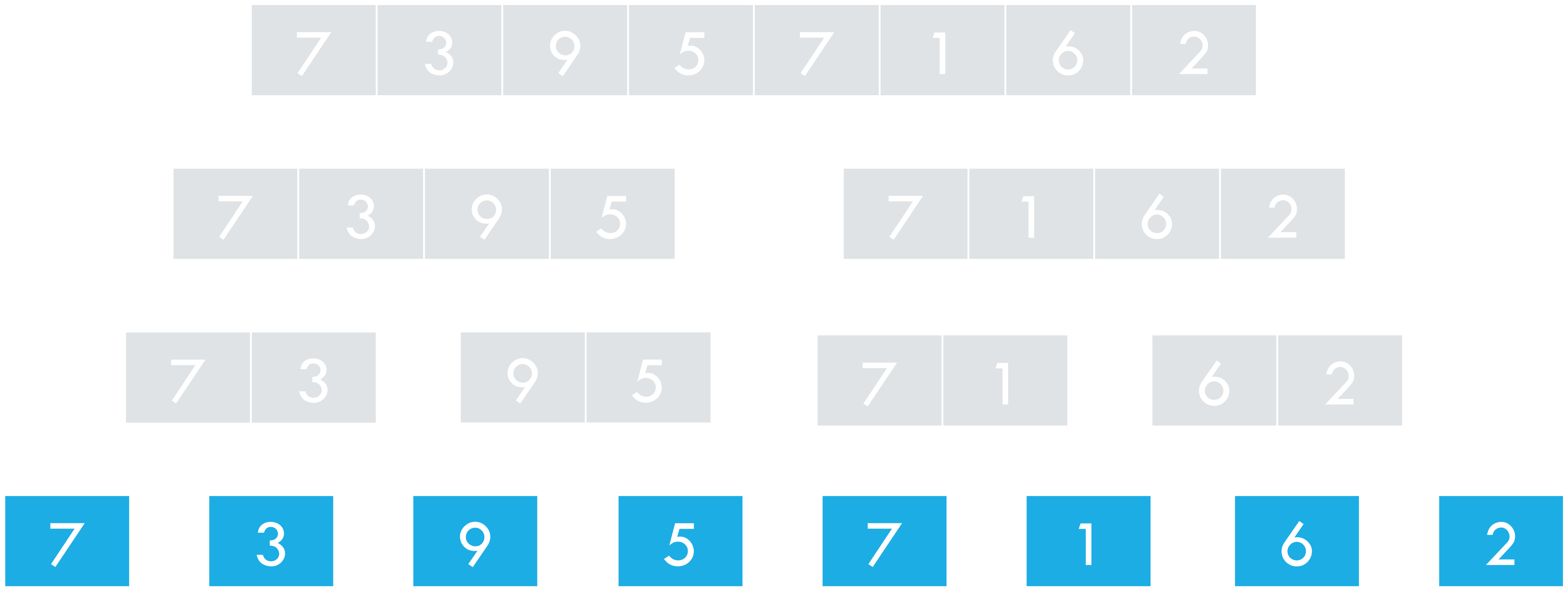
MERGESORT: RECURSE "DOWNWARDS"



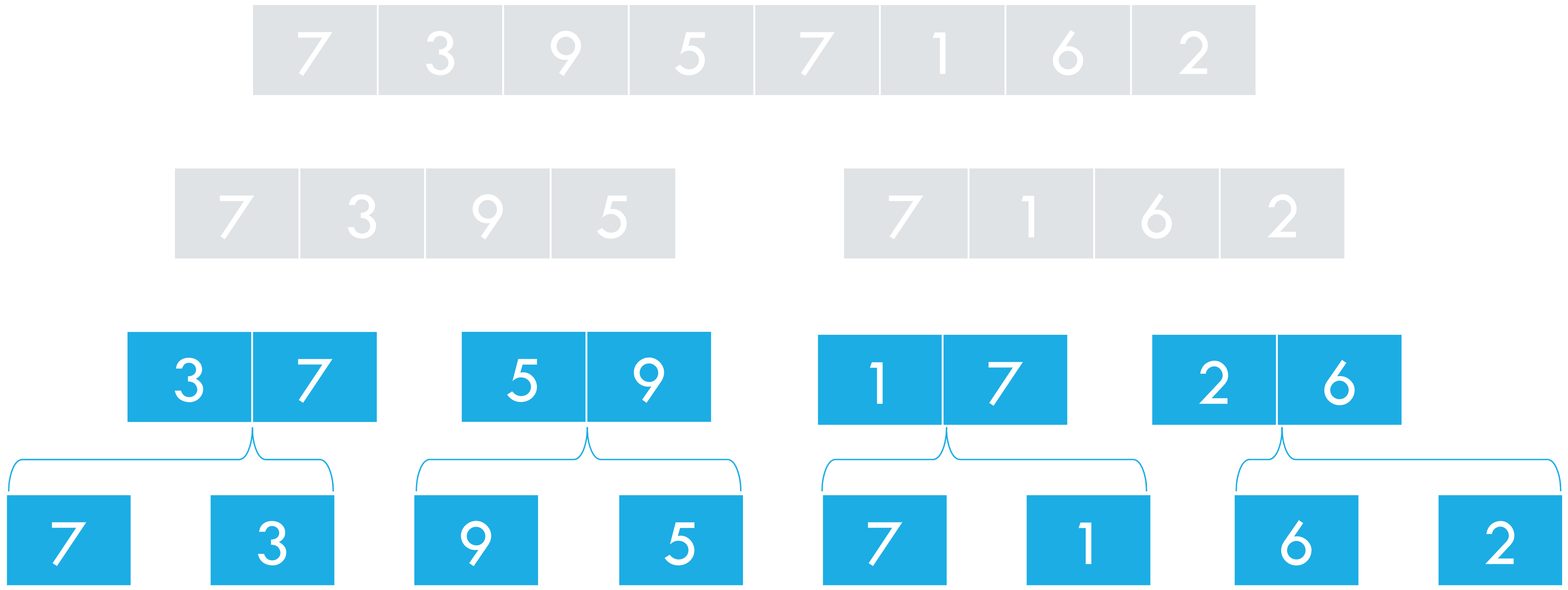
MERGESORT: RECURSE “DOWNWARDS”



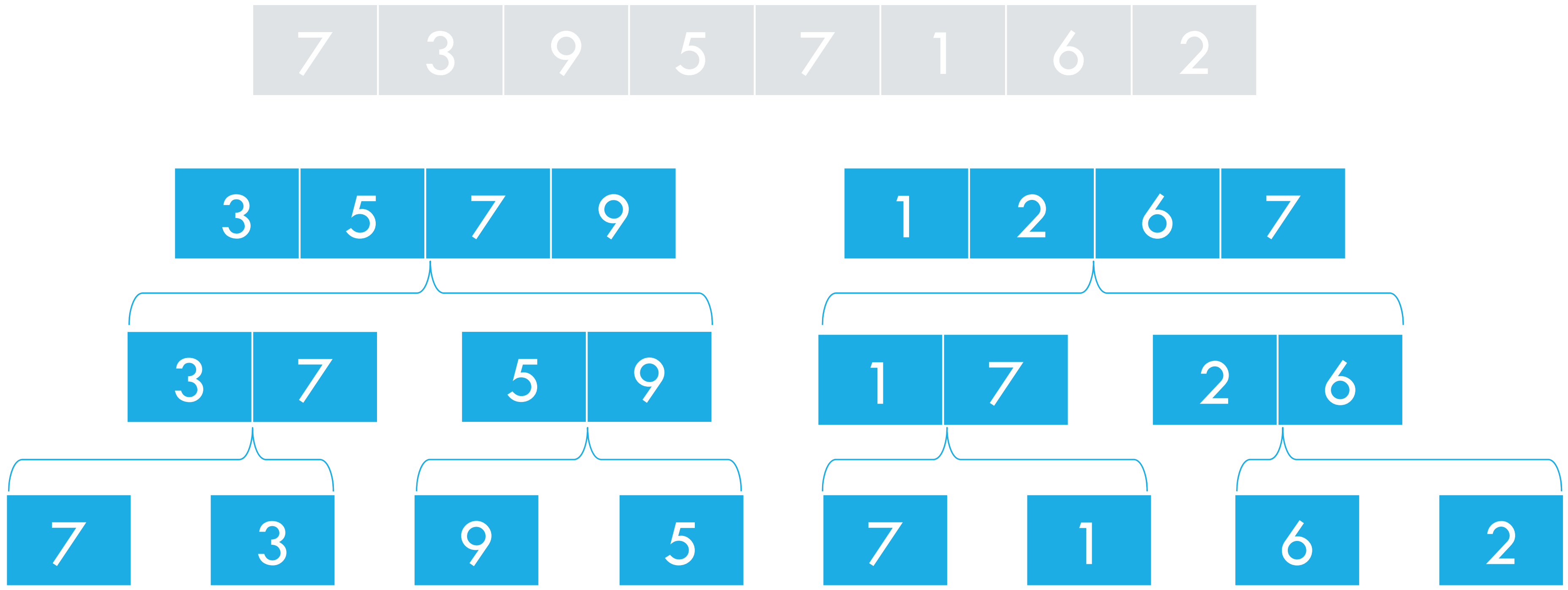
MERGESORT: MERGING “UPWARDS”



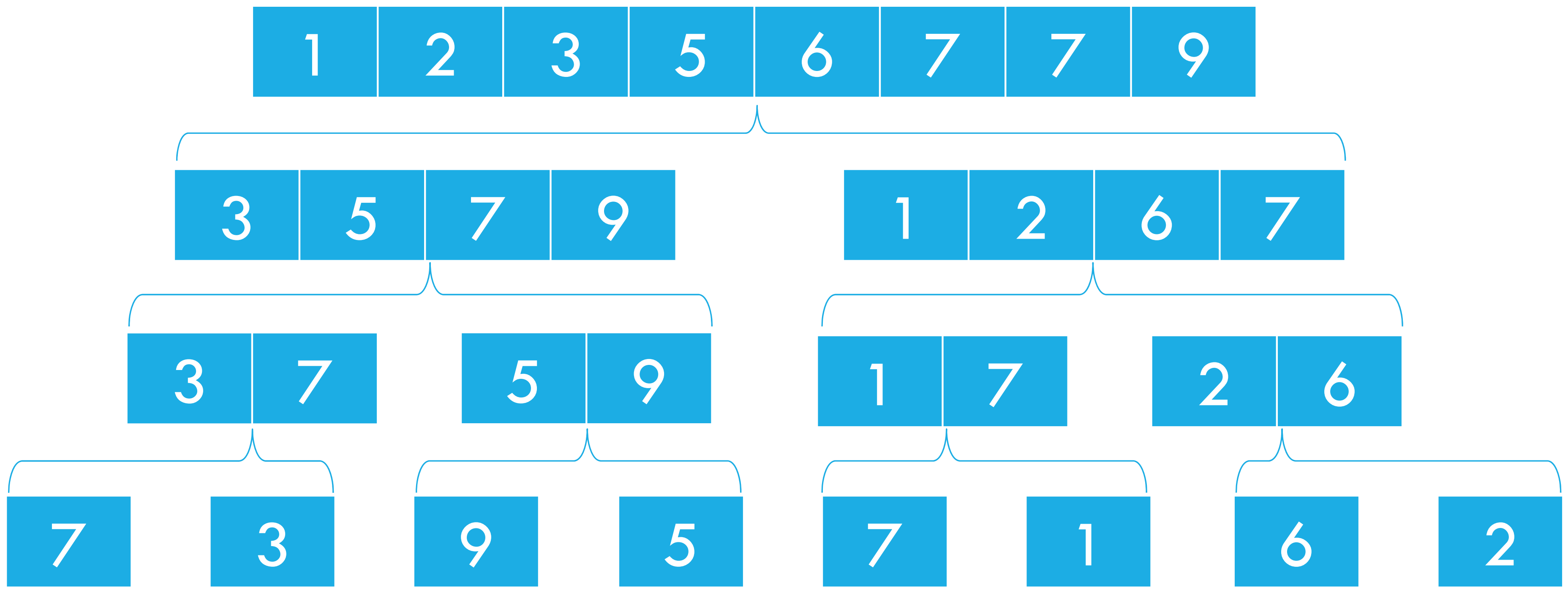
MERGESORT: MERGING “UPWARDS”



MERGESORT: MERGING “UPWARDS”



MERGESORT: MERGING “UPWARDS”



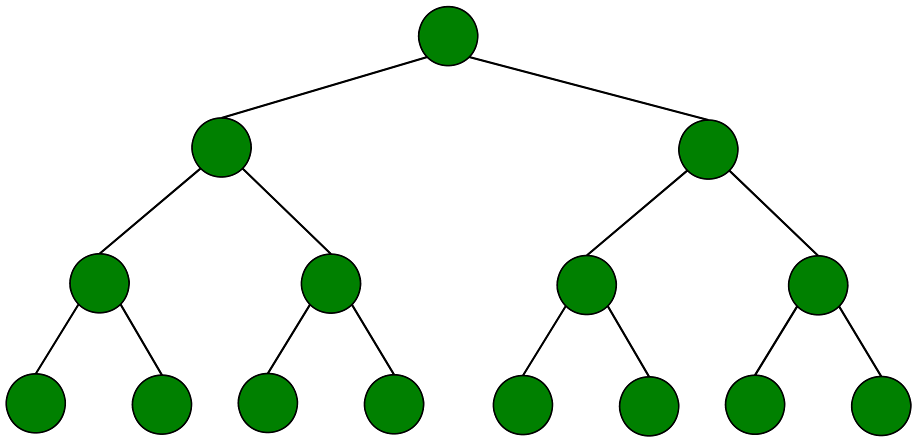
OPTIMAL SUB-STRUCTURE

Property of many problems we study:

- **GREEDY** algorithms
  - Dijkstra’s Algorithm
  - Minimum Spanning Tree algorithms

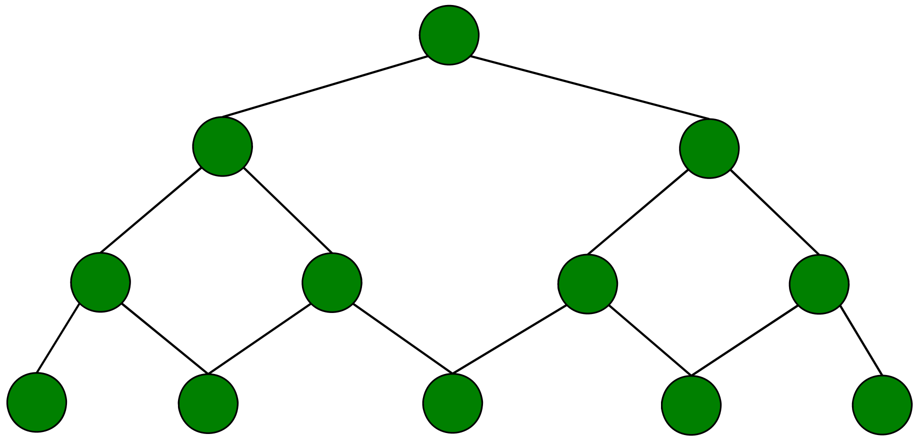
- **Divide-and-conquer** algorithms
  - MergeSort

OPTIMAL SUB-STRUCTURE



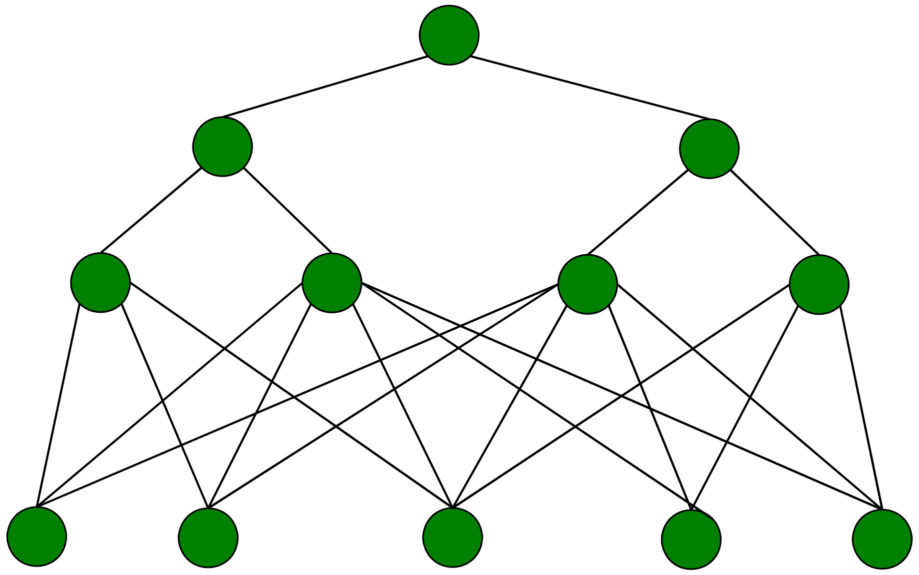
OVERLAPPING SUB-PROBLEMS

The same smaller problem is used to solve multiple different bigger problems.



OVERLAPPING SUB-PROBLEMS

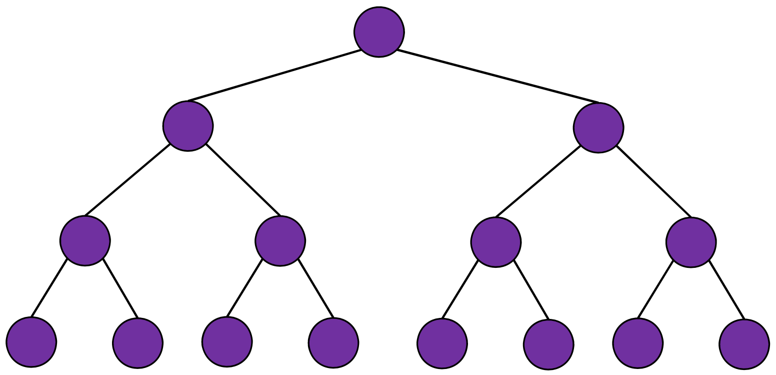
The same smaller problem is used to solve multiple different bigger problems.



