Final Review

A High-level Picture

What we have learned

I) Basic Techniques for algorithmic analysis

Asymptotic analysis (big-O notation), proofs of correctness, runtime analysis
Solving recurrences: Recursion tree method, iteration method, master theorem

II) Sorting Algorithms

Insertion sort, Merge sort, Quick sort, Sorting lower bound, linear sorting algorithms,
Sorting data structures (Binary search tree and Red black tree)

A High-level Picture

What we have learned (cont.)

III) 5 algorithmic paradigms

Divide and Conquer: Merge sort, Quick sort, Integer multiplication, Select_k

Randomized Algorithm:

Las Vegas: Quick sort, Quick select, Majority element, Hash tables, Expected runtime analysis

Monte-Carlo: Karger’s Algorithm for finding minimum cut, Probability of success analysis

Graph Algorithm:

Graph Basics: Graph representation, DAG, DFS, BFS, Topological Ordering, In-order traversal of BST

Shortest Path: Using BFS, Dijkstra’s Algorithm (SSSP), Bellman-Ford (SSSP), Floyd-Warshall (APSP)

SCC: Kosaraju’s Algorithm

Global Minimum Cut: Karger’s Algrothm

Maxflow-Mincut: Ford-Fulkerson Algorithm

Greedy Algorithm

Frog Hopping, Proof of correctness

Minimum Spanning Tree: Prim’s Algorithm (lightest edge), Kruskal’s Algorithm (cheapest edge)

Dynamic Programming

Four steps of designing dynamic programming algorithm

Bellman-Ford Algorithm (SSSP), Floyd-Warshall (APSP)

Longest Common Subsequence, 0/1 and Unbounded Knapsack

Basic Technics for Algorithm Analysis

Big-O Notation

Big-O notation is a mathematical notation for upper-bounding a function's **rate of growth**.

- Informally, it can be determined by ignoring constants and non-dominant growth terms.

Examples

- $n + 137 = O(n)$
- $3n + 42 = O(n)$
- $n^2 + 3n - 2 = O(n^2)$
- $n^3 + 10n^2 \log n - 15n = O(n^3)$
- $2^n + n^2 = O(2^n)$

Big-O Notation

Formally speaking, let f, g: N → N.

Then $f(n) = O(g(n))$ iff

$\exists n_0 \in N, c \in R.$

$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$

Intuitively, this means that f(n) is upper-bounded by g(n) aka f(n) is "at most as big as" g(n).

Big-Ω Notation

Let f, g: N → N.

Then f(n) = Ω(g(n)) iff

$\exists n_0 \in N, c \in R.$

$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$

Intuitively, this means that f(n) is lower-bounded by g(n) aka f(n) is "at least as big as" g(n).

Big-Θ Notation

$f(n) = \Theta(g(n))$ iff both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

More verbosely, let $f, g: N \to N$.

Then $f(n) = \Theta(g(n))$ iff

$\exists n_0 \in N$, c_1 and $c_2 \in R$.

$\forall n \in N$.

$(n \geq n_0 \to c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$

Intuitively, this means that f(n) is lower and upper-bounded by g(n) aka f(n) is “the same as” g(n) (in terms of growth rate).

Runtime Analysis

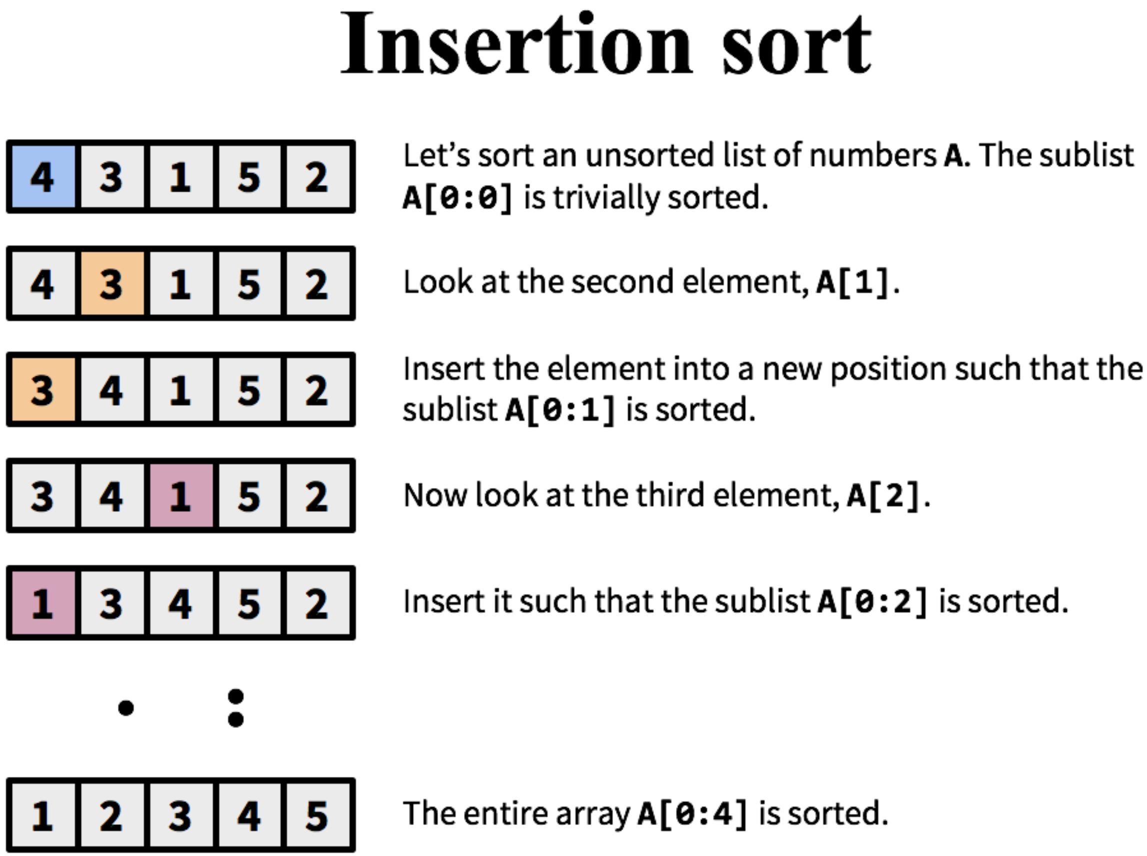
We usually care about the worst case of an algorithm, i.e., to find the upper bound $O(f(n))$ of the runtime.

$f(n)$ is a function of the input size n, e.g., $\log(n)$, n^2, $n\log n$, etc.

The input size can be length of array, number of nodes/edges in a graph, etc.

Runtime: Direct method

To calculate the total number of operations directly.



Runtime: Solving Recurrence

Some algorithms can not be easily analyzed directly

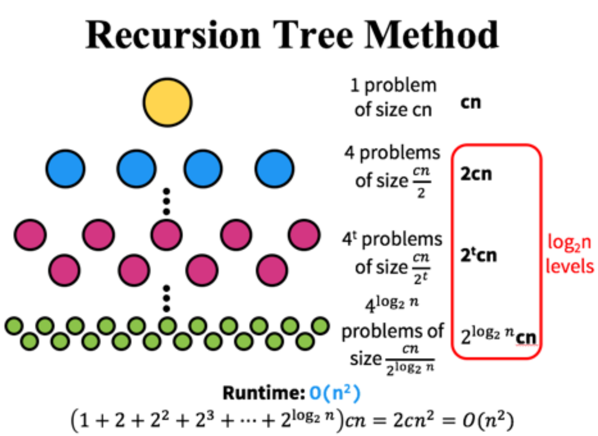
- Especially programs that involve recursive programming