Both have optimal substructure

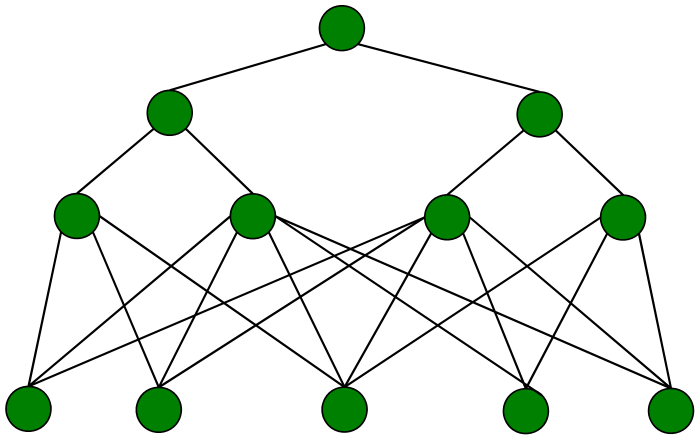
DYNAMIC PROGRAMMING

No overlapping subproblems



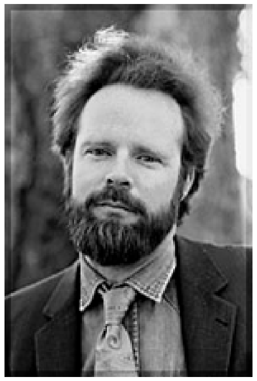
Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

FLOYD-WARSHALL (1962)



Dynamic programming:

Shortest paths have optimal sub-structure:

If P is the shortest path (u→v→w), then P contains the shortest path from (u→v) and from (v→w).

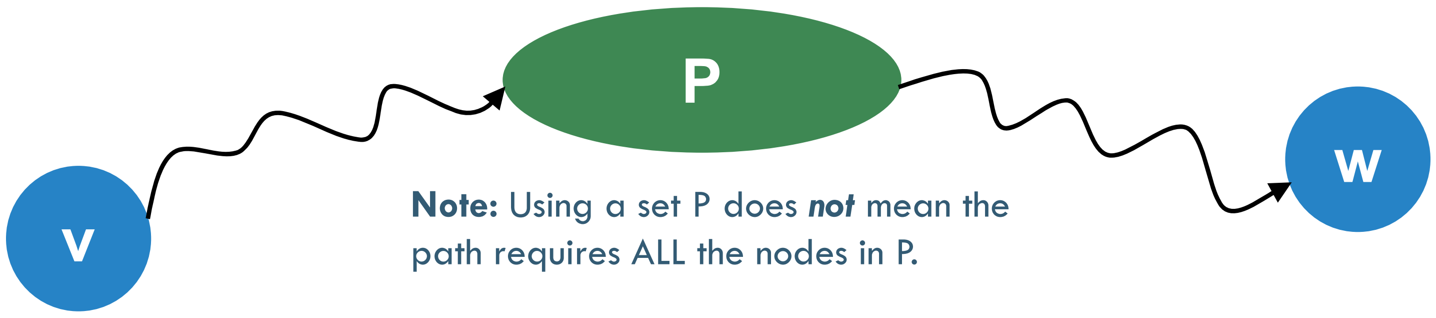
Shortest paths have overlapping subproblems

Many shortest path calculations depends on the same sub-pieces.

Hard question: what are the right subproblems?

FLOYD-WARSHALL

Let $S[v, w, P]$ be the distance (of the shortest path) from v to w that ***only uses*** intermediate nodes in the set P.



Note: Using a set P does ***not*** mean the path requires ALL the nodes in P.

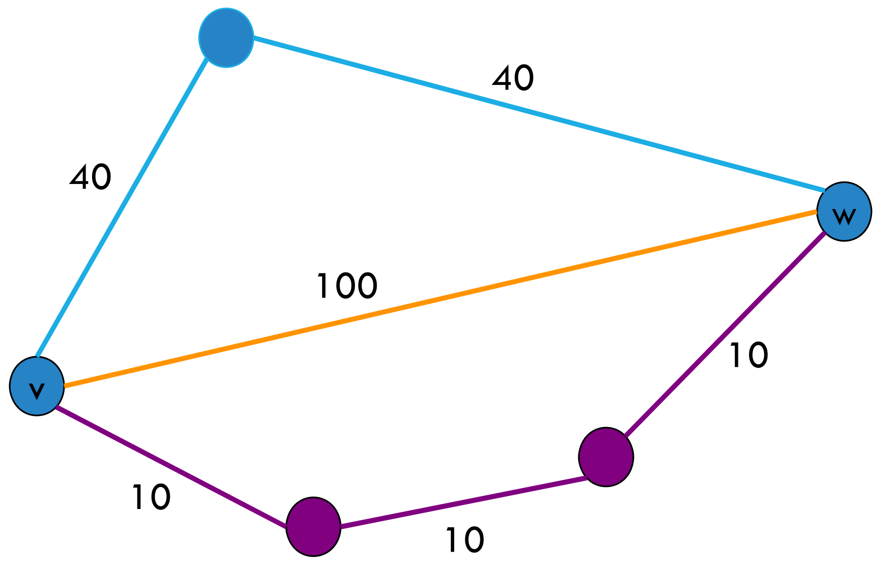
FLOYD-WARSHALL

Let $S[v, w, P]$ be the distance (of the shortest path) from v to w that only uses intermediate nodes in the set P.

P_1 = no nodes (empty set)

P_2 = blue nodes

P_3 = purple nodes



FLOYD-WARSHALL

Let $S[v, w, P]$ be the distance (of the shortest path) from v to w that only uses intermediate nodes in the set P.

P_1 = no nodes (empty set)

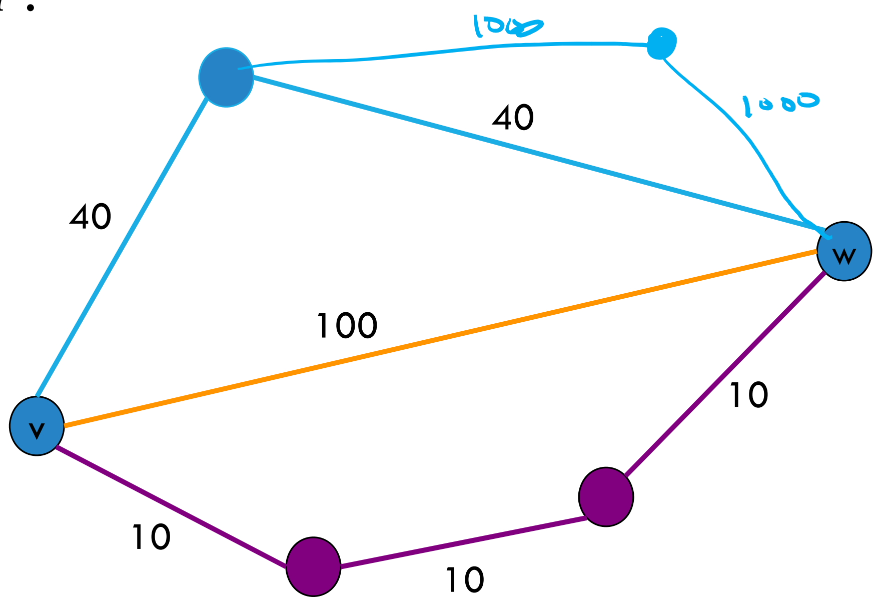
P_2 = blue nodes

P_3 = purple nodes

$S(v, w, P_1) = 100$

$S(v, w, P_2) = 80$

$S(v, w, P_3) = 30$



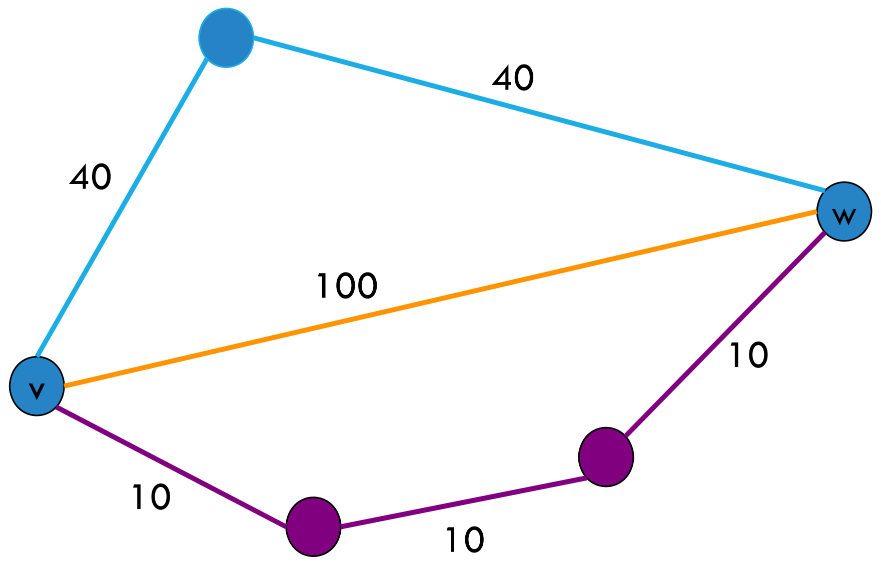
FLOYD-WARSHALL

Let $S[v, w, P]$ be the distance (of the shortest path) from v to w that only uses intermediate nodes in the set P.

Base case:

$S[v, w, \emptyset] = E[v, w]$

$E[v, w]$ = weight of edge from v to w.
(if no edge: $E[v, w] = \infty$)

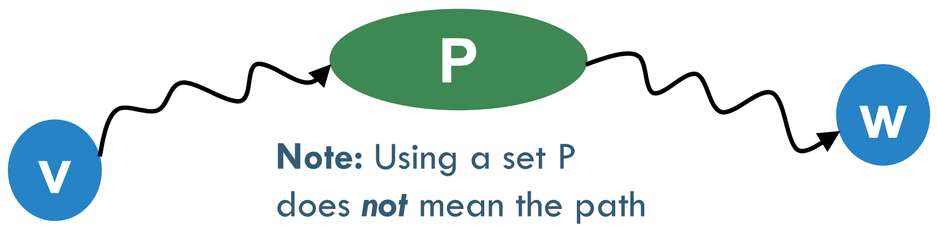


FLOYD-WARSHALL

Which sets P do we need to check for a graph with n nodes (nodes are labelled from $1, 2, \dots, n$)?

Check increasingly large sets:

$P_0 = \varnothing$
$P_1 = \{1\}$
$P_2 = \{1, 2\}$
$P_3 = \{1, 2, 3\}$
$P_4 = \{1, 2, 3, 4\}$
…
$P_n = \{1, 2, 3, 4, \dots, n\}$

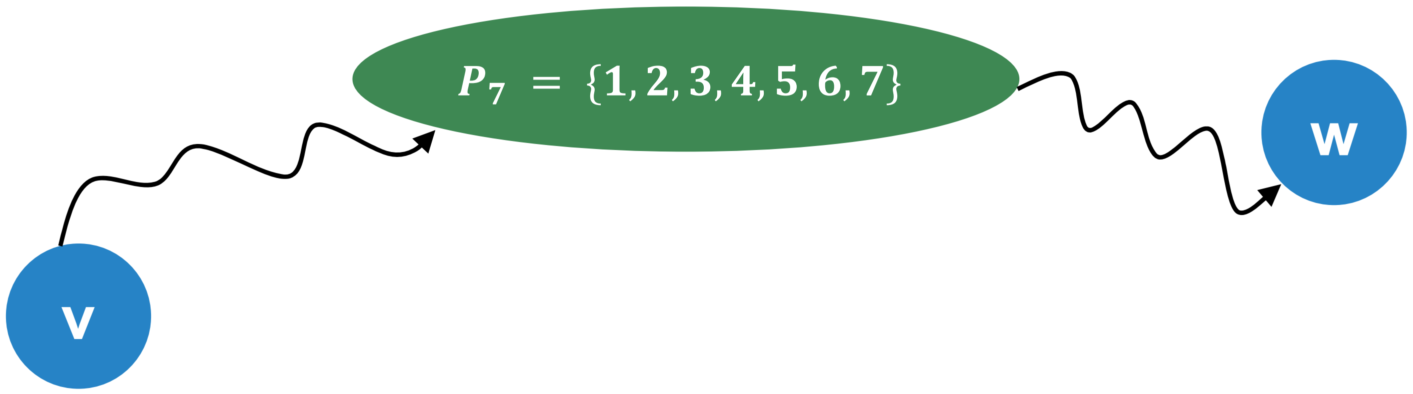


Note: Using a set P does ***not*** mean the path requires ALL the nodes in P.

FLOYD-WARSHALL

Use the **precalculated** subproblems:

Assume we have calculated $S[v, w, P_7] = 42$.
How do we calculate $S[v, w, P_8]$?



Slide from the previous lecture!