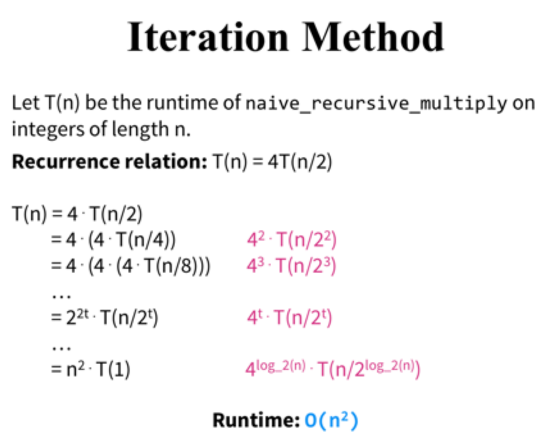
Write out the Recurrence relation

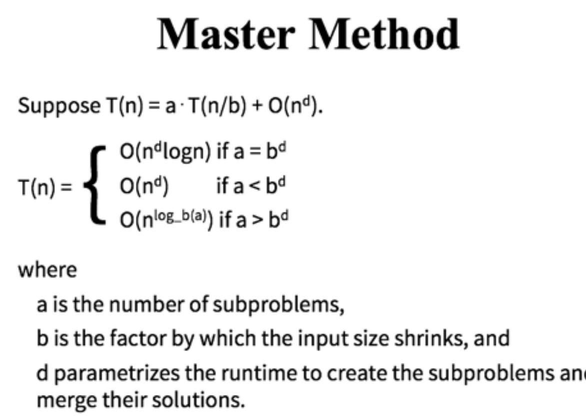
- e.g. T(n) = 4T(n/2) + O(n), runtime of the big problem represented as runtime of sub-problems

Solve the recurrence relation

- Recursion Tree Method, Iteration Method, The Master Theorem

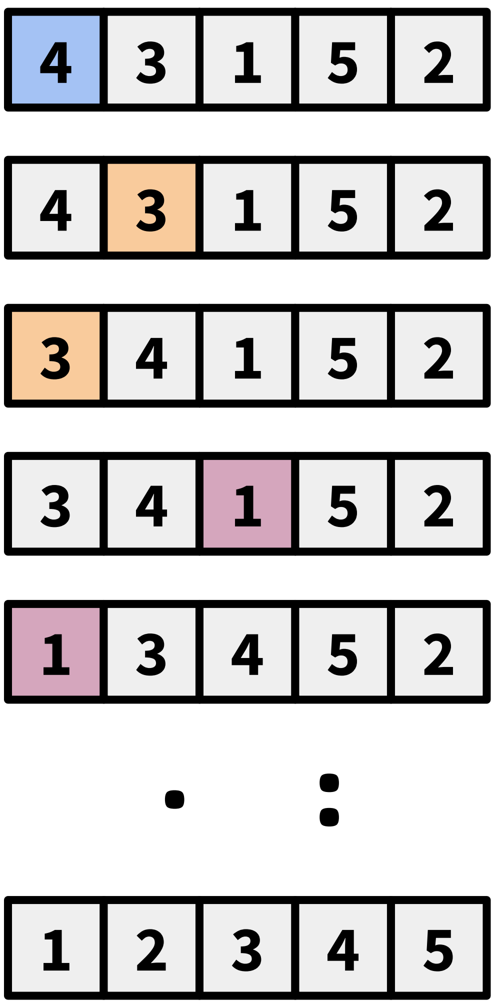






Sorting Algorithms

Insertion sort



Let's sort an unsorted list of numbers **`A`**. The sublist **`A[0:0]`** is trivially sorted.

Look at the second element, **`A[1]`**.

Insert the element into a new position such that the sublist **`A[0:1]`** is sorted.

Now look at the third element, **`A[2]`**.

Insert it such that the sublist **`A[0:2]`** is sorted.

The entire array **`A[0:4]`** is sorted.

Total work: $O(n^2)$

Mergesort

Let's use divide and conquer to improve upon insertion sort!

4	8	1	5	3	2	6	7

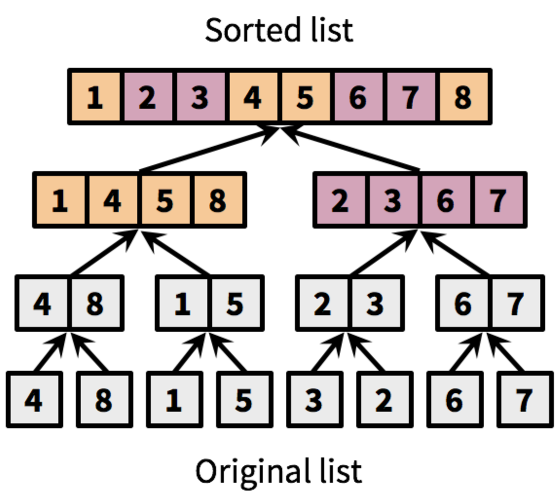
Let's sort an unsorted list of numbers **A**.

1	4	5	8	2	3	6	7

Recursively sort each half, `A[0:3]` and `A[4:7]`, separately.

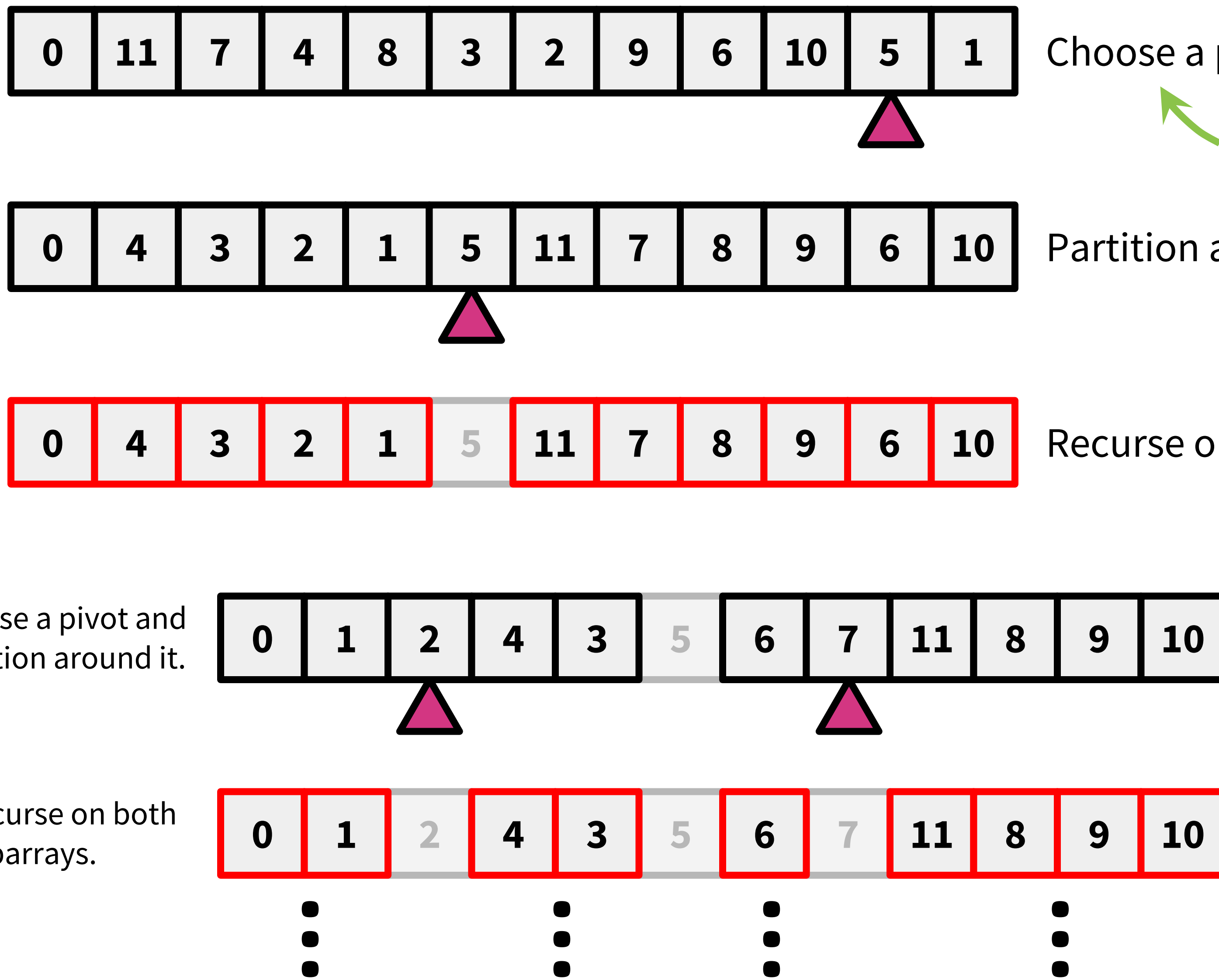
1	2	3	4	5	6	7	8

Merge the results from each half together.



Total work: `O(nlogn)`

Quicksort



Choose a pivot.

At random, a variant known as **randomized quicksort**.

Partition around it.

Recurse on both subarrays.

Choose a pivot and partition around it.

Choose a pivot and partition around it.

Recurse on both subarrays.