REMEMBER RELAX?

```
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:

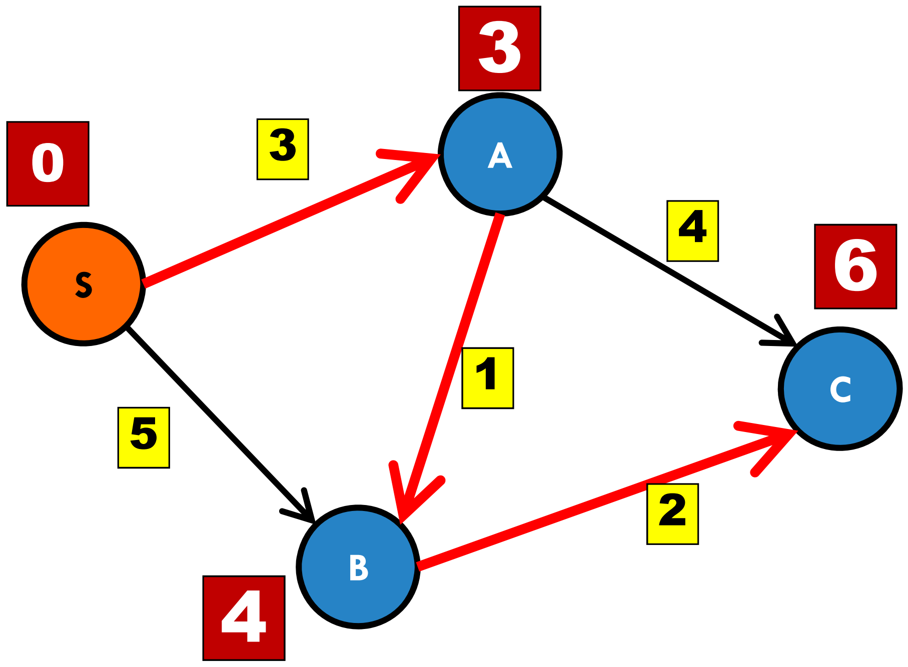
relax(w,v):

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

If yes:

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

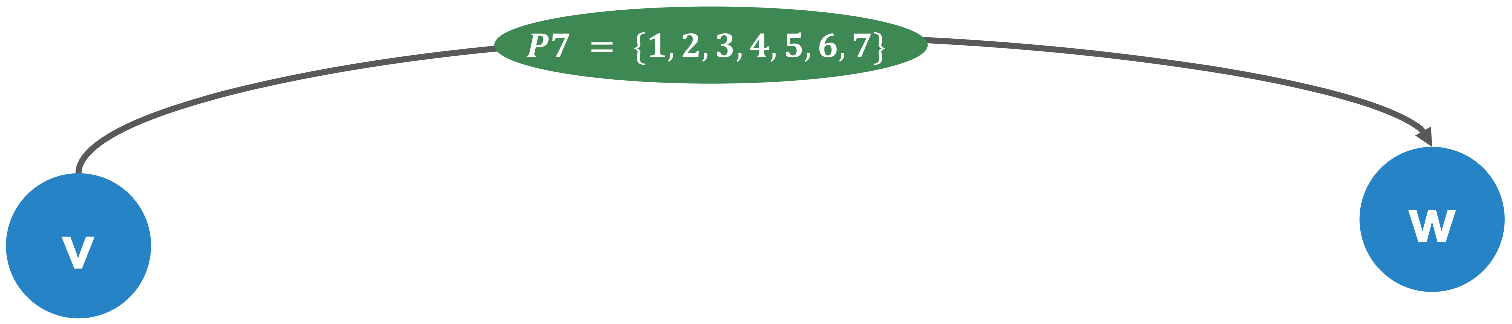
This creates a **predecessor subgraph**



FLOYD-WARSHALL

Use the **precalculated** subproblems:

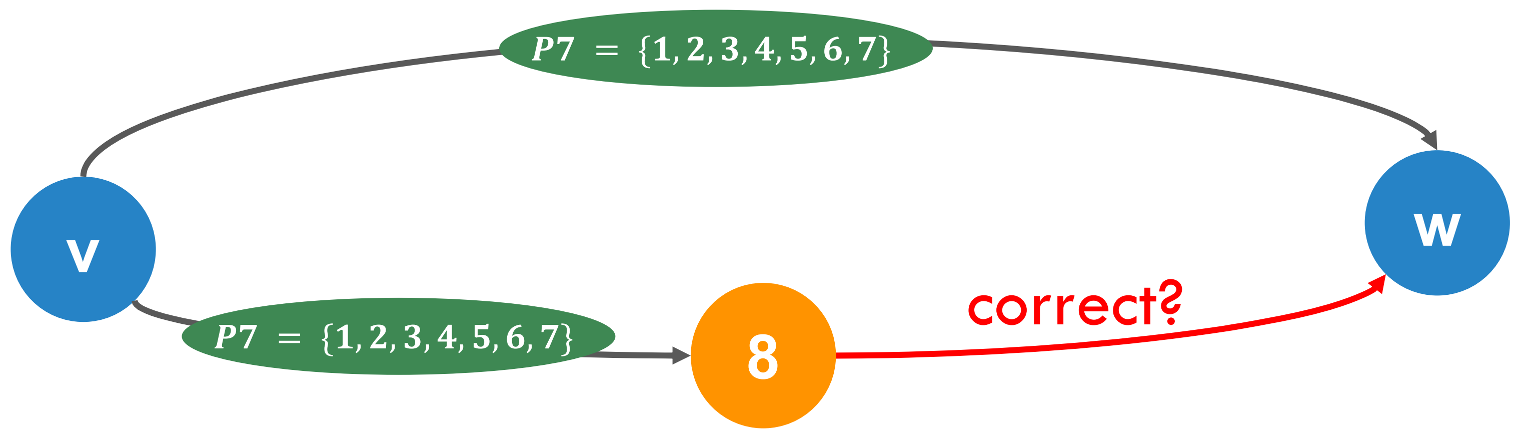
$$S[v, w, P_8] = \min(\ S[v, w, P_7], \qquad ?\,)$$



FLOYD-WARSHALL

Use the **precalculated** subproblems:

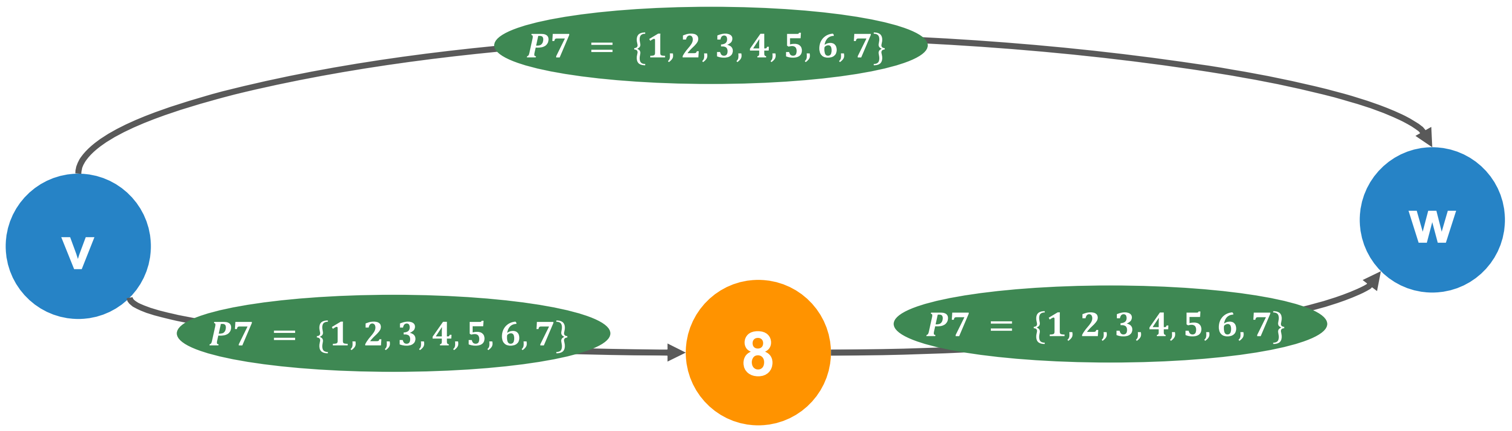
$$S[v,w,P_8] = \min(\ S[v,w,P_7],\qquad S[v,8,P_7] + E[8,w]\ )$$



FLOYD-WARSHALL

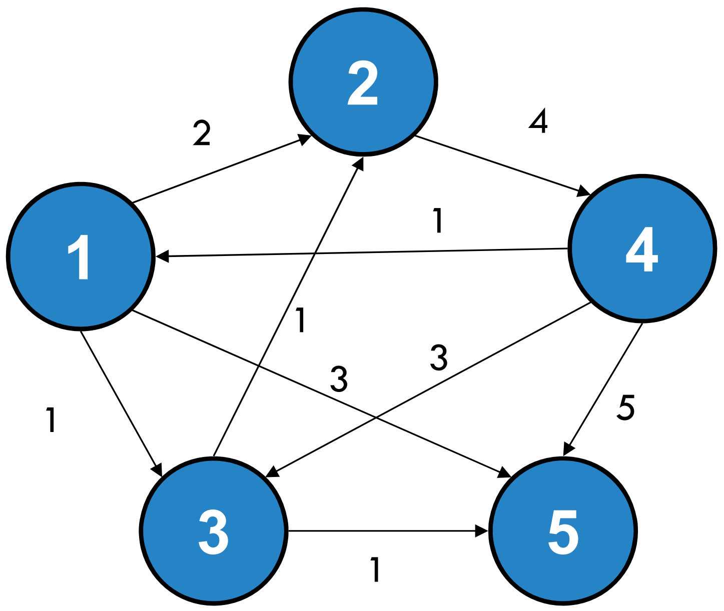
Use the **precalculated** subproblems:

$$S[v, w, P_8] = \min(\ S[v, w, P_7], \qquad S[v, 8, P_7] + S[8, w, P_7]\ )$$



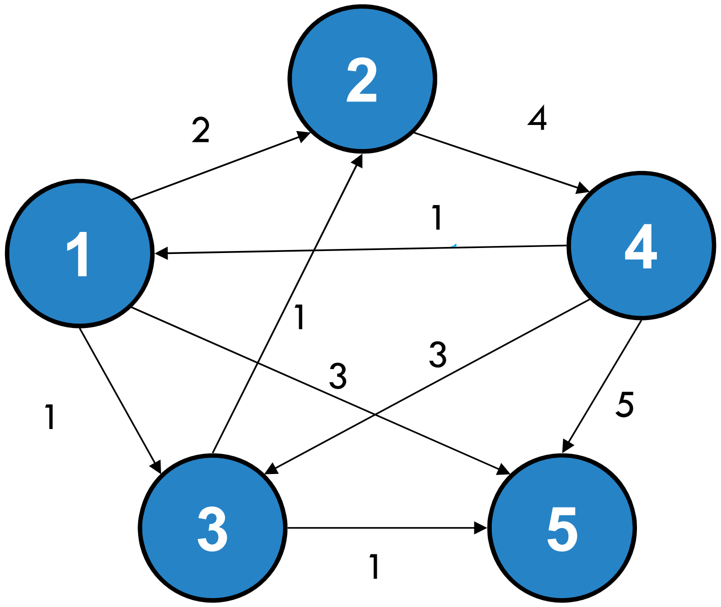
$P7 = \{1, 2, 3, 4, 5, 6, 7\}$
v
w
$P7 = \{1, 2, 3, 4, 5, 6, 7\}$
8
$P7 = \{1, 2, 3, 4, 5, 6, 7\}$

EXAMPLE



INITIALIZATION

$$S[v, w, P_0] \ = \underline{E[v, w]}$$



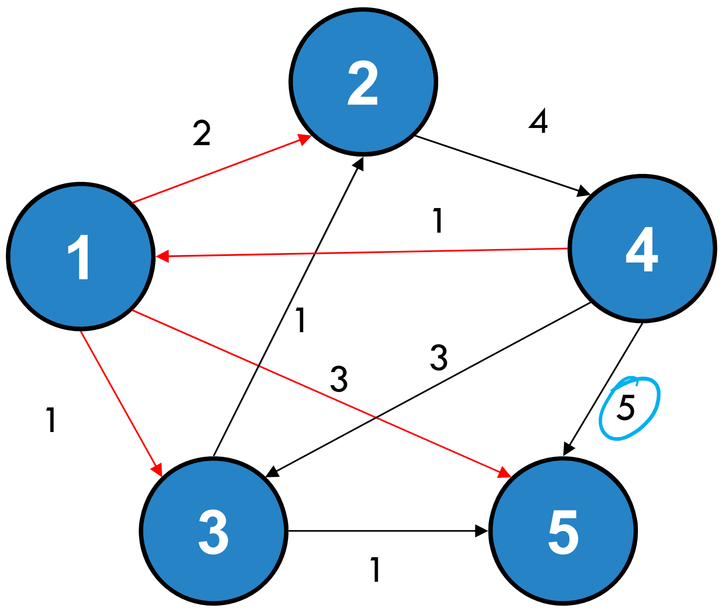
	1	2	3	4	5
1	0	2	1	∞	3
2	∞	0	∞	4	∞
3	∞	1	0	∞	1
4	1	∞	3	0	5
5	∞	∞	∞	∞	0

STEP: $P_1 = \{1\}$

5 1 + 3 = 4

$S[v,w,P_1] = \min(S[v,w,P_0], S[v,1,P_0] + S[1,w,P_0])$

$P_0 = \{\}$



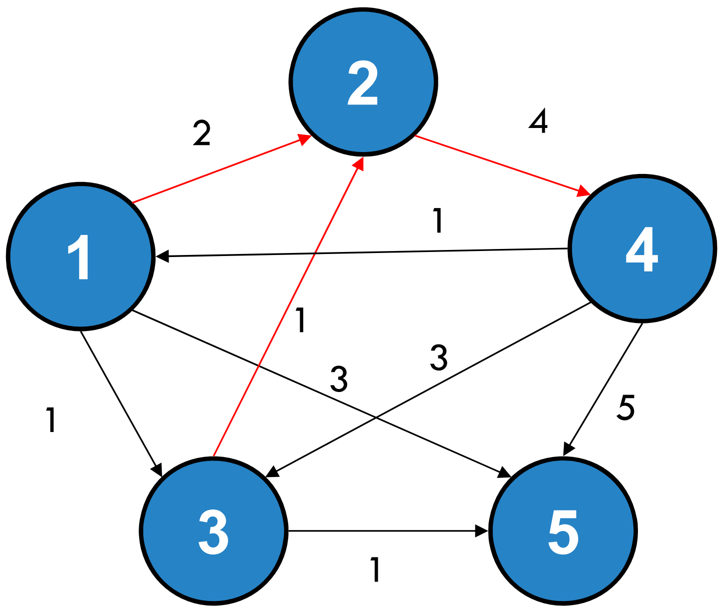
	1	2	3	4	5
1	0	2	1	∞	3
2	∞	0	∞	4	∞
3	∞	1	0	∞	1
4	1	∞	3	0	5
5	∞	∞	∞	∞	0

	1	2	3	4	5
1	0	2	1	∞	3
2	∞	0	∞	4	∞
3	∞	1	0	∞	1
4	1	**3**	**2**	0	**4**
5	∞	∞	∞	∞	0

STEP: $P_2 = \{1,2\}$

$S[v,w,P_2] = \min(S[v,w,P_1], S[v,2,P_1] + S[2,w,P_1])$

$P_1 = \{1\}$



$S[v,w,P_1] =$

	1	2	3	4	5
1	0	2	1	∞	3
2	∞	0	∞	4	∞
3	∞	1	0	∞	1
4	1	3	2	0	4
5	∞	∞	∞	∞	0

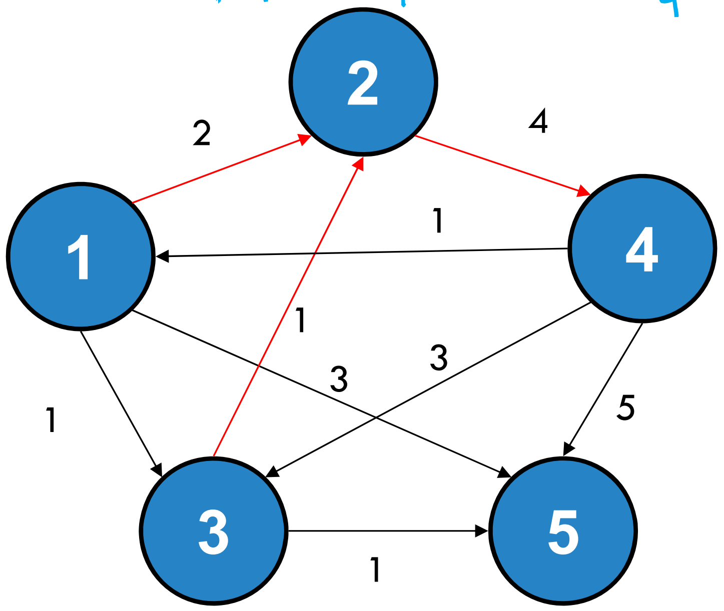
$S[v,w,P_2] =$

	1	2	3	4	5
1	0	2	~~?~~ 1	~~?~~ 6	3
2	∞	0	∞	4	∞
3	∞	1	0	?	1
4	1	3	2	0	4
5	∞	∞	∞	∞	0

STEP: $P_2 = \{1,2\}$

$$S[\underset{1}{v},\underset{4}{w},P_2] = \min(S[\underset{1}{v},\overset{\infty}{\underset{4}{w}},P_1], S[\underset{1}{v},\overset{2}{2},P_1] \overset{+}{+} S[2,\overset{4}{\underset{4}{w}},P_1]\overset{=6}{)}$$

$P_1 = \{1\}$



$S[v,w,P_1]$

	1	2	3	4	5
1	0	2	1	∞	3
2	∞	0	∞	4	∞
3	∞	1	0	∞	1
4	1	3	2	0	4
5	∞	∞	∞	∞	0

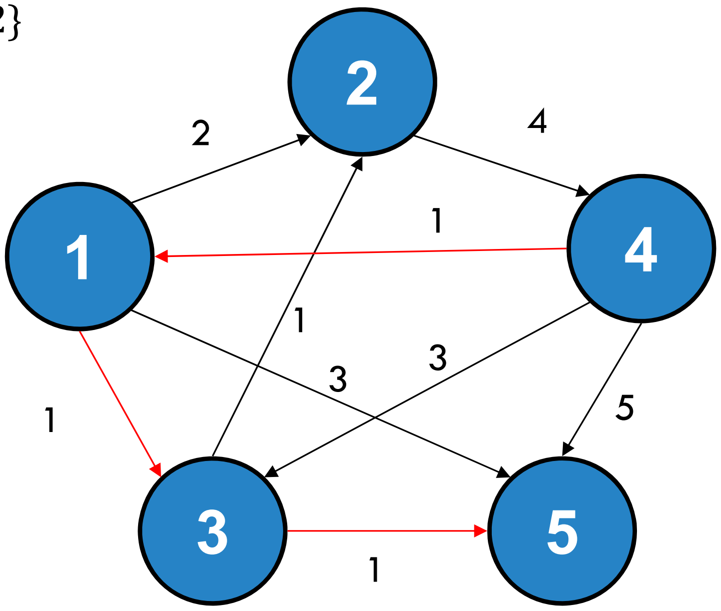
⇓

	1	2	3	4	5
1	0	2	1	6	3
2	∞	0	∞	4	∞
3	∞	1	0	5	1
4	1	3	2	0	4
5	∞	∞	∞	∞	0

STEP: $P_3 = \{1,2,3\}$

$S[v,w,P_3] = \min(S[v,w,P_2], S[v,3,P_2] + S[3,w,P_2])$

$P_2 = \{1,2\}$



	1	2	3	4	5
1	0	2	1	6	3
2	∞	0	∞	4	∞
3	∞	1	0	5	1
4	1	3	2	0	4
5	∞	∞	∞	∞	0

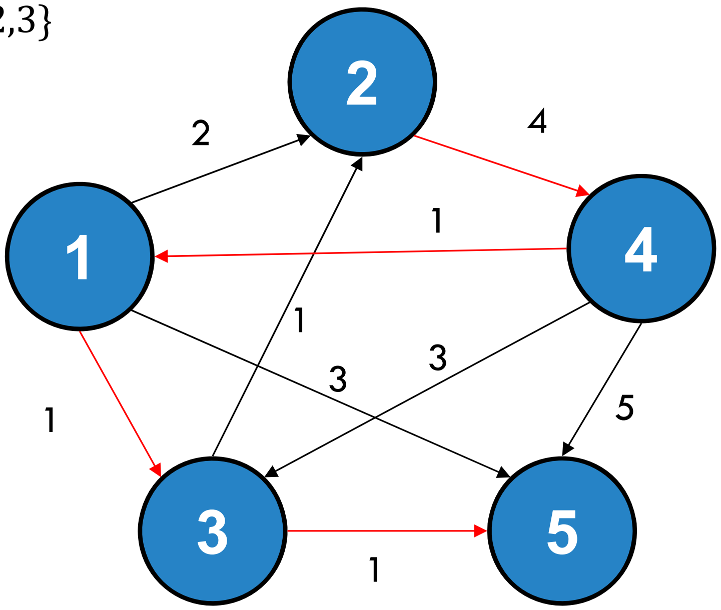
$\Downarrow$

	1	2	3	4	5
1	0	2	1	6	**2**
2	∞	0	∞	4	∞
3	∞	1	0	5	1
4	1	3	2	0	**3**
5	∞	∞	∞	∞	0

STEP: $P_4 = \{1,2,3,4\}$

$S[v,w,P_4] = \min(S[v,w,P_3], S[v,4,P_3] + S[4,w,P_3])$

$P_3 = \{1,2,3\}$



	1	2	3	4	5
1	0	2	1	6	2
2	∞	0	∞	4	∞
3	∞	1	0	5	1
4	1	3	2	0	3
5	∞	∞	∞	∞	0

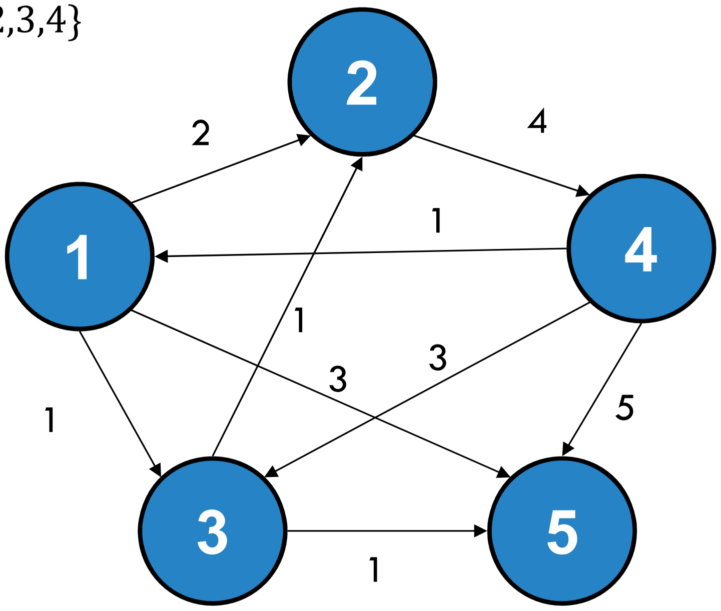
$\Downarrow$

	1	2	3	4	5
1	0	2	1	6	2
2	5	0	6	4	7
3	6	1	0	5	1
4	1	3	2	0	3
5	∞	∞	∞	∞	0

STEP: $P_5 = \{1,2,3,4,5\}$

$S[v,w,P_5] = \min(S[v,w,P_4], S[v,5,P_4] + S[5,w,P_4])$

$P_4 = \{1,2,3,4\}$

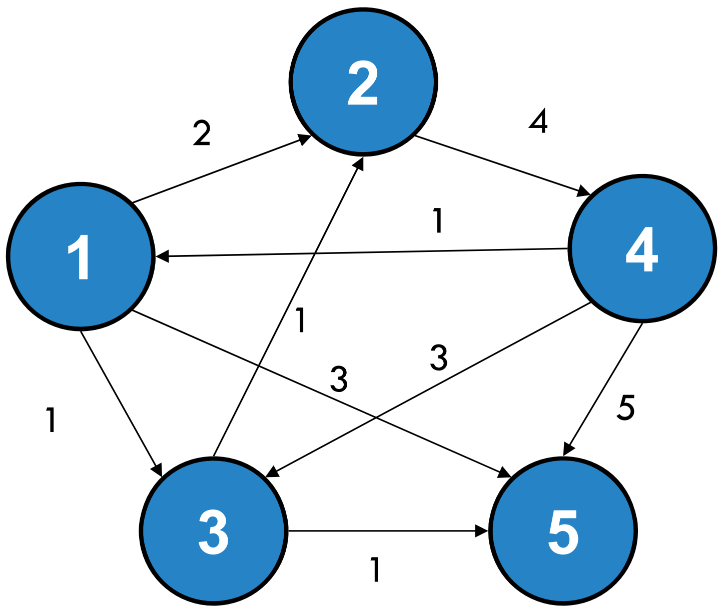


	1	2	3	4	5
1	0	2	1	6	2
2	5	0	6	4	7
3	6	1	0	5	1
4	1	3	2	0	3
5	∞	∞	∞	∞	0

$\Downarrow$

	1	2	3	4	5
1	0	2	1	6	2
2	5	0	6	4	7
3	6	1	0	5	1
4	1	3	2	0	3
5	∞	∞	∞	∞	0

DONE! ☺

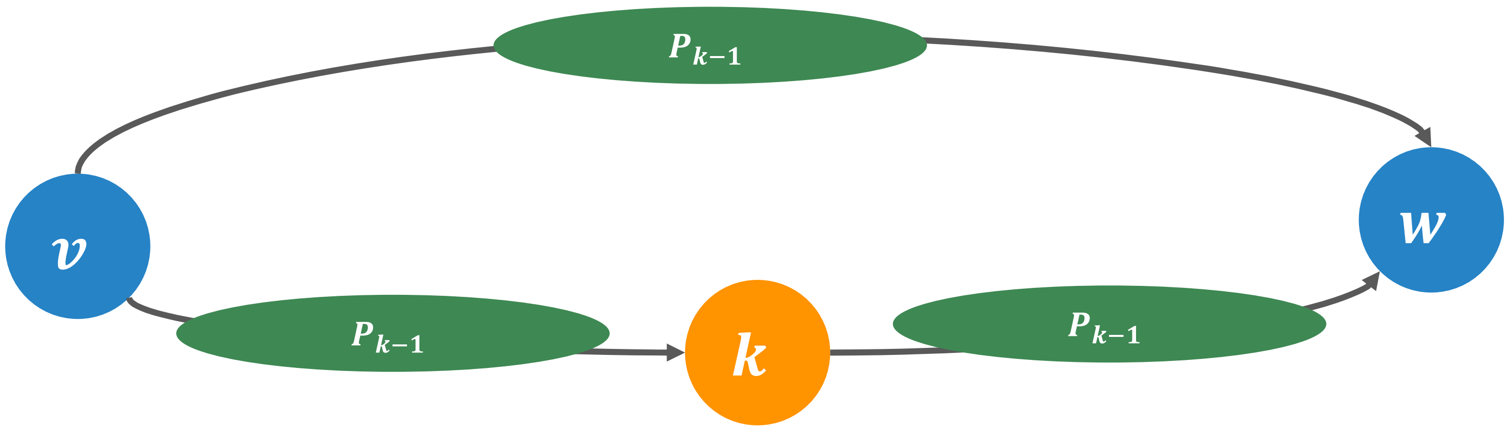


	1	2	3	4	5
1	0	2	1	6	2
2	5	0	6	4	7
3	6	1	0	5	1
4	1	3	2	0	3
5	∞	∞	∞	∞	0

FLOYD-WARSHALL

Use the **precalculated** subproblems:

$$S[v,w,P_k] = \min(\ S[v,w,P_{k-1}], \qquad S[v,k,P_{k-1}] + S[k,w,P_{k-1}]\,)$$



FLOYD-WARSHALL: PSEUDOCODE