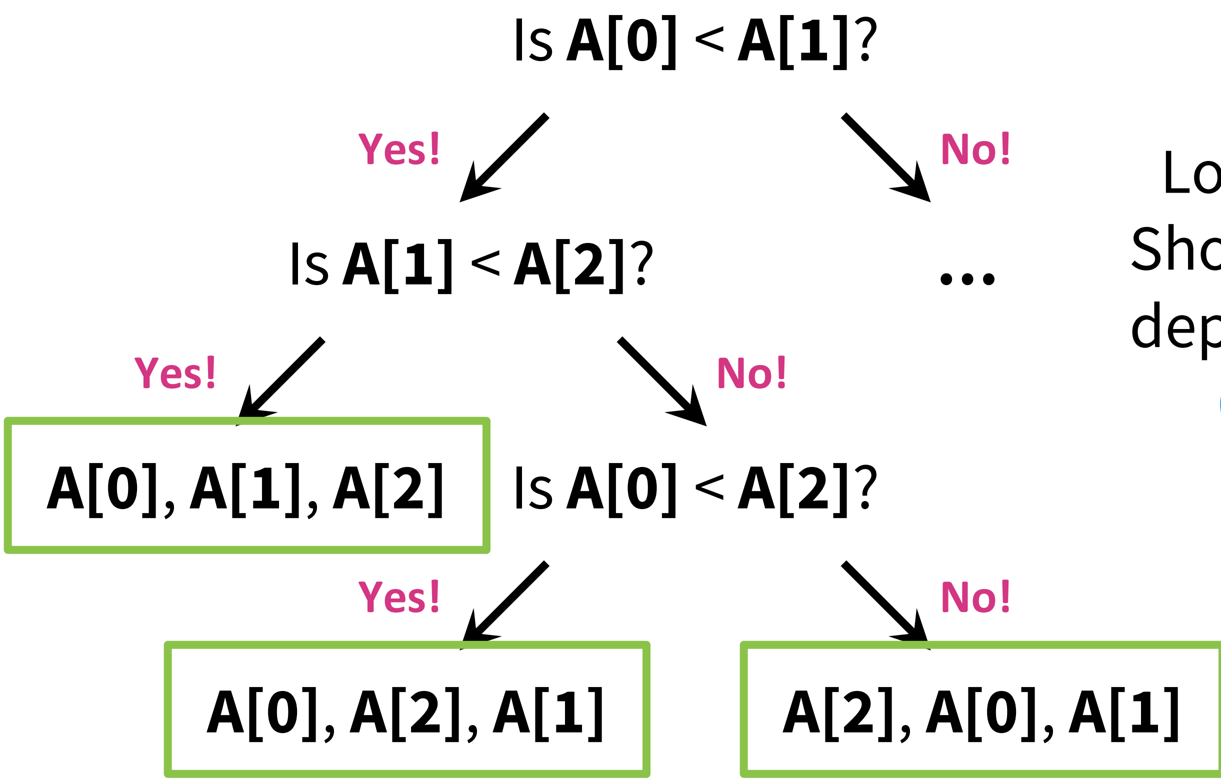
Recurse on both subarrays.

Expected: $O(n\log n)$ Worst-case: $O(n^2)$

Sorting Lower Bounds

We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree.

Suppose we want to sort three items in **A**.



Lower Bound =
Shortest possible
depth of the tree:
`O(nlogn)`

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	**0**	**3**	**1**	**1**	**3**	**1**	**0**

	0	1	2	3
`counts`	**3**	**3**	**0**	**2**

`result`	**0**	**0**	**0**	**1**	**1**	**1**	**3**	**3**

Runtime: **O(n+k)**

```
n: input size
k: maximum input number
```

Counts array: each index represents the count of the number in list A.
e.g., counts[2] stores the count of 2 in A

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A

005	210	014	125	031	555	477	095

j

2

A_j

(0, 005)	(2, 210)	(0, 014)	(1, 125)	...	(0, 095)

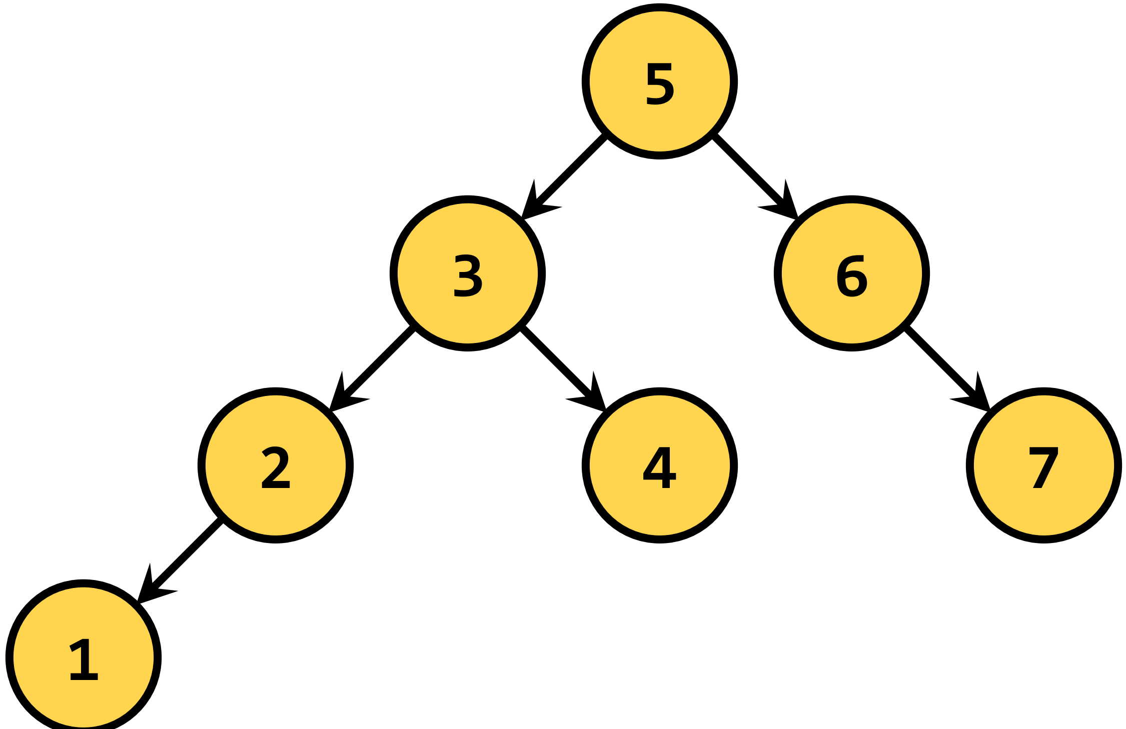
result

005	014	031	095	125	210	477	555

Runtime: `O(d(n+k))`

```
D: # of digits
n: input size
k: hashtable length
```

Binary Search Tree

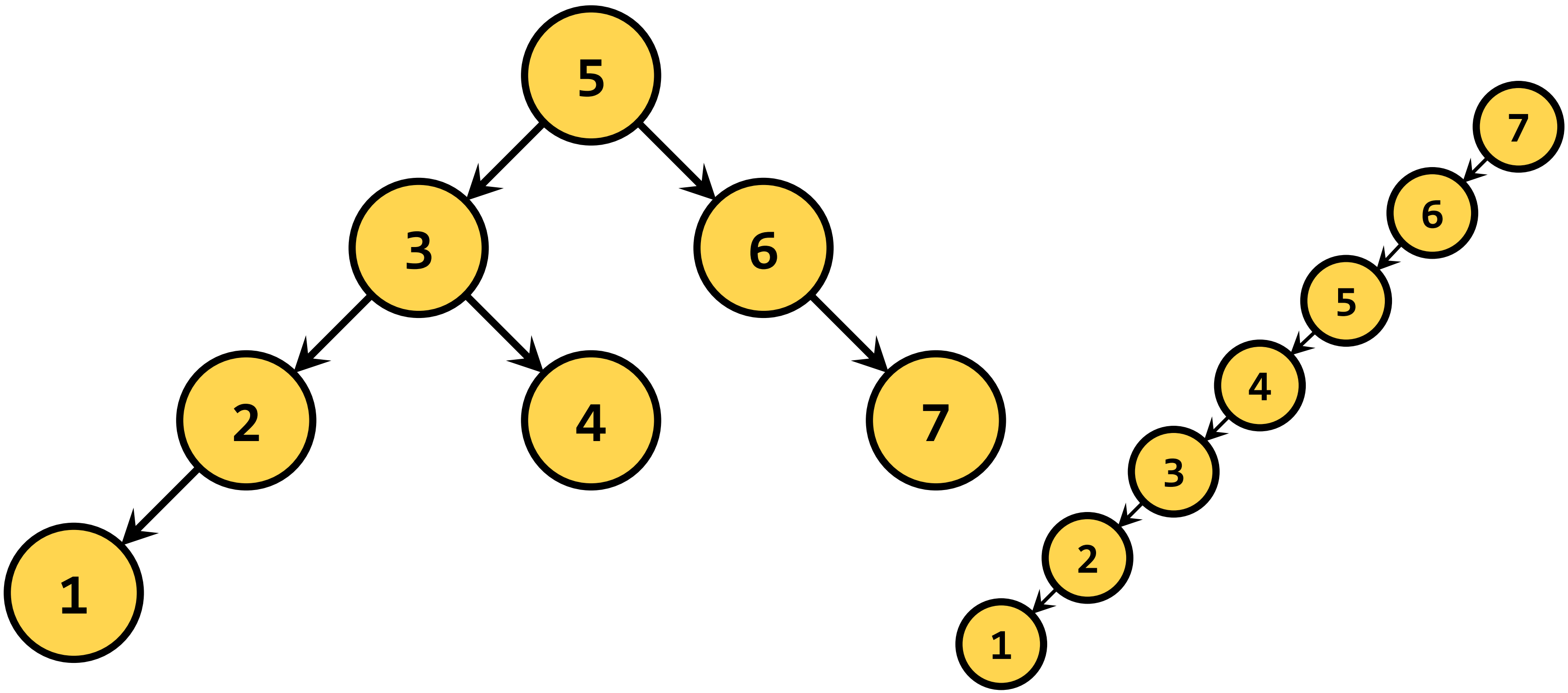


`Search, insertion, deletion`

compares the desired key to the current vertex, visiting left or right children as appropriate.

`O(depth of the tree)`

BST can be extremely unbalanced



`O(depth of the tree)`

Red-Black Trees

There exist many ways to achieve this proxy for balance, but here we'll study the **red-black tree**.

1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices v, all paths from v to its NIL descendants have the same number of **black** vertices.

We can be sure that the tree is pretty balanced as long as these proxy properties hold.

`O(depth of the tree) = O(logn)`

Randomized Algorithms

Randomized Algorithms

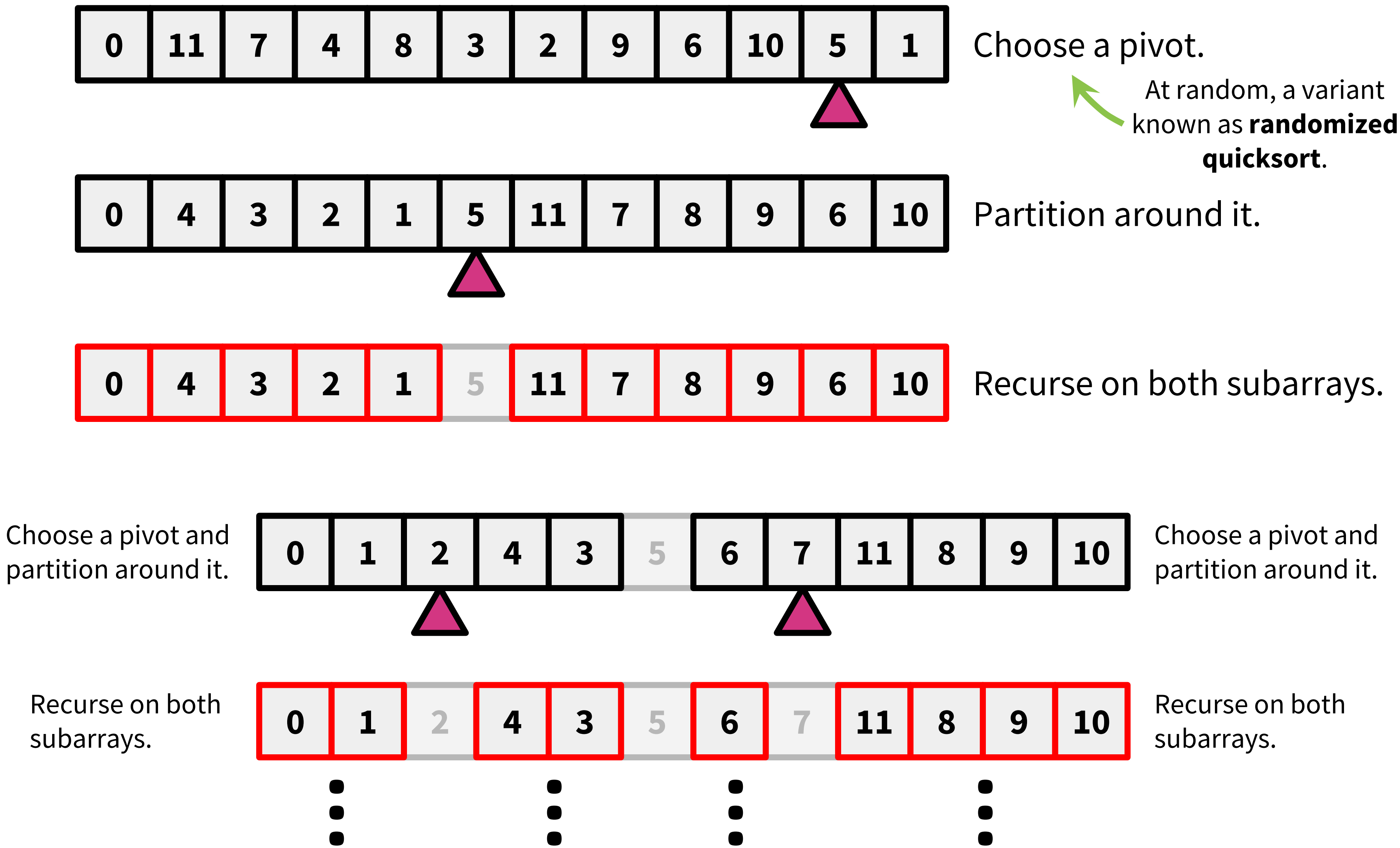
Two types of randomized algorithms

Las Vegas vs Monte Carlo

Las Vegas algorithms guarantee correctness, but not runtime.
E.g., Quick sort.

Monte Carlo algorithms guarantee runtime, but not correctness.
E.g., Karger's algorithm.