```
Function FloydWarshall(G)
    S = Array of size |V|x|V| //memoization table S has |V| rows and |V| columns

    // Initialize every pair of nodes
    for v = 0 to |V|-1                  |
        for w = 0 to |V|-1              |  O(V^2)
            S[v,w] = E[v,w]             |
                                                  +
    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for k = 0 to |V|-1   - ✓                            }
        for v = 0 to |V|-1                              }  O(V^3)
            for w = 0 to |V|-1                          }
                S[v,w] = min(S[v,w], S[v,k]+S[k,w])     }
    return S
```

What is the running time? $O\left(V^3\right)$

TODAY: LEARNING OUTCOMES

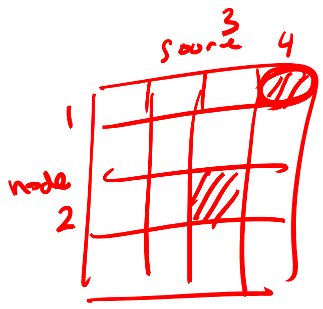
By the end of this session, students should be able to:

- State the **all-pairs shortest-paths** (APSP) problem
- Explain **Floyd-Warshall** and apply it to solve the APSP problem.
- Analyze the **computational complexity of Floyd-Warshall algorithm.**

Poll Everywhere

https://bit.ly/2LvG9bq

QUESTIONS?



QUIZ 3

Will cover everything up to SSSP

No Minimum Spanning Trees (MSTs)

Focus on topics not in Quiz 1 or 2.

- Hashing
- Graph Searching
- Single-source Shortest Paths

ADVICE:

Understanding* + *Analyzing the problem is usually 50-90% of the battle.

Read the question **properly**. If you are unsure, **ask.**

The solution may ***not*** be obvious at first.

It may take ***some hard thinking*** to “see” it.

The ***best solution*** is ***often not obvious*** at first.

HASH TABLES

Harold Soh
harold@comp.nus.edu.sg

HASH TABLES

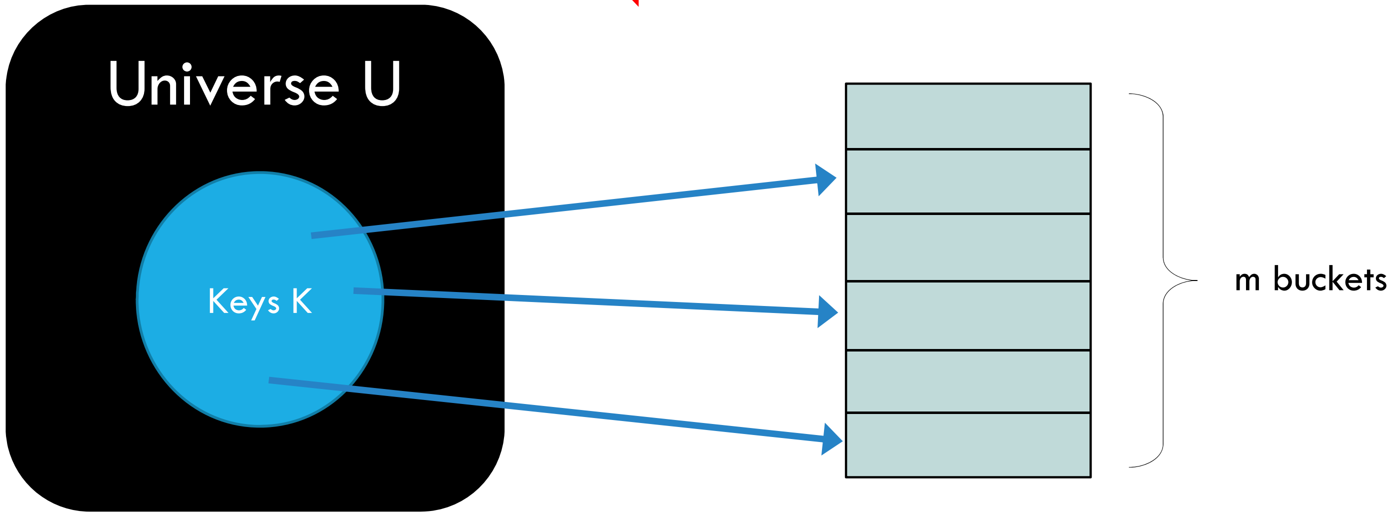
no free lunch!

Data Structure	Avg. Insert Time	Avg. Search Time	Avg. Max/Min	Avg. Floor/ Ceiling
Unordered Array / Linked List	O(1)	O(n)	O(n)	O(n)
Ordered Array / Linked List	O(n)	O(1)	O(1)	O(log n)
Balanced Binary Search Tree (AVL)	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table	**O(1)**	**O(1)**	**O(n)**	**O(n)**

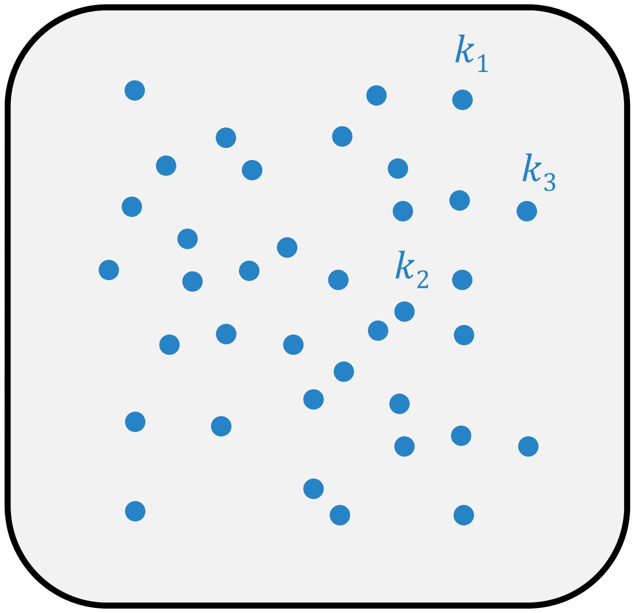
HASH FUNCTIONS

Define a hash function $h: U \to \{0, \dots, m-1\}$

- Store key k in bucket $h(k)$
- Time complexity: Time to compute h + Time to access bucket
- Assume: computing h takes $O(1)$ **This may not be true in practice!**



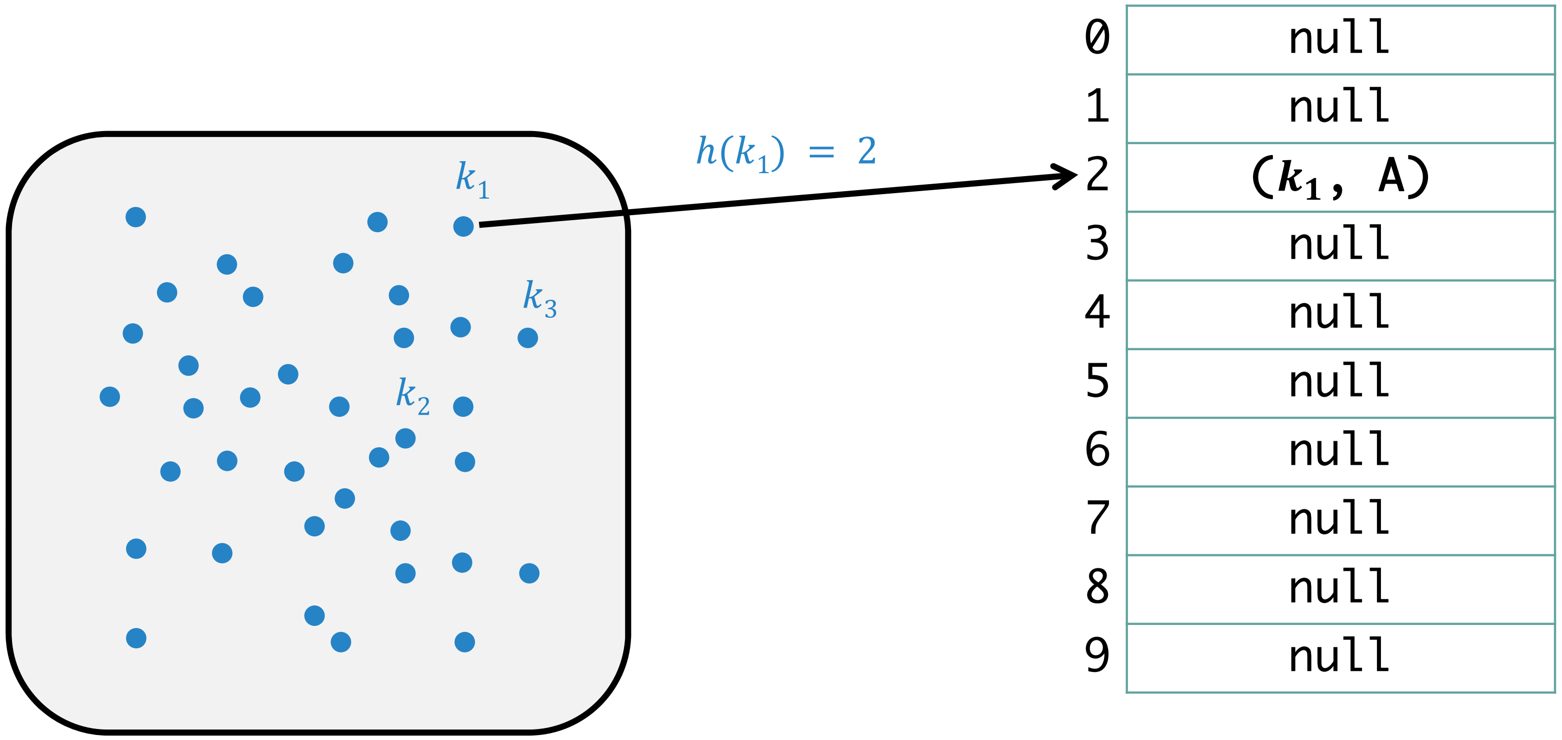
HASHING EXAMPLE



0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

HASHING EXAMPLE

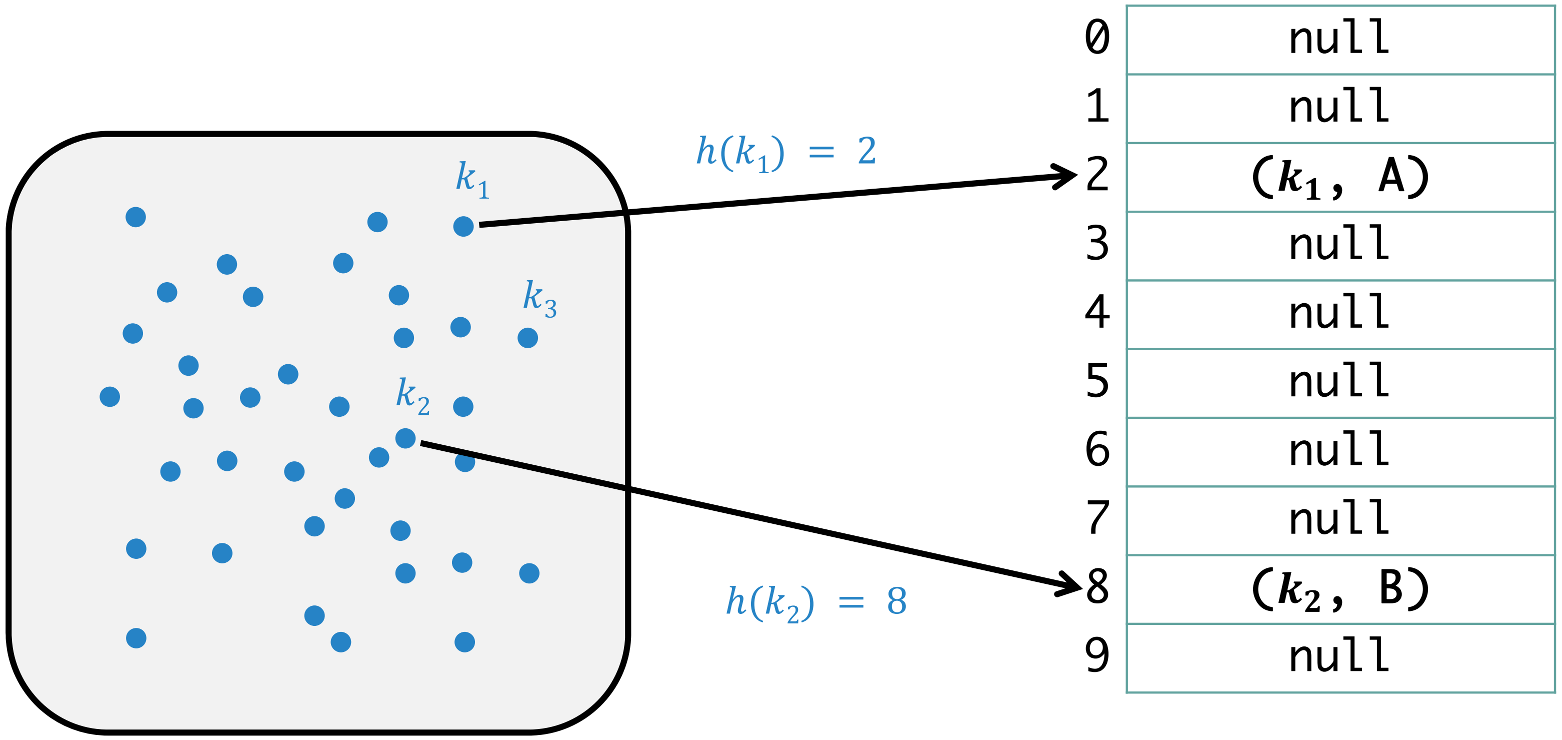
insert(k_1, A)



HASHING EXAMPLE

insert(k_1, A)

insert(k_2, B)



HASHING EXAMPLE

Collision!

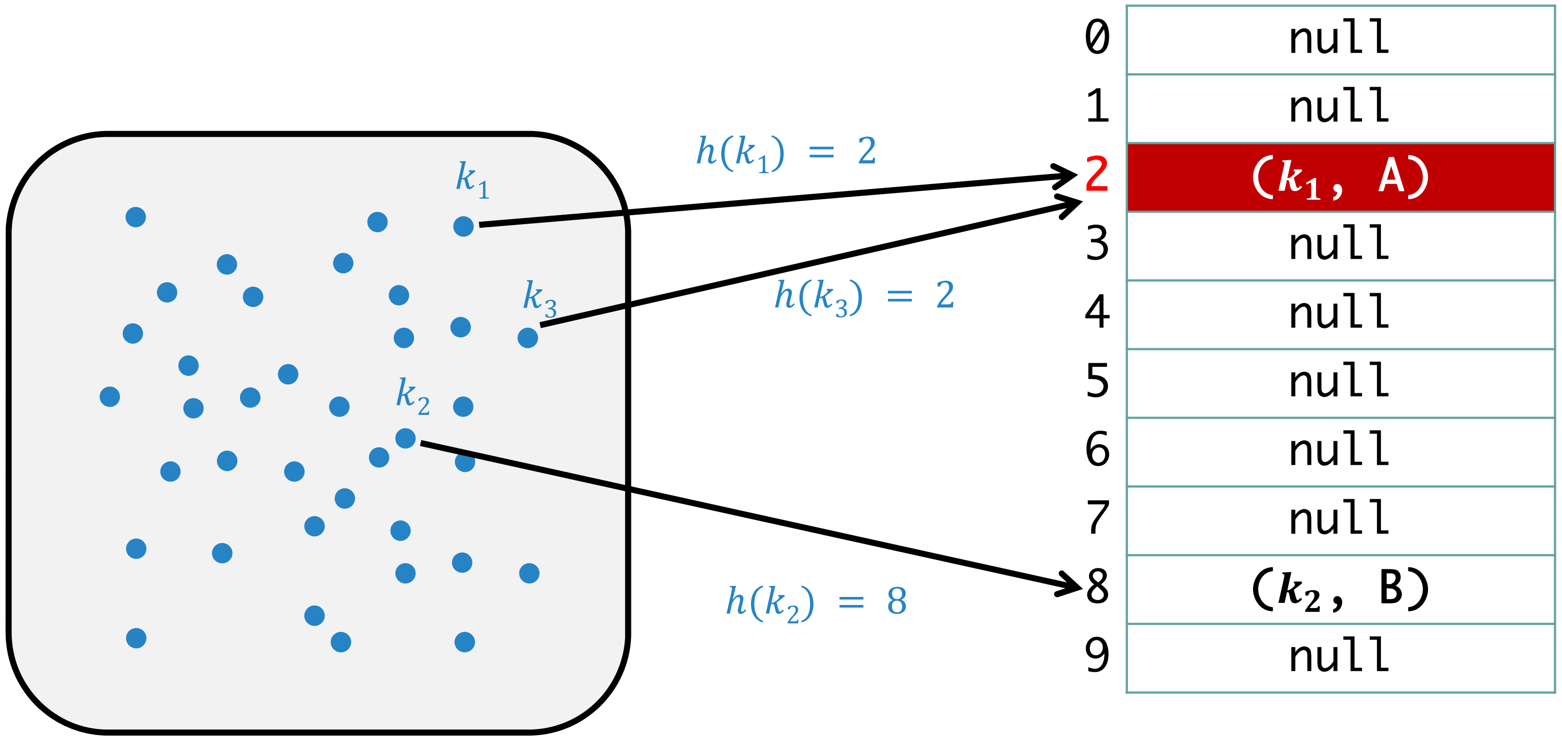
Two distinct keys k_1 and k_2 **collide** if:

$$h(k_1) = h(k_2)$$

insert(k_1, A)

insert(k_2, B)

insert(k_3, C)



COLLISIONS ARE A FACT OF LIFE

If you don't know the keys in advance.

- Otherwise, you can derive a perfect hash (google gperf)

Have a policy for handling collisions:

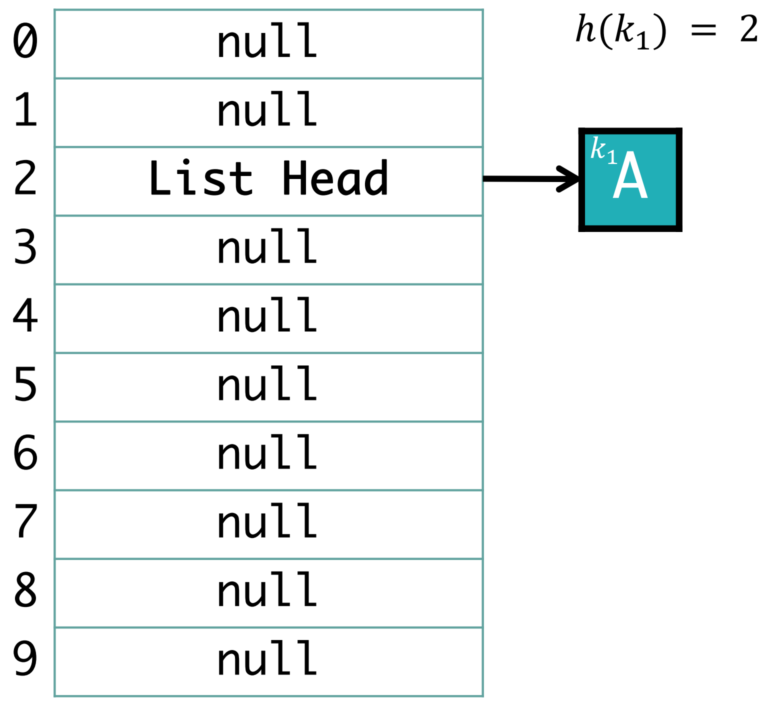
- Chaining (or Separate Chaining)
- Open Addressing

CHAINING

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)

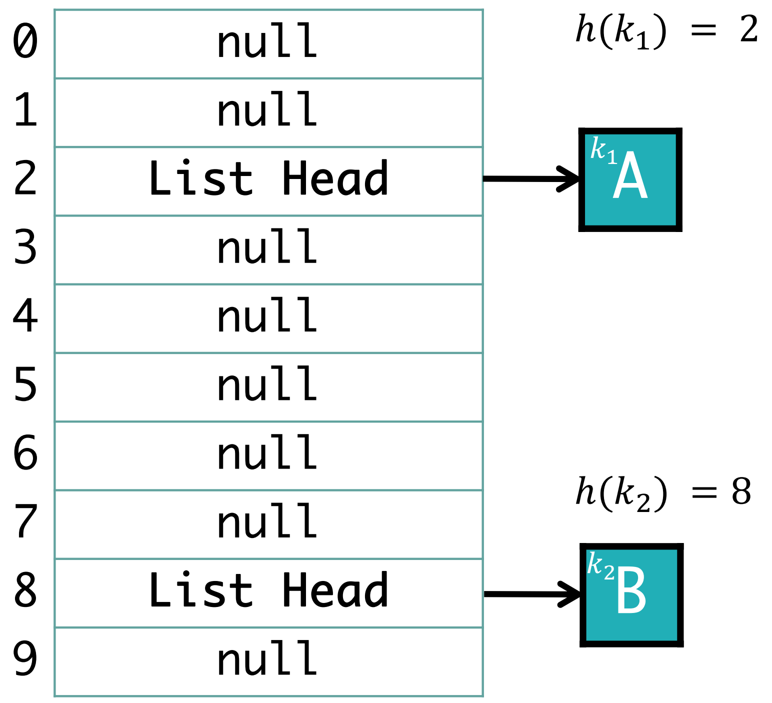


CHAINING

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)



CHAINING

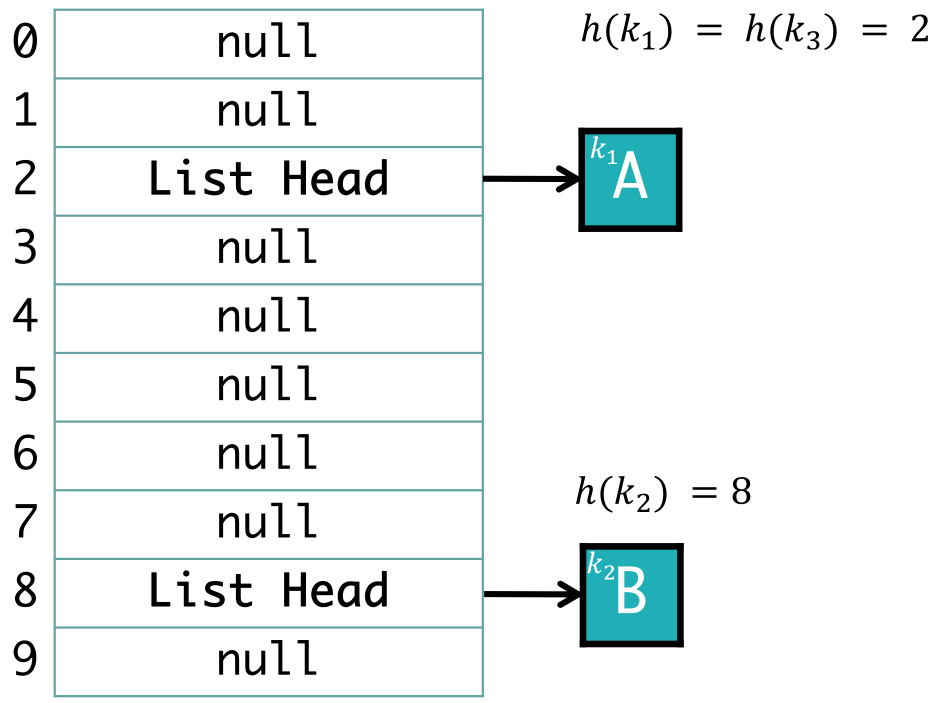
Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)
insert(k_3, C)

Collision.

but it's ok!



CHAINING

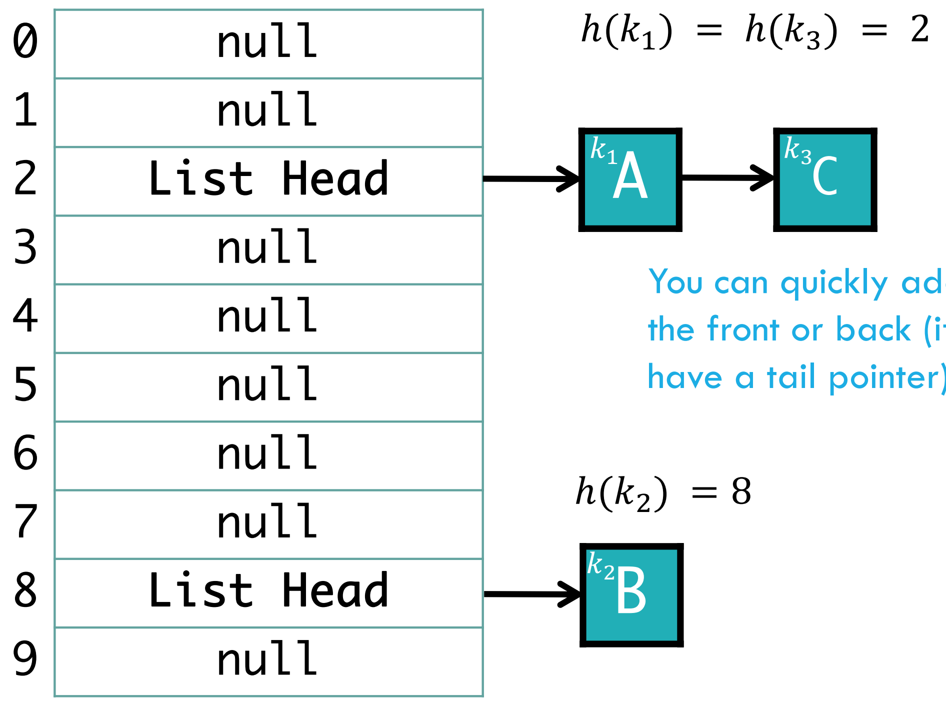
Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)
insert(k_3, C)

Collision.

but it's ok!



You can quickly add k_3 to the front or back (if you have a tail pointer)

SIMPLE UNIFORM HASHING ASSUMPTION

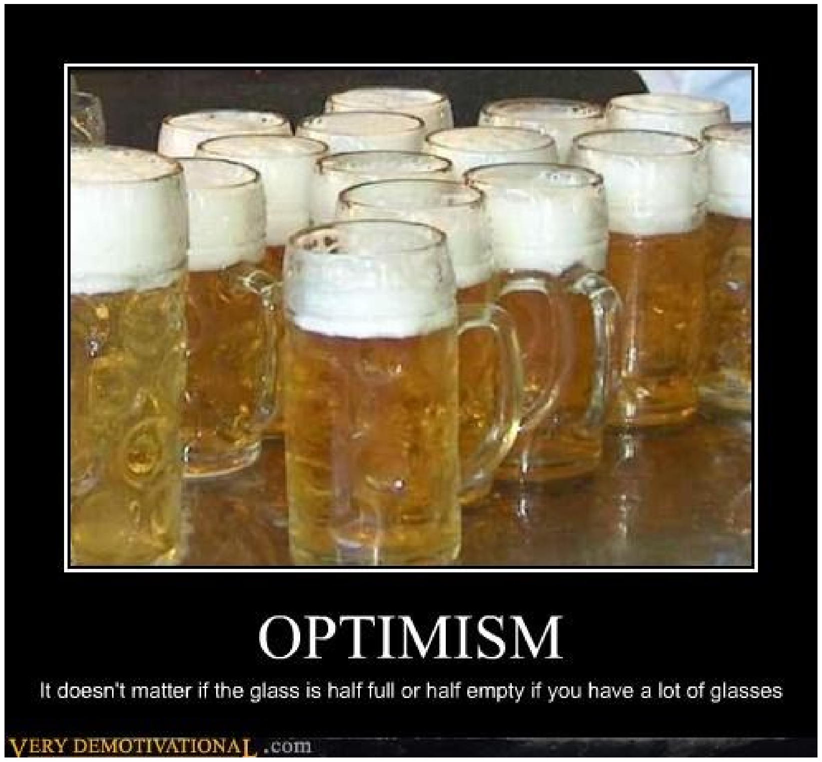
An **optimistic** assumption:

Every key is ***equally likely*** to map to every bucket

Keys are mapped ***independently***.

Intuition:

- Each key is put in a random bucket.
- As long as enough buckets, not too many keys in any one bucket.


OPTIMISM
It doesn't matter if the glass is half full or half empty if you have a lot of glasses
VERY DEMOTIVATIONAL .com

WHAT IS THE AVERAGE SEARCH TIME…

under the simple uniform hashing assumption (SUHA)

We have:

- m buckets
- n items
- Assume $n = \alpha m$ and $m \geq n$
- α is the "load factor"

$$\text{Expected search time} = \underbrace{1}_{\text{hashing + array access}} + \underbrace{\text{expected \# items per bucket}}_{\text{linked list traversal}}$$

What is the average search time under SUHA?

A. $O(m)$

B. $O(n)$

C. $\boldsymbol{O(1)}$

D. $O(m + n)$

E. I thought we were doing some fingerprint stuff?

Why?

WHAT IS THE AVERAGE SEARCH TIME...

$\mathbb{E}[X] = (p(\text{success}) \times 1) + (p(\text{fail}) \times 0)$
$= p(\text{success})$

under the simple uniform hashing assumption (SUHA)

We have:

- m buckets
- n items
- Assume $\boxed{n = \alpha m}$ and $m \geq n$ $\quad \alpha = \frac{n}{m}$
- α is the "load factor"

Expected search time $= \underbrace{1}_{\text{hashing + array access}} + \underbrace{\text{expected \# items per bucket}}_{\text{linked list traversal}}$

Proof Sketch:

Indicator random variables

$X(i,j) = 1$ if item i is in bucket j (success)
$X(i,j) = 0$ otherwise (fail)

Expected number of items in bucket b:

$$\mathbb{E}\left[\underbrace{\sum_{i}^{n} X(i,b)}\right] = \sum_{i}^{n} \underbrace{\mathbb{E}[X(i,b)]}$$

$$= \sum_{i}^{?} \frac{1}{m} = \frac{n}{m} = \alpha$$

Since $m > n$

$$\mathbb{E}\left[\sum_{i} X(i,b)\right] = O(1)$$

COLLISIONS ARE A FACT OF LIFE

If you don't know the keys in advance.

- Otherwise, you can derive a perfect hash (google gperf)

Have a policy for handling collisions:

- Chaining (or Separate Chaining)
- Open Addressing

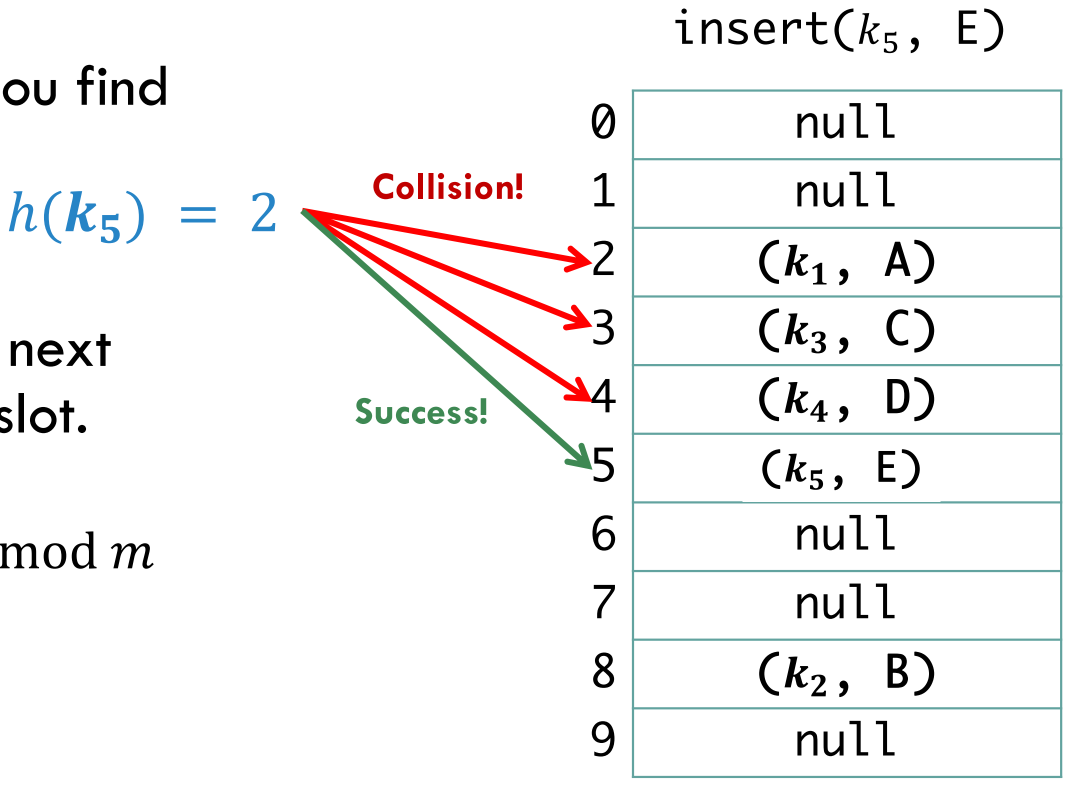
OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

$$\text{index } i = (\underbrace{h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$$



A PROBLEM: PRIMARY CLUSTERS

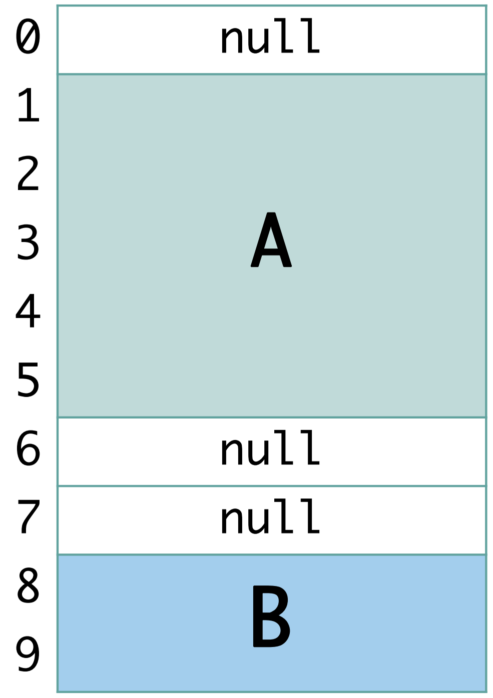
cluster = collection of consecutive occupied slots

In a hash table of size 10, consider 2 clusters:

- A: size 5
- B: size 2

Probability that a new inserted key k has a bucket in:

- cluster A? 5/10
- cluster B? 2/10



OPEN ADDRESSING: QUADRATIC PROBING

Linear probing: index $i = (h(k) + \text{step} \times 1) \bmod m$

$\Downarrow$

Quadratic probing: index $i = (h(k) + \text{step}^2) \bmod m$

Example:

- $h(k) = 3, m = 7$
- Step 0: $i = h(k) = 3$
- Step 1: $i = (h(k) + 1) \bmod 7 = 4$
- Step 2: $i = (h(k) + 4) \bmod 7 = 0$
- Step 3: $i = (h(k) + 9) \bmod 7 = 5$

Is this a good probing method?

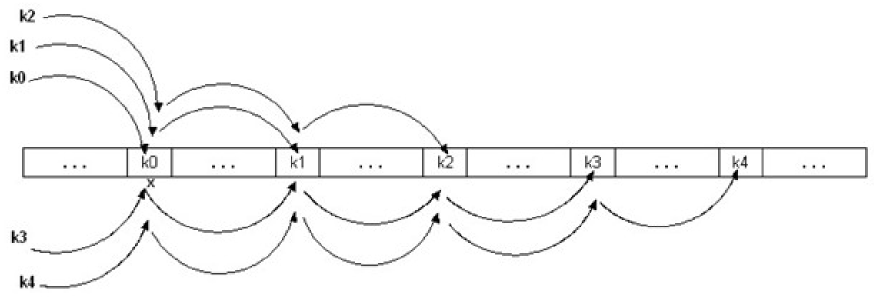
ONE PROBLEM: SECONDARY CLUSTERING

Milder form of the clustering problem.

Because: if two keys have the same probe position, their probe sequences are the same.

Clustering around different points (rather than the primary probe point)

index $i = (h(k) + \text{step}^2) \bmod m$



How many probe sequences can there be? m

DOUBLE HASHING

Use a second hashing:

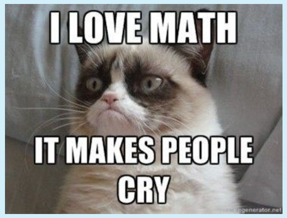
index i $= (h_1(k) \; + \; \text{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing more unique probing sequences.

Intuition:

- **$h_1(k)$ provides good "random" base address**
- **$h_2(k)$ provides good "random" sequence**

Up to how many unique indexing sequences does double hashing provide?

A. m
B. $2m$
C. m^2
D. 2^m
E. 

DOUBLE HASHING

Use a second hashing:

index i $= (h_1(k)\ +\ \mathrm{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing **up to m^2** probing sequences.

To work:

- Needs careful choice for h_1 and h_2
- Make m prime and $h_2 < m$
- Also, $h_2(k) \neq 0$ **Why?**

One technique is to choose:

$$h_2 = (ak\ mod\ b) + 1$$

where $b < m$

Poll Everywhere

https://bit.ly/2LvG9bq

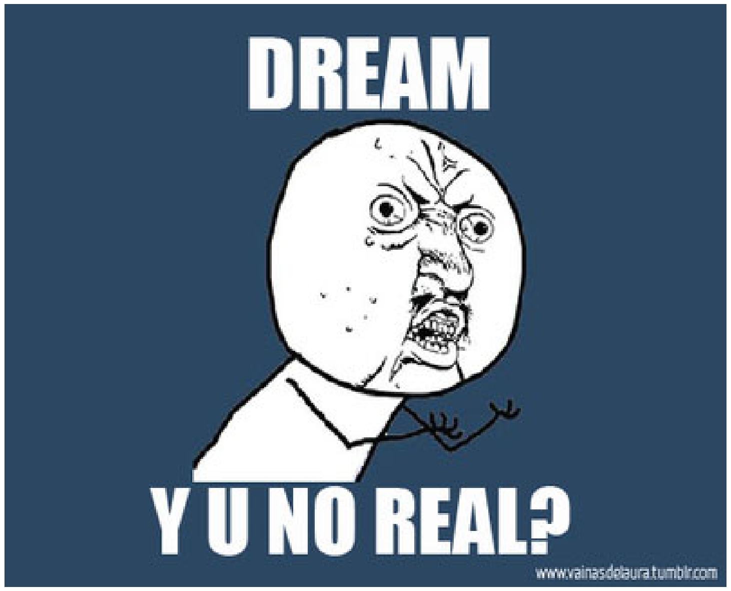
QUESTIONS?

SUHA IS A DREAM!

Simple Uniform Hashing doesn't exist (in general).

BUT: Tells us properties of a "good" hashing function:

A. Consistent: same key maps to same bucket.
B. Fast to compute, $O(1)$
C. Scatter the keys into different buckets as uniformly as possible $\in [0..m-1]$



DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

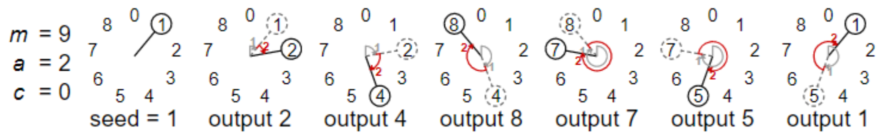
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



Poll Everywhere

https://bit.ly/2LvG9bq

REGULARITY AND COMMON DIVISORS

Division method: $h(k) \;=\; k \bmod m$

If k and m have a common divisor d

then:

$$\underbrace{k}_{\text{divisible by } d} = \underbrace{im}_{\text{divisible by } d} + \underbrace{k \bmod m}_{\text{divisible by } d}$$

Assume chaining. How much of the table do we use if both x and m have a common divisor d?

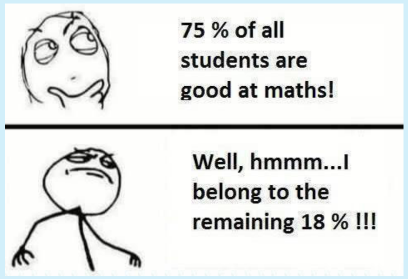
A. d

B. 2d

C. $1/d^2$

D. $1/d$

E.


75 % of all students are good at maths!
Well, hmmm...I belong to the remaining 18 % !!!

REGULARITY AND COMMON DIVISORS

Division method: $h(k) = k \bmod m$

If k and m have a common divisor d

then:

$$\underbrace{k}_{\text{divisible by } d} = \underbrace{im}_{\text{divisible by } d} + \underbrace{k \bmod m}_{\text{divisible by } d}$$

Choose m such that it has no common factors with any k

will only use 1 out of every d slots!

0	(k_1, A)
1	null
2	null
d=3	(k_2, B)
4	null
5	null
2d=6	(k_3, C)
7	null
8	null
3d=9	(k_4, D)

DIVISION METHOD

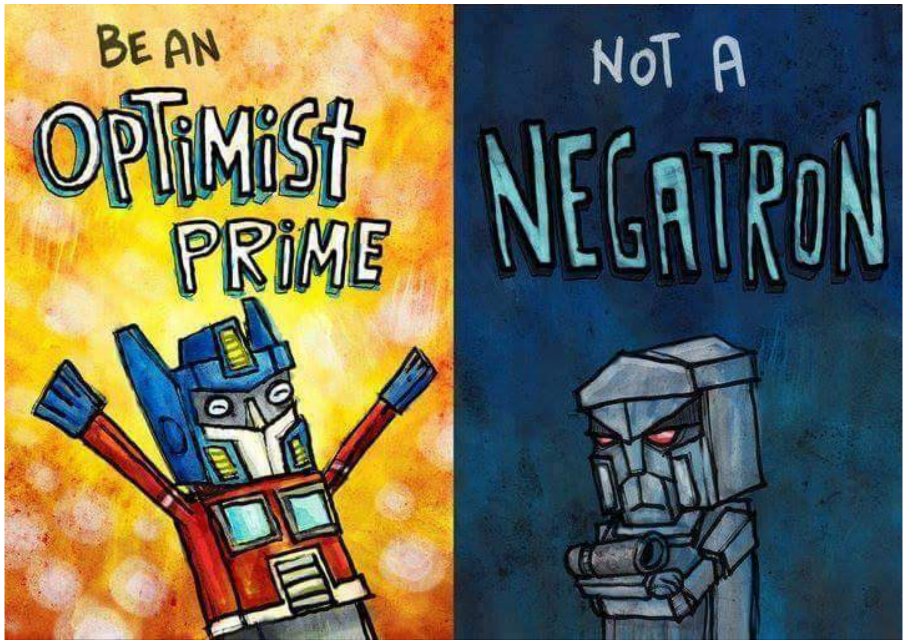
Choose m to be **prime**

- Avoid powers of 2 and powers of 10

In practice: popular and easy

But not always the most effective.

Slow (no more shifts)



DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

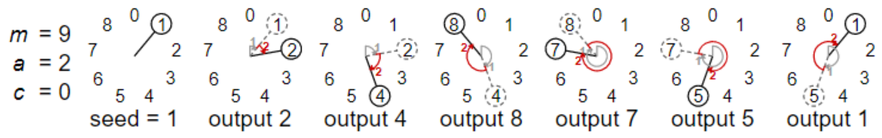
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



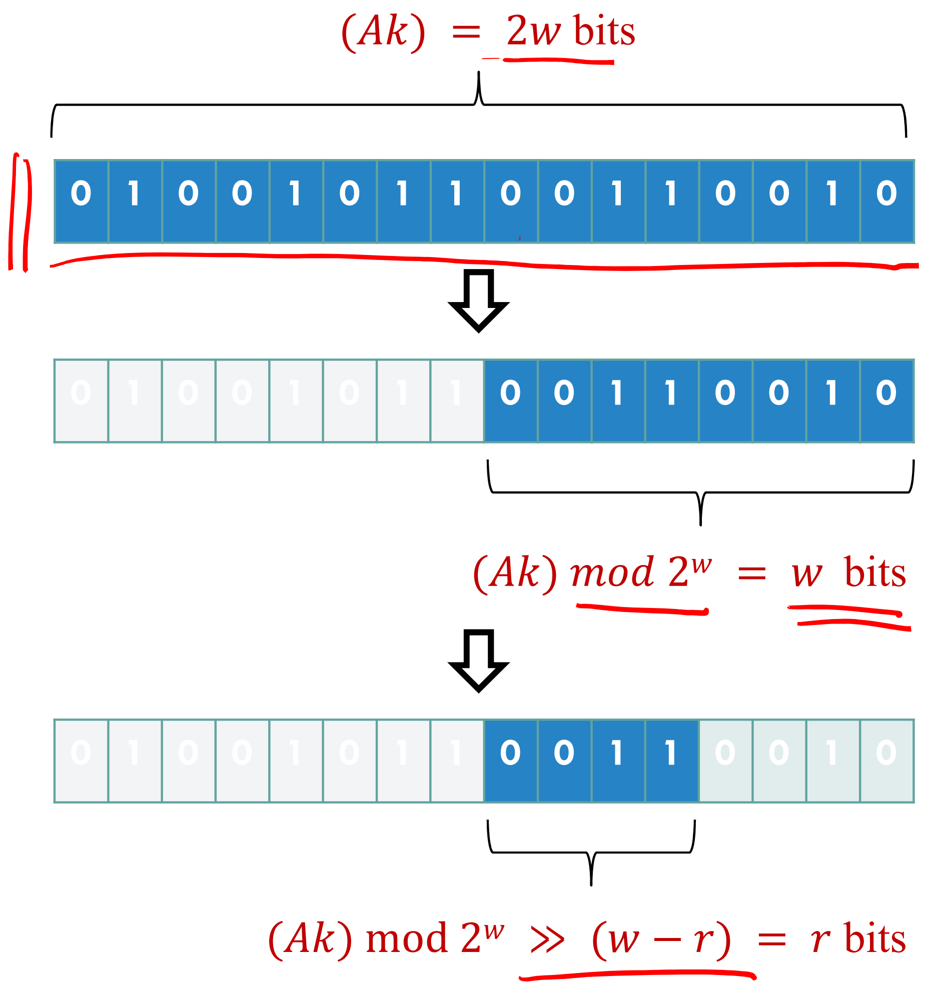
MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$



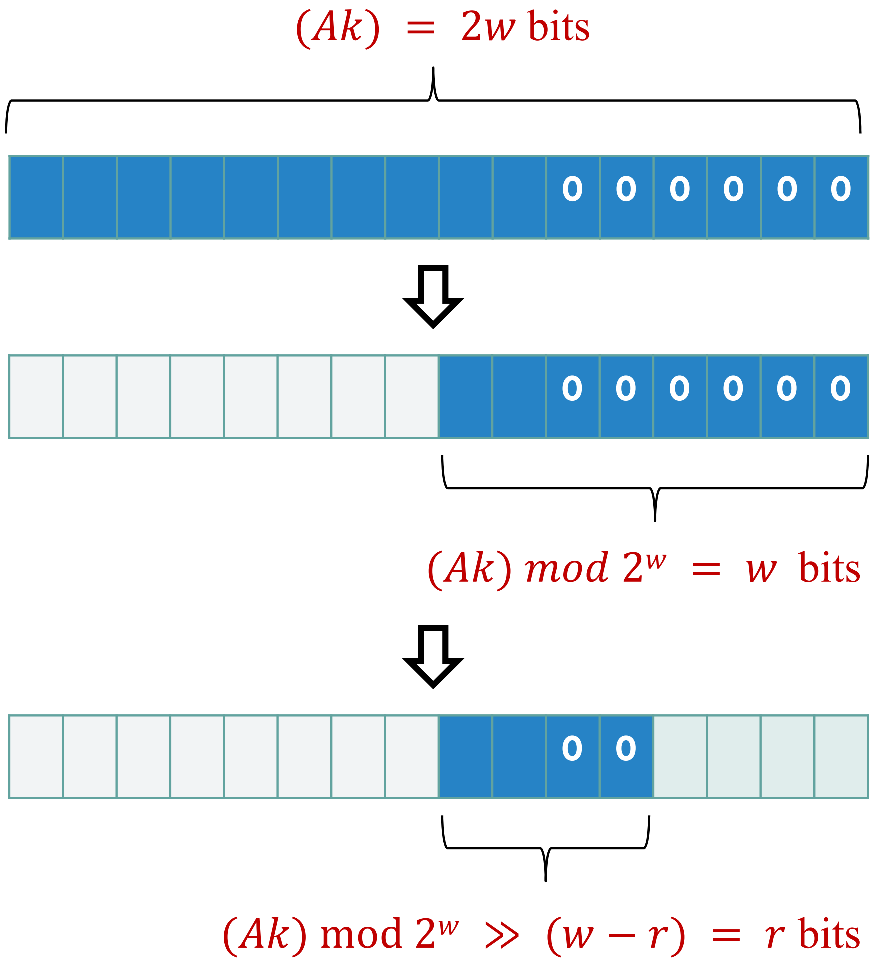
MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$


$(Ak) \; = \; 2w$ bits
0 0 0 0 0 0
0 0 0 0 0 0
$(Ak) \; mod \; 2^w \; = \; w$ bits
0 0
$(Ak) \bmod 2^w \gg (w-r) \; = \; r$ bits

Consider what happens when A is even, say $2^{w-1} + 64$. (see example in the right)

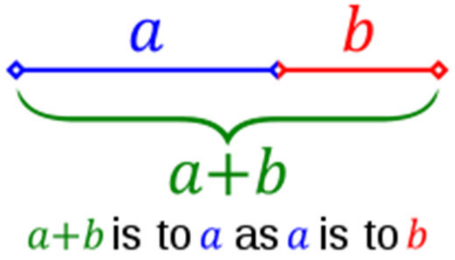
Point: even numbers cause at least one bit of information loss.

MULTIPLICATION METHOD

Choose A, r carefully

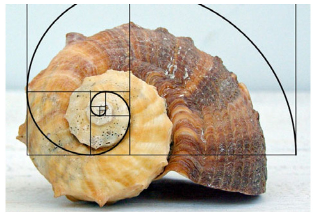
In practice: works well with A is chosen well (e.g., odd)

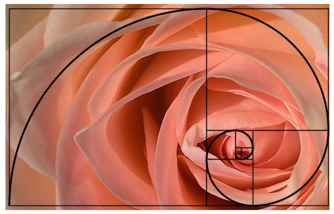
Knuth recommends $A \approx \frac{\sqrt{5}-1}{2} \cdot 2^{32}$ for $w = 32$ bit words



Donald Knuth

wrote The Art of Computer Programming (TAOCP)





DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

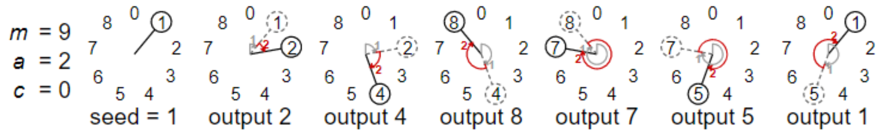
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



Poll Everywhere

https://bit.ly/2LvG9bq

QUESTIONS?

SAMPLE PROBLEM: NEAREST REPEATED WORDS (WARMUP)

Harold Soh
harold@comp.nus.edu.sg

NEAREST REPEATED WORDS

“I am so happy we’re getting more problems to solve. Nothing pleases me more than solving problems. I love solving problems. Especially tough problems. The harder the better! Give me more problems!”

Difficulty level 2 (out of 5).

NEAREST REPEATED WORDS