Quicksort



Expected: $O(n \log n)$ Worst-case: $O(n^2)$

Quickselect

Select the k-th smallest emelement

```
algorithm quickselect(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime

Expected: $O(n)$ Worst-case: $O(n^2)$

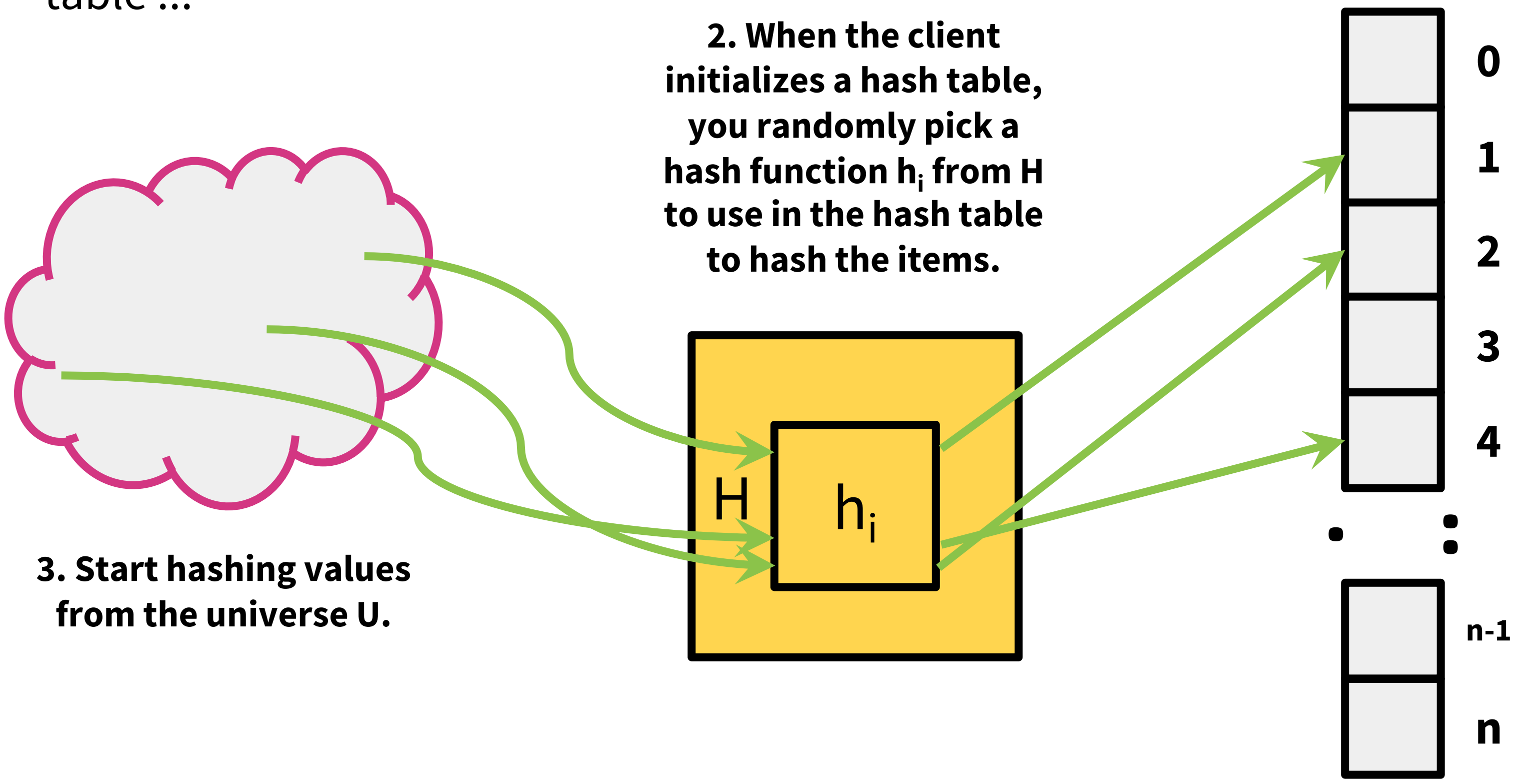
Think of this as the adversary chooses the randomness.

Hash Tables

Let's say you wanted to implement a hash table ...

1. You choose your set of hash functions H, likely a universal hash family like H = mod p mod n.

2. When the client initializes a hash table, you randomly pick a hash function h_i from H to use in the hash table to hash the items.



3. Start hashing values from the universe U.

Universal Hash Family

How can we design hash function set H of reasonable size?

Recall the bound that allowed us to achieve this guarantee:

E[number of items in u_x's bucket] = $\sum_y P[h(u_x) = h(u_y)] = 1 + \sum_{y \neq x} P[h(u_x) = h(u_y)]$

$= 1 + \sum_{y \neq x} 1/n = 1 + (n-1/n) \leq 2$

This bound is so important, there's a special name for sets H that satisfy it.

A **hash family** is a fancy name for a set of hash functions.

A **universal hash family** describes a set of hash functions that satisfy the bound: $P_{h \in H}[h(u_x) = h(u_y)] \leq 1/n$, i.e., the probability of collision is bounded by 1/n.

The exhaustive set of hash functions is an example of a universal hash family but, as discussed previously, it's too big to be practical.

Randomized Algorithms

Two types of randomized algorithms

Las Vegas vs Monte Carlo

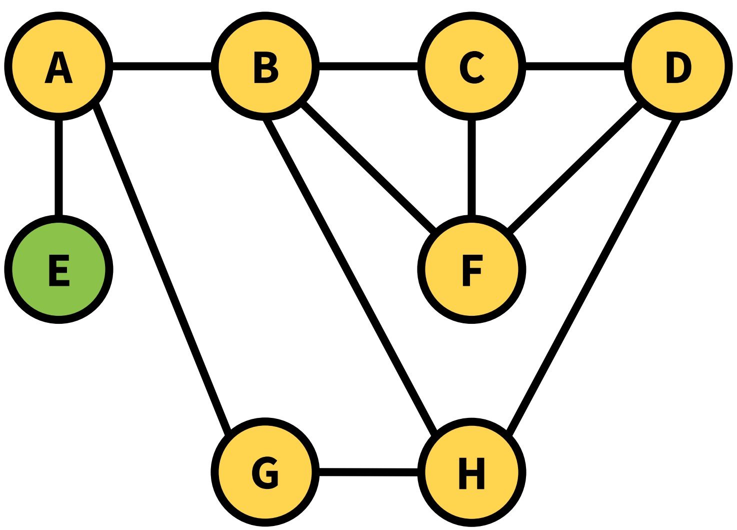
Las Vegas algorithms guarantee correctness, but not runtime.
E.g., Quick sort.

Monte Carlo algorithms guarantee runtime, but not correctness.
E.g., Karger's algorithm.

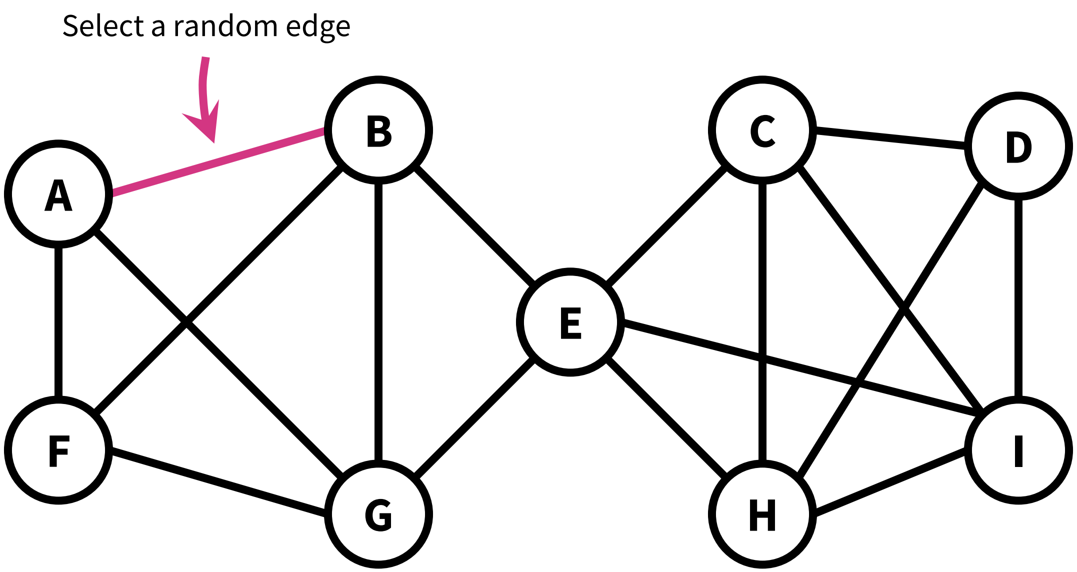
Karger's Algorithm and Minimum Cut

A **global minimum cut** is a cut that has the fewest edges possible crossing it.

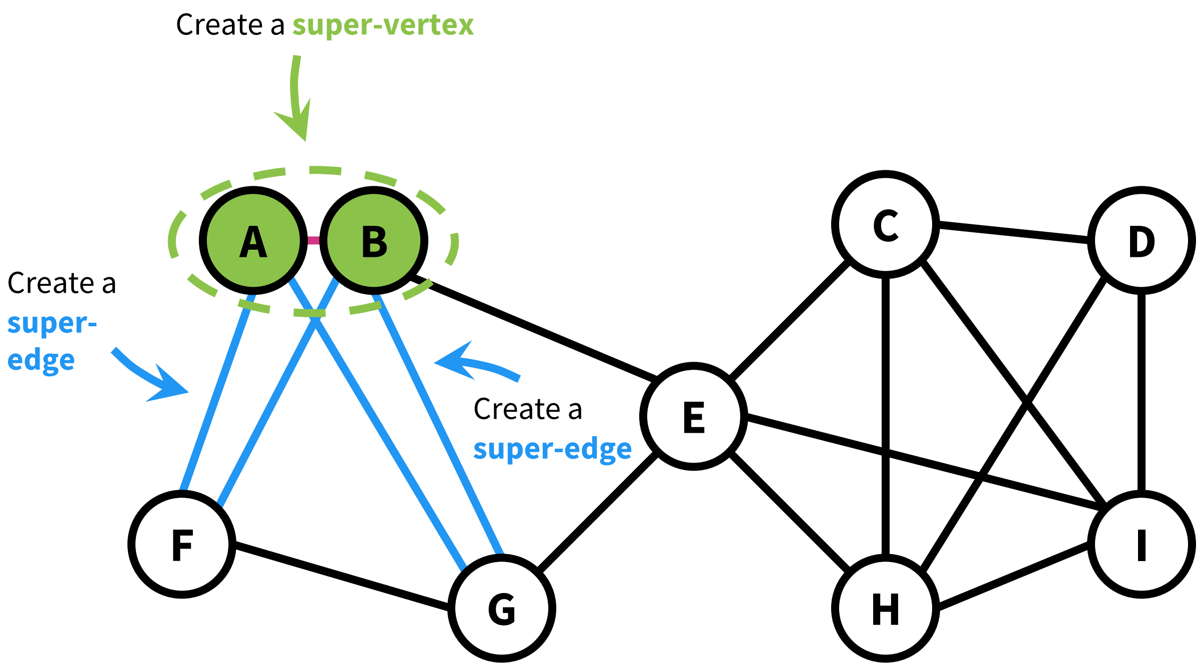
e.g. The global minimum cut is "{A, B, C, D, F, G, H} and {E}".



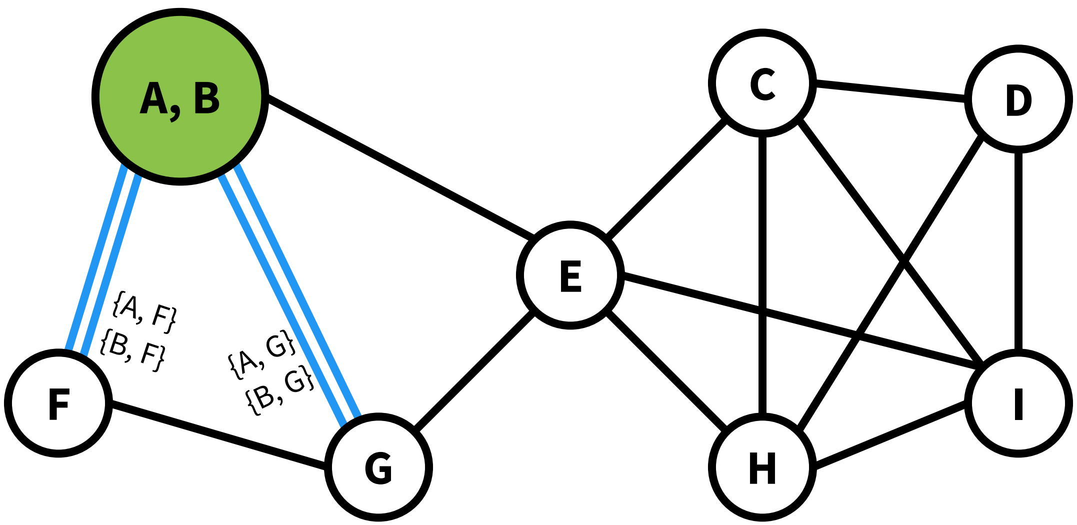
Karger's Algorithm



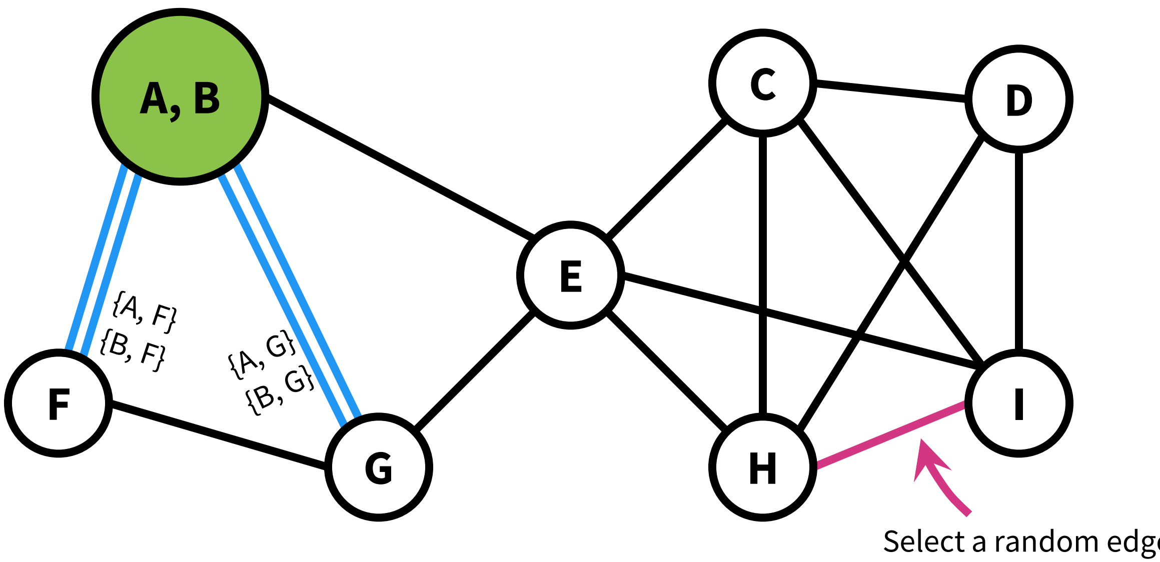
Karger's Algorithm



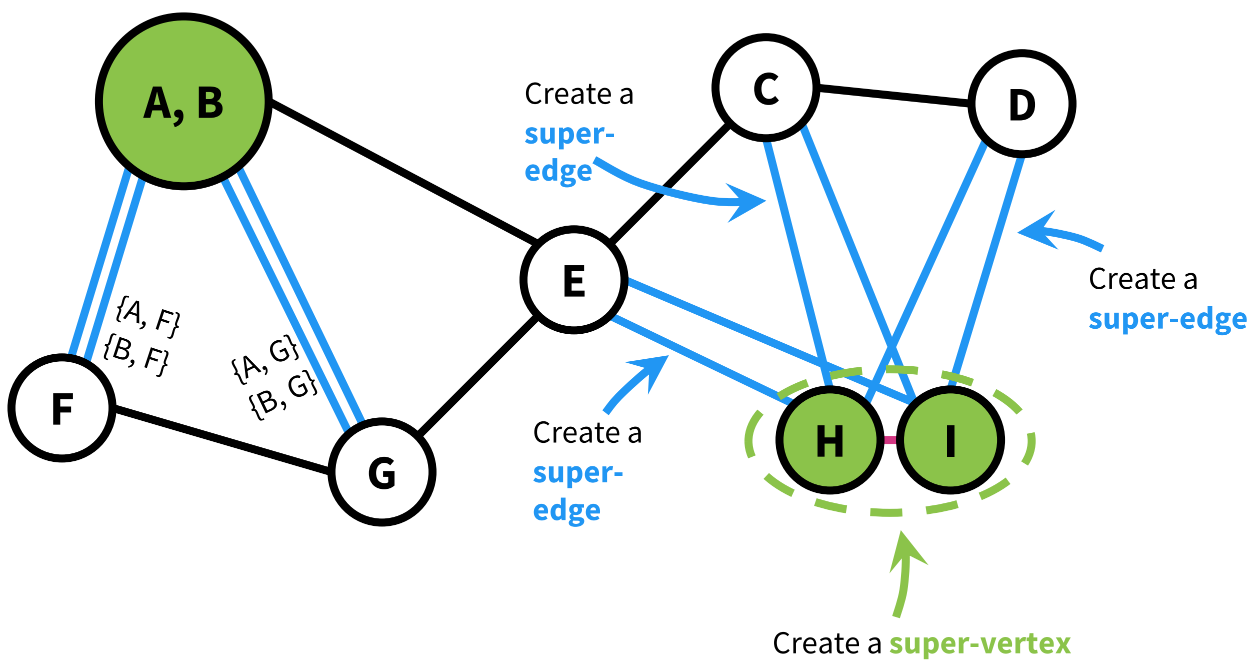
Karger’s Algorithm



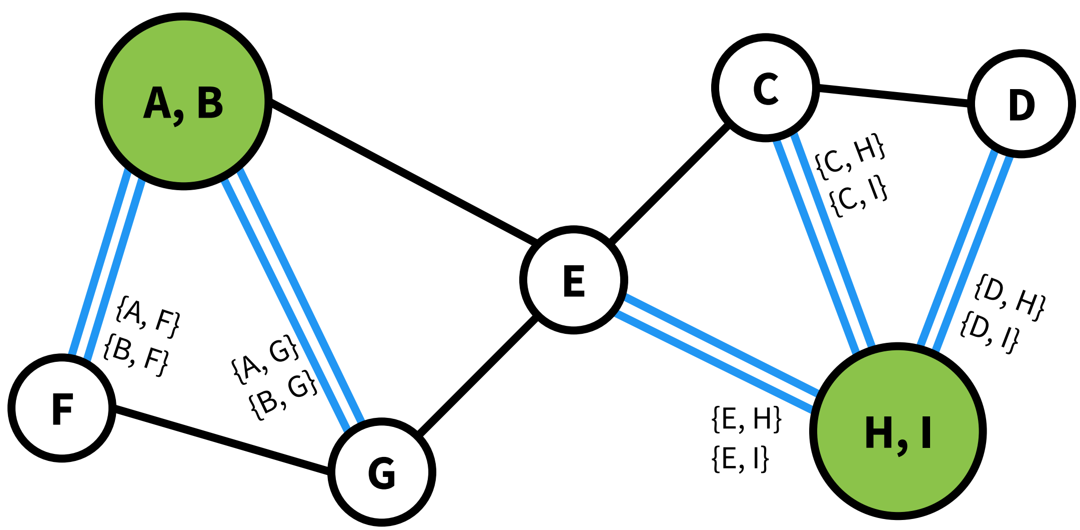
Karger's Algorithm



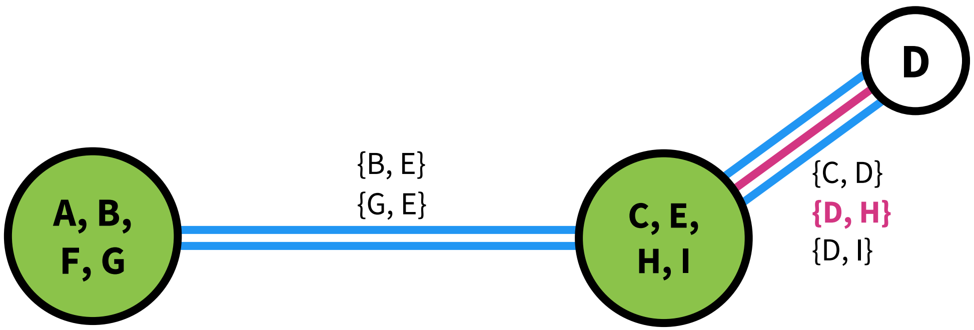
Karger's Algorithm



Karger's Algorithm



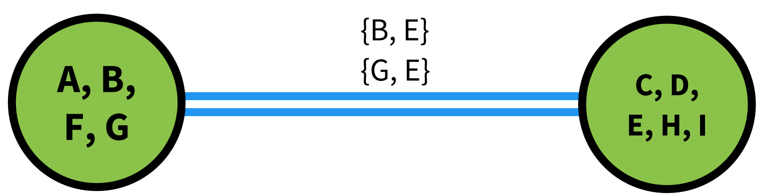
Karger's Algorithm



Karger’s Algorithm

The minimum cut is given by the remaining super-vertices.