```
 "I am so happy we're getting more problems to
solve. Nothing pleases me more than solving
problems. I love solving problems. Especially tough
problems. The harder the better! Give me more
problems!"
```

Assume words are given to you in a list. Describe the most efficient algorithm you can think of for finding the ***distance*** (in terms of number of words) between the *closest* repeated word. You do not need to provide pseudocode. Ignore punctuation and capitalization.

Extra: Can you provide an algorithm for the case when memory is limited. Minimize the amount of memory required for your algorithm. What are the trade-offs?

NEAREST REPEATED WORD

Idea:

Assume words are in a list

maintain a minimum distance, dmin

loop through the words.

For each word w at position i, find the closest repetition by scanning forward in the list until the same word is found, say at position j.

if the distance d = j-i is less than dmin, update dmin = d

```
the
harder
the
better
give
me
more
problems
```

NEAREST REPEATED WORD: COMPLEXITY?

Idea:

Assume words are in a list

maintain a minimum distance, dmin

loop through the words.

For each word w at position i, find the closest repetition by scanning forward in the list until the same word is found, say at position j.

if the distance d = j-i is less than dmin, update dmin = d

Looping through each word takes n time.
for each word, we need to scan forwards.

$$(n-1)+(n-2)+\cdots+(n-(n-1))$$
$$=1+2+\ \ldots+(n-1)=O(n^2)$$

Can we do better?

NEAREST REPEATED WORD: IDEA

Idea:

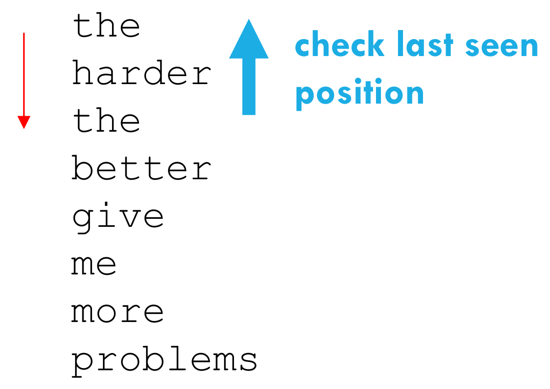
Assume words are in a list

maintain a minimum distance, dmin

loop through the words.

For each word w at position i, find the closest repetition by scanning forward in the list until the same word is found, say at position j.

if the distance d = j-i is less than dmin, update dmin = d