e.g. The cut is “{A, B, F, G} and {C, D, E, H, I}”; the edges that cross this cut are {B, E} and {G, E}.



Karger's Algorithm

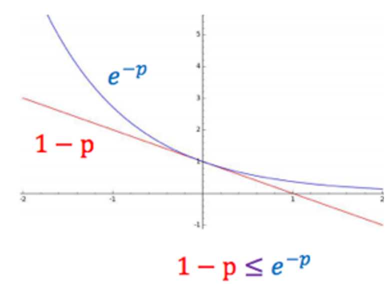
Probability of success = 1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$
T = (nC2) ln (1/0.1) times

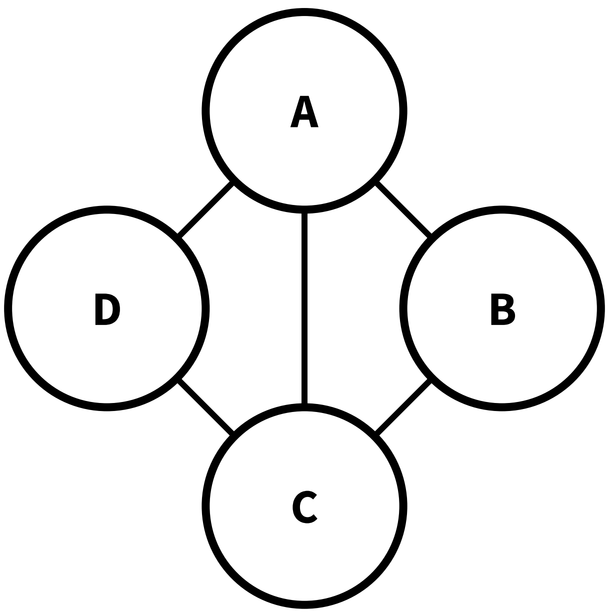
Suppose we want to find the min-cut with probability p.
Then we must repeat Karger **T = (nC2) ln (1/(1-p)) times**.

Graph Algorithms

Undirected Graphs

An undirected graph has vertices and edges.

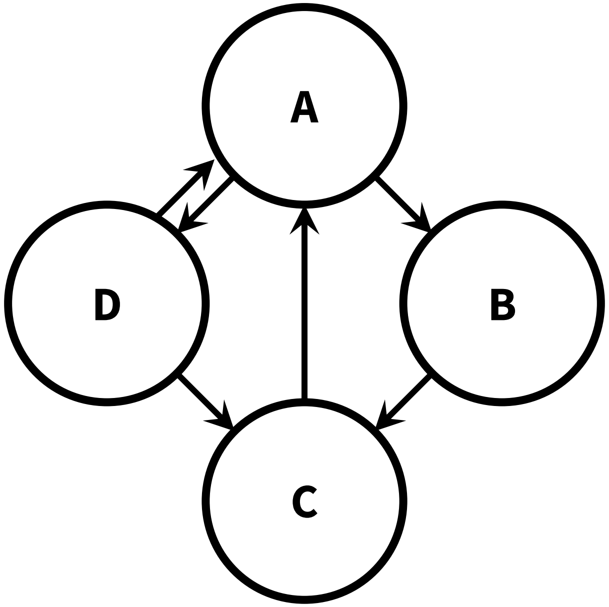
- V is the set of vertices and E is the set of edges.
- Formally, an undirected graph is G = (V, E).
- e.g. V = {A, B, C, D} and E = { {A, B}, {A, C}, {A, D}, {B, C}, {C, D} }



Directed Graphs

A directed graph has vertices and **directed** edges.

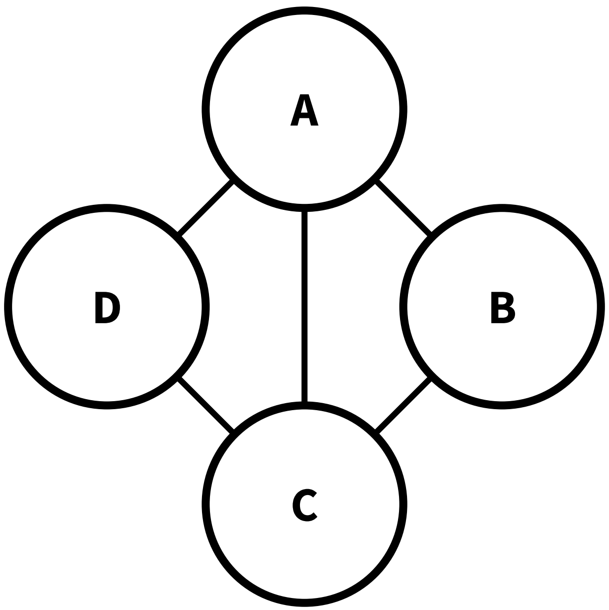
- V is the set of vertices and E is the set of directed edges.
- Formally, a directed graph is G = (V, E)
- e.g. V = {A, B, C, D} and E = { [A, B], [A, D], [B, C], [C, A], [D, A], [D, C] }



Graph Representations

How do we represent graphs?

(1) Adjacency matrix



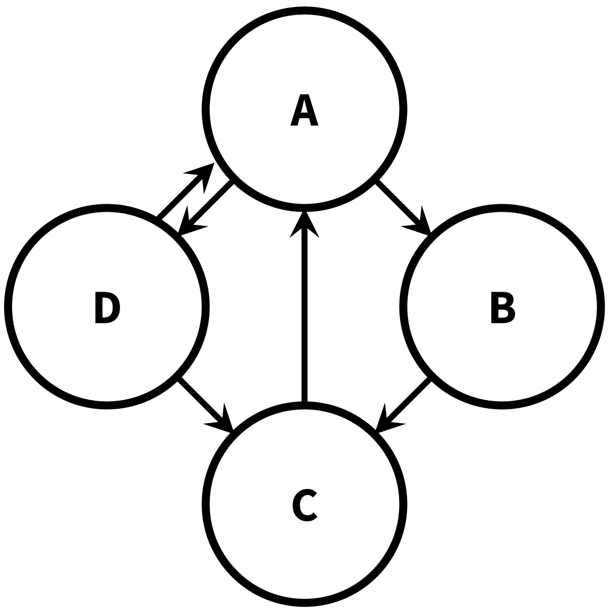
$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Symmetric matrix

Graph Representations

How do we represent graphs?

(1) Adjacency matrix



destination

source

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

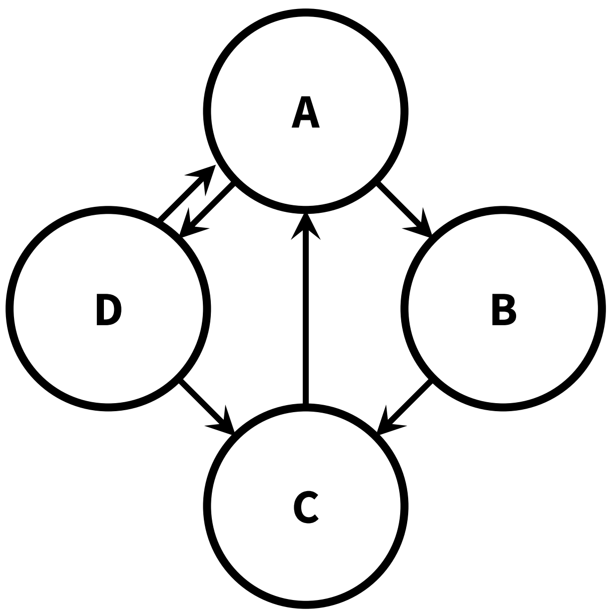
Unsymmetric matrix

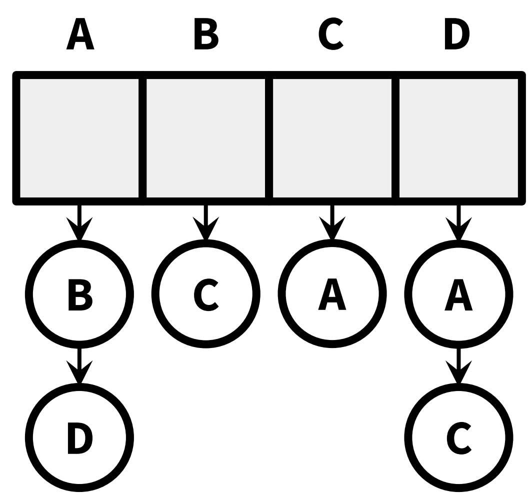
Graph Representations

How do we represent graphs?

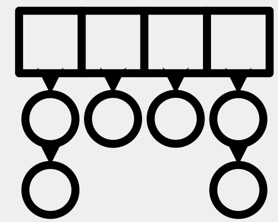
(1) Adjacency matrix

(2) Adjacency list





Graph Representations

	Adjacent matrix	Adjacent list
For G = (V, E)	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	
Edge Membership Is e = [u, v] in E?	O(1)	O(deg(u))
Neighbor Query What are the neighbors of u?	O(|V|)	O(deg(u))
Space requirements	$O(\|V\|^2)$	O(|V|+|E|)

Generally, better for sparse graphs.

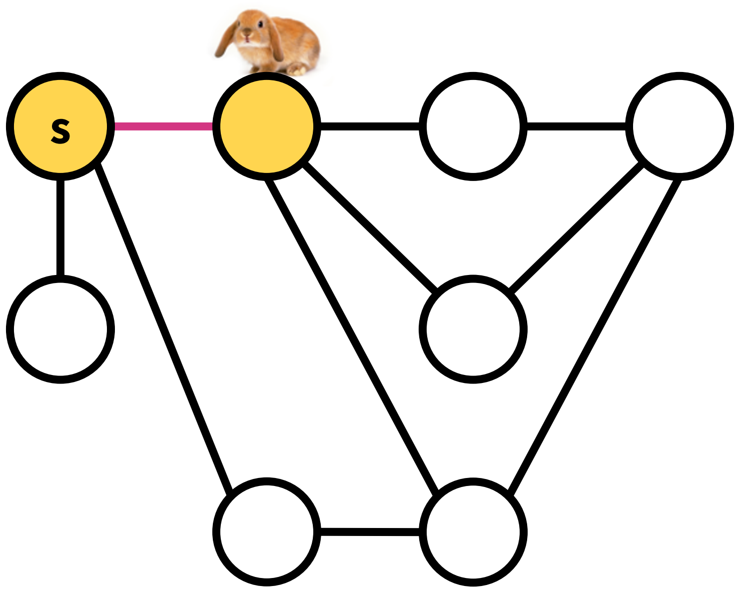
We'll assume this representation, unless otherwise stated.

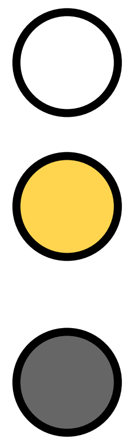
Explain with directed graph; relationship between the two representations; an example in e-commerce

Depth-First Search

An analogy

A smart bunny exploring a labyrinth with chalk (to mark visited destinations) and thread (to retrace steps).



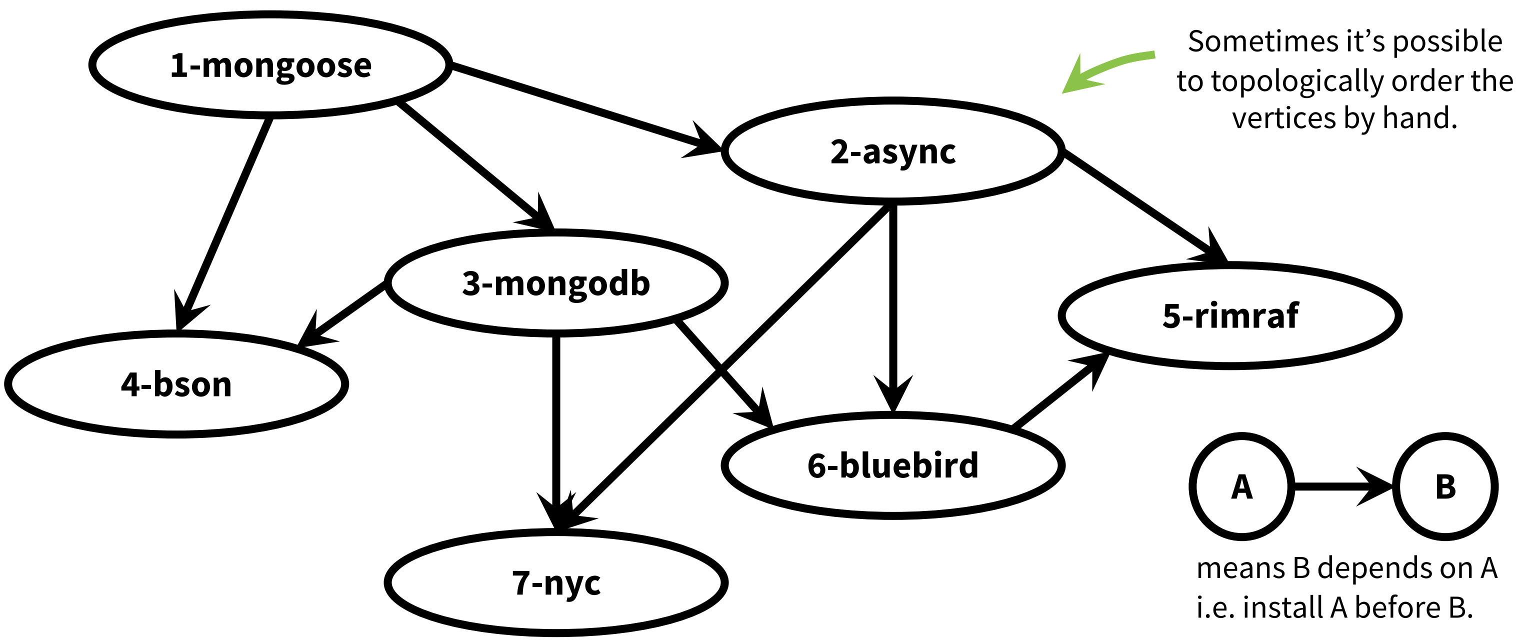


DFS for Topological Ordering

Application of DFS: Given a package dependency graph, in