```
the
harder
the
better
give
me
more
problems
```

NEAREST REPEATED WORD: VERSION 2

Idea:

Assume words are in a list

maintain a minimum distance, dmin

maintain a hash table H with word keys and last seen position

for each word w at position i,

- lookup the word in H
- if H contains w, check if the distance d=i-H[w]. If d < dmin, update dmin = d.
- Then update: H[w] = i

NEAREST REPEATED WORD: VERSION 2: COMPLEXITY

Idea:

Assume words are in a list

maintain a minimum distance, dmin

maintain a hash table H with word keys and last seen position

for each word w at position i,

- lookup the word in H
- if H contains w, check if the distance d=i-H[w]. If d < dmin, update dmin = d.
- Then update: H[w] = i

Looping through each word takes n time.
for each word, we check hash table $O(1)$ so, total $O(n)$.

Here we have assumed the hashing function is $O(1)$. If the max length of a word is m, then the cost is $O(mn)$ assuming a linear hash function or a trie is used.

Poll Everywhere

https://bit.ly/2LvG9bq

QUESTIONS?

ON TO GRAPHS!

Harold Soh
harold@comp.nus.edu.sg

TERMINOLOGY SUMMARY

Graph: $G = \langle V, E \rangle$

Degree of a node: number of edges connected to it

Diameter: longest shortest path between two different nodes

Connected Graph: path between any two nodes

Clique: fully connected graph

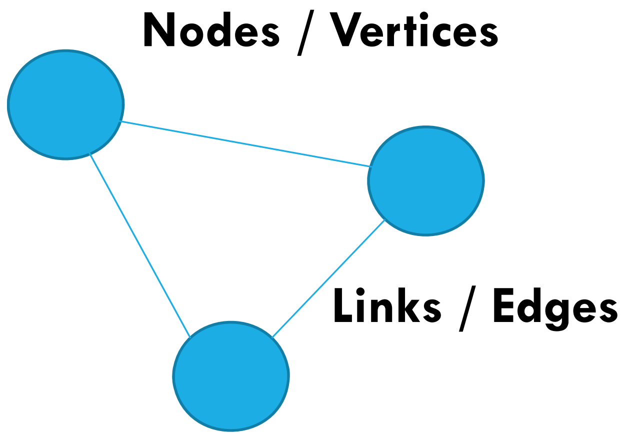
Line Graph: a line (duh!)

Star: central node connected to all other nodes.

UNDIRECTED GRAPHS: A FORMAL DEFINITION

Graph $G = \langle V, E \rangle$ ("a tuple of two sets")

- V is a set of nodes
- E is a set of edges
  - $E \subseteq \{ (v, w) : v, w \in V \}$



Simple Graph:

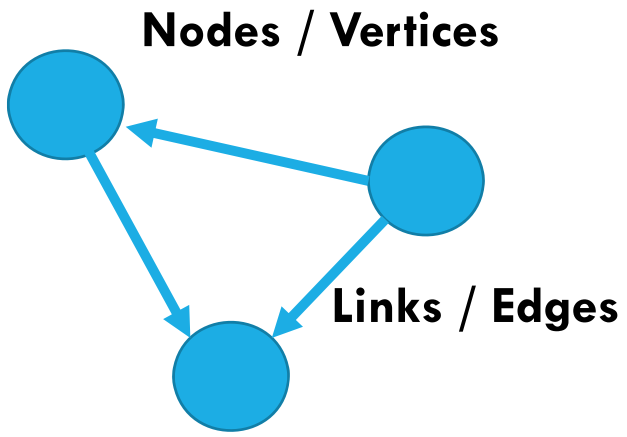
- $e = (v, w)$ for $v \neq w$ ("no self loops")
- $\forall e_1, e_2 \in E : e_1 \neq e_2$ ("only one edge per pair of nodes")

DIRECTED GRAPHS

Graph $G = \langle V, E \rangle$ ("a tuple of two sets")

- V is a set of nodes
- E is a set of edges
  - $E \subseteq \{ \underbrace{(v, w)}_{\text{Order matters!}} : v, w \in V \}$

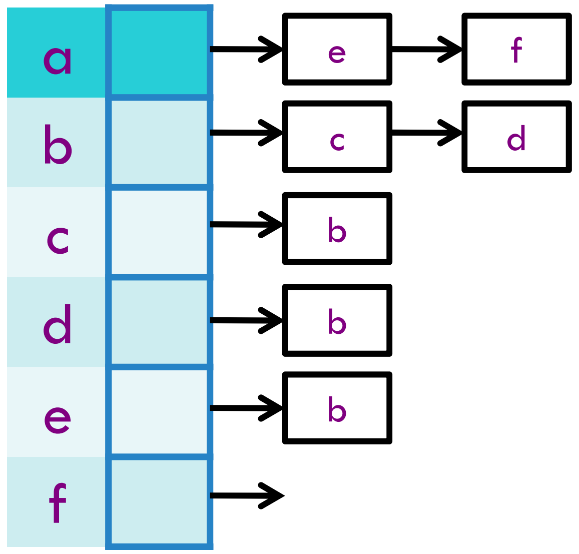
(v, w) means an edge pointing from $v \rightarrow w$

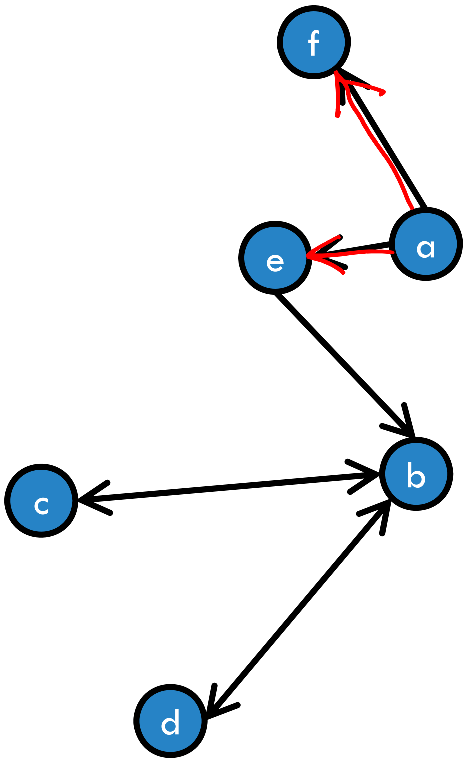


ADJACENCY LIST

Directed Graph consists of:

- Nodes: stored in an array
- Outgoing Edges: linked list per node



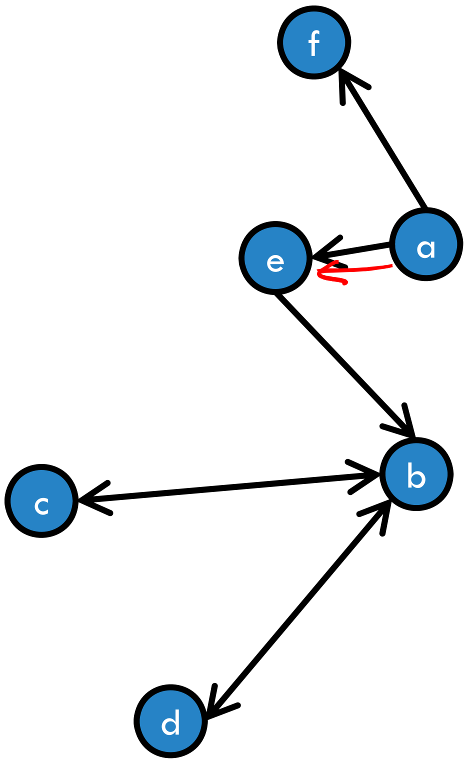


ADJACENCY MATRIX

row col.

$$A[\underline{v}][\underline{w}] = 1 \text{ iff } (\underline{v}, \underline{w}) \in E$$

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	0	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	0	1	0	0	0	0
f	0	0	0	0	0	0



SEARCHING A GRAPH

Goal:

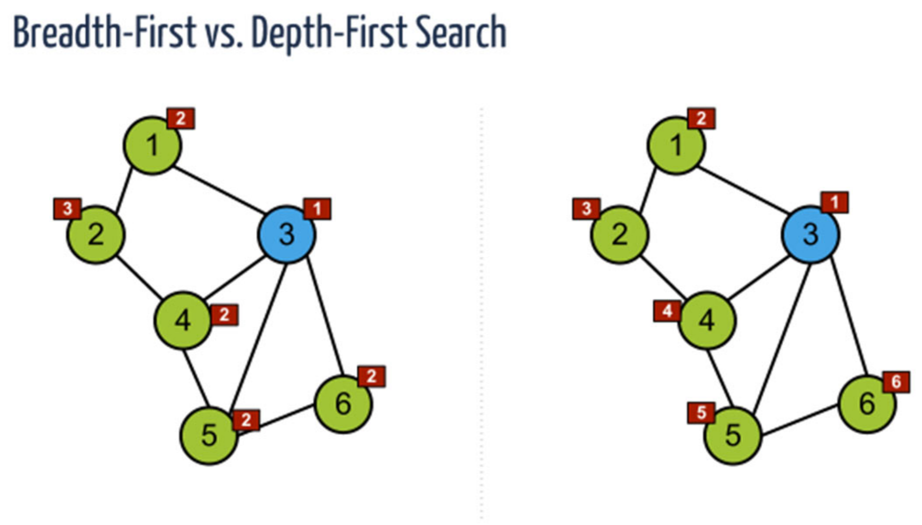
- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.
  - Or: visit **all** the nodes in the graph

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list

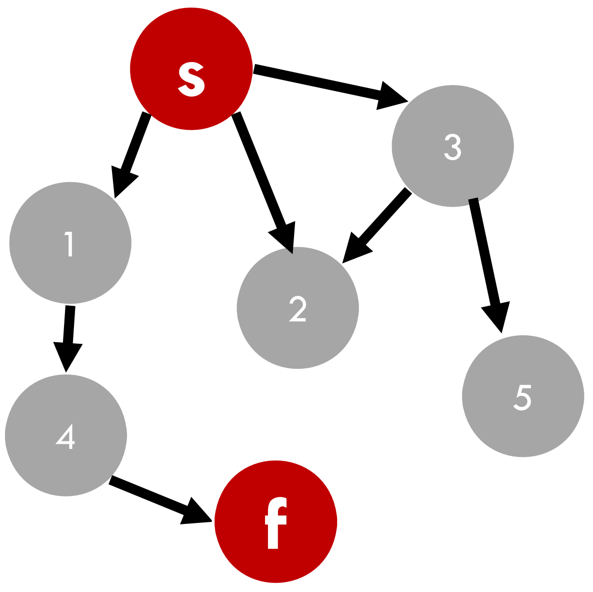


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:



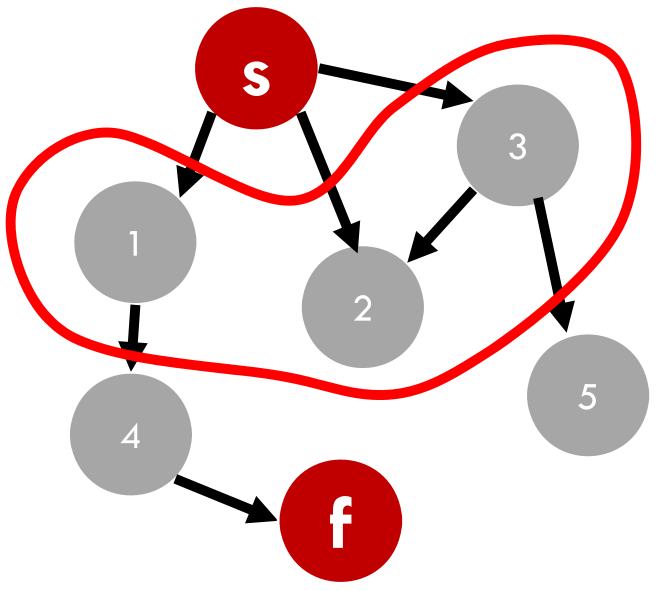
Found!

Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:



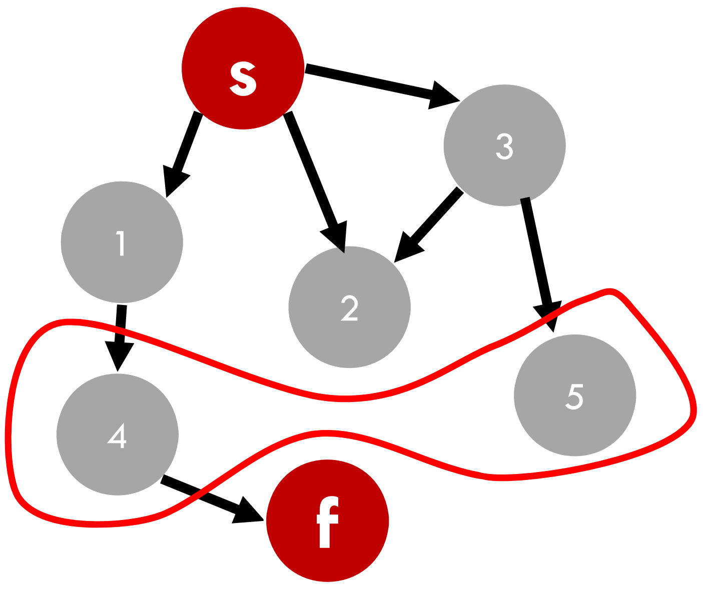
Found!

Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:



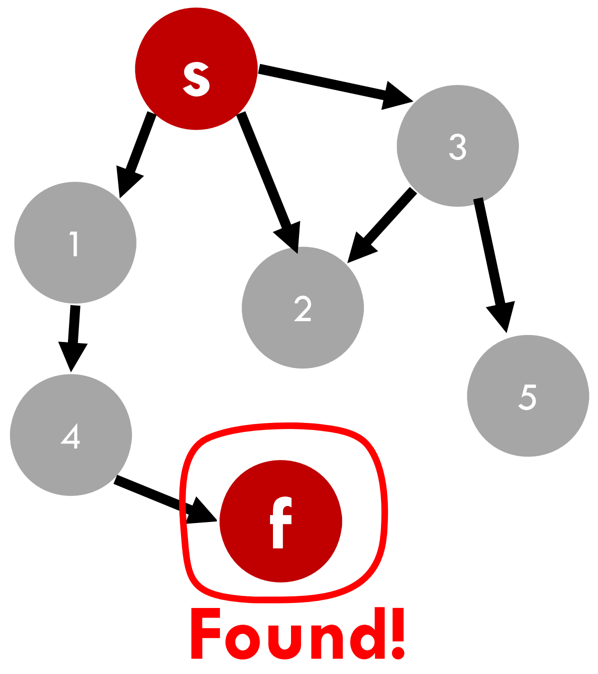
Found!

Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

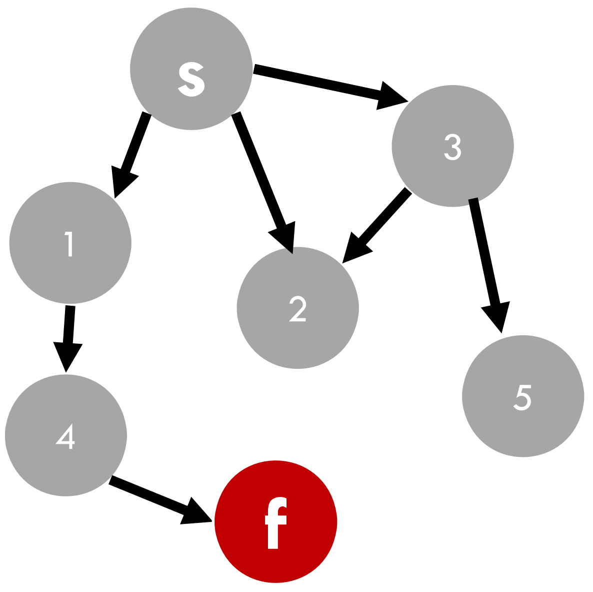


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



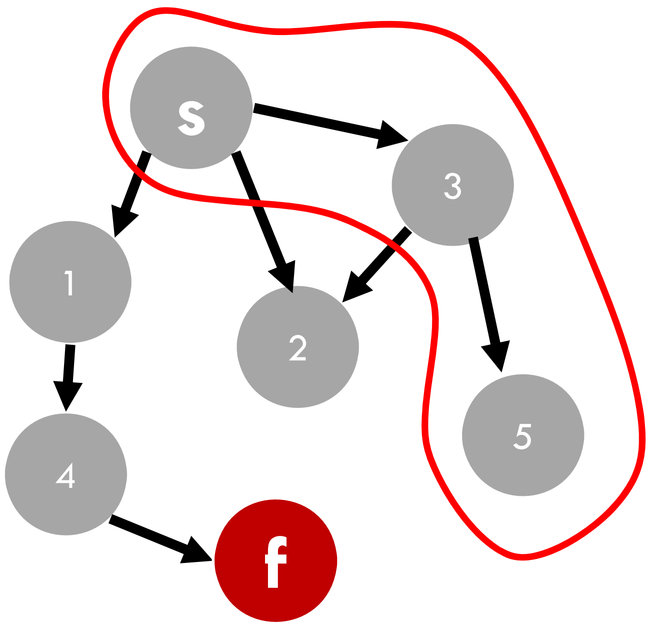
Found!

Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



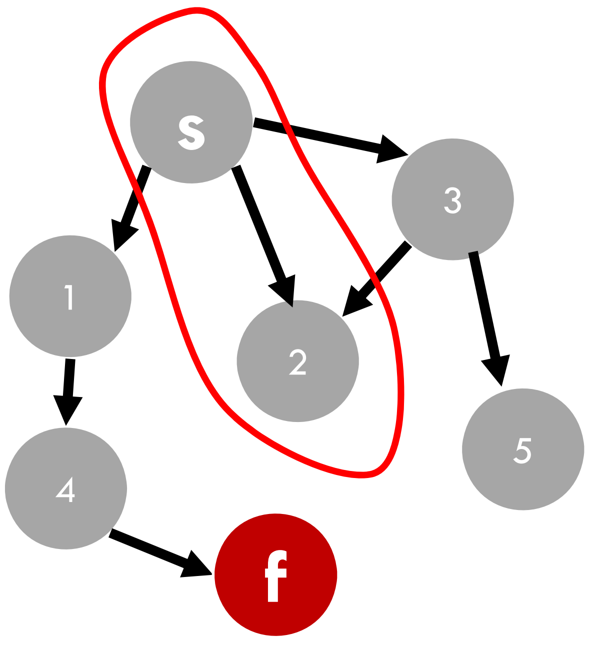
Found!

Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



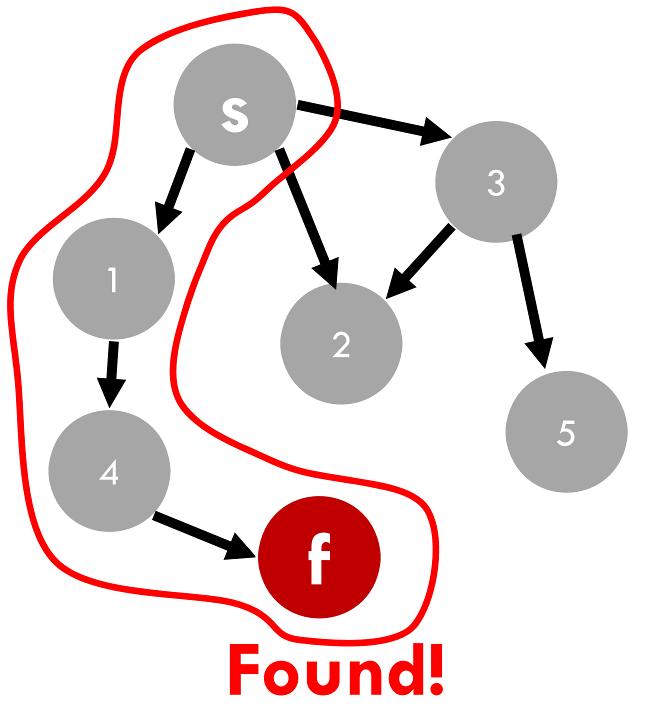
Found!

Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



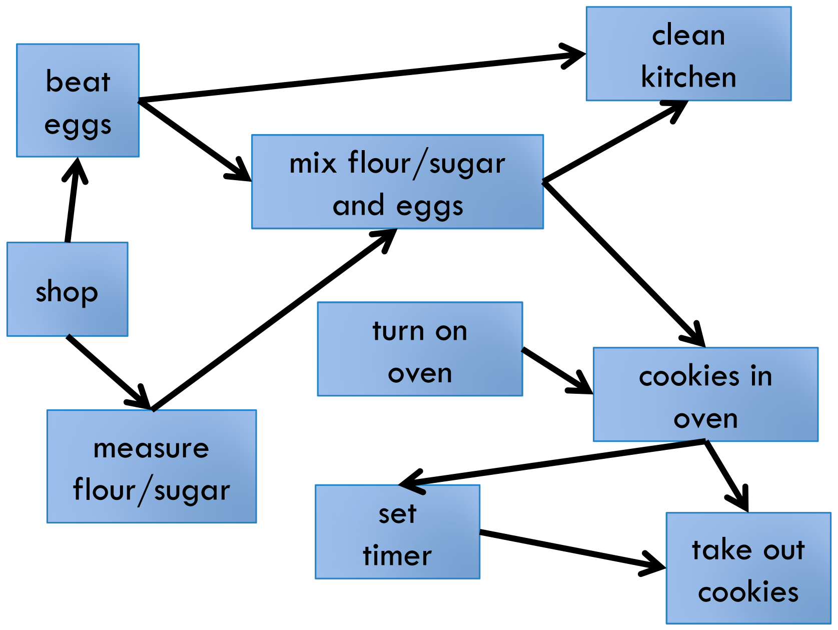
TOPOLOGICAL SORT

Input(s):

- Input is a graph. Any graph?
- A DAG!
- Represented as a?
- Adjacency list

Output(s):

- A list of nodes in topological order.
  - No node in the list can have an incoming edge from a node that appears later (in the list).



KAHN'S ALGORITHM

Start at any node v with no incoming edges.

Add v to our list

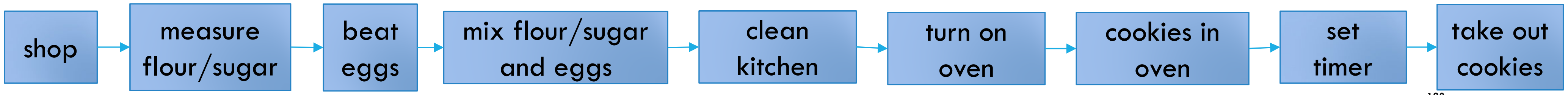
Remove v and all its outgoing edges.

Repeat

Pseudocode:

```
L = list()
S = list()
add all nodes with no incoming edge to S
while S is not empty:
    remove node v from S
    add v to tail of L
    for each of v's neighbors u
        remove edge e where source is v
        if u has no other incoming edges
            add u to S
```

What is the time complexity? $O(V + E)$



TOPOLOGICAL SORT USING DFS (ASSUME DAG)

Idea: Process node when it is “last” visited.

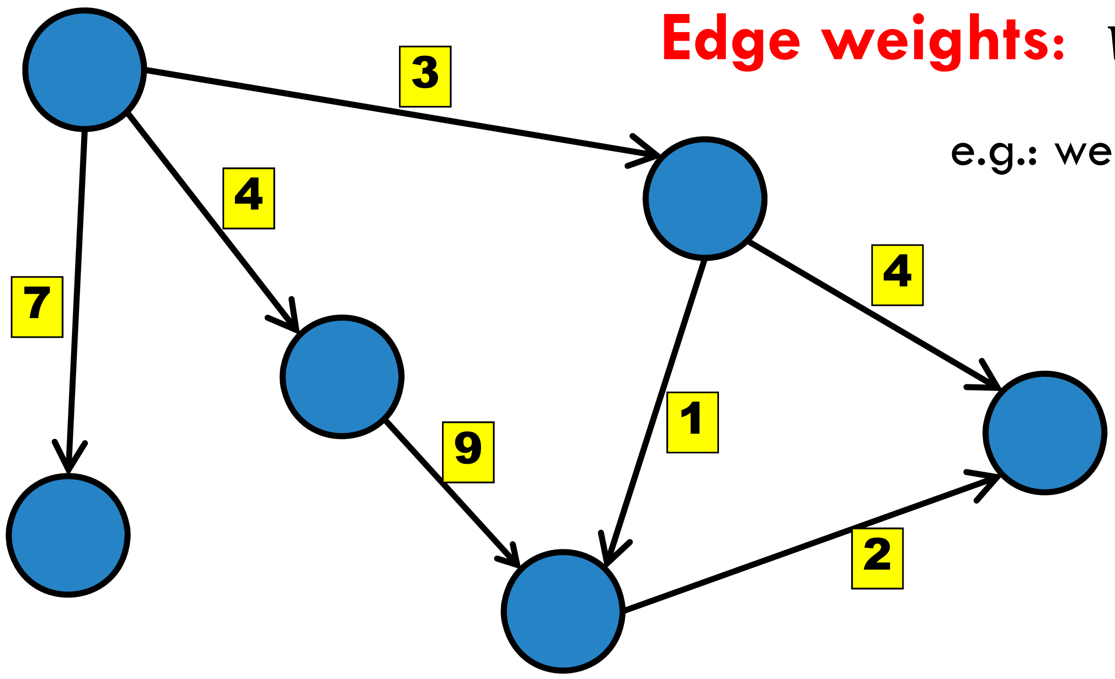
```
L = list()
while there are unvisited nodes
    v = select unvisited node
    DFS(G, v, L)
```

```
DFS(G,v,L)
    if v is visited
        return
    else
        for each of v’s neighbor u
            DFS(G, u, L)
    visit(v)
    L.pushFront(v)
```

How can we quickly check if there are unvisited nodes or select unvisited nodes?

List/ Hash Table / Set

WEIGHTED GRAPHS

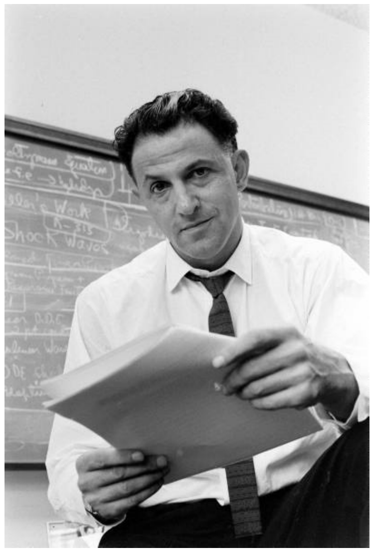


Edge weights: $w(e)\colon E \rightarrow \mathbb{R}$

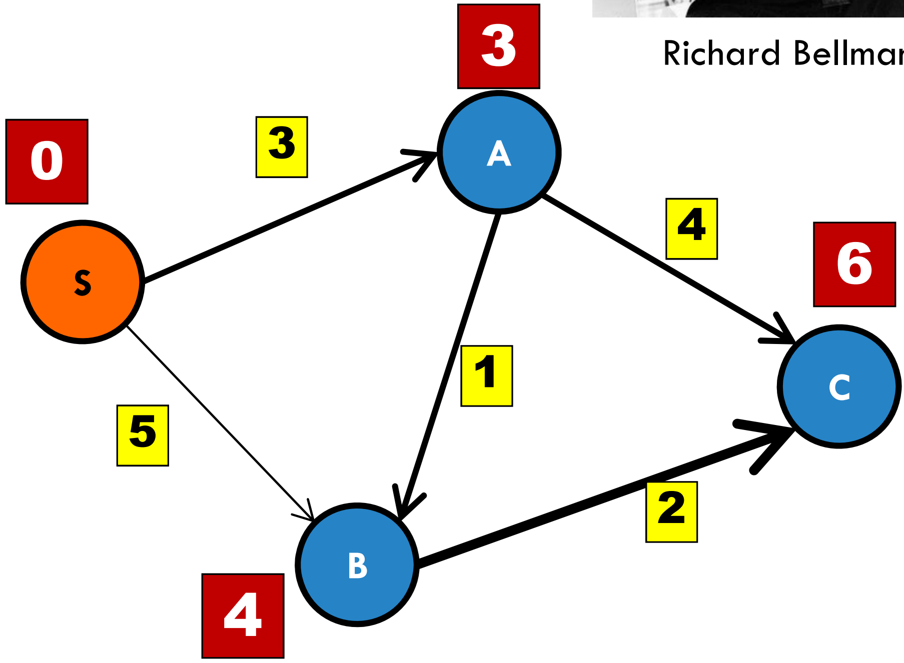
e.g.: weight = distance

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



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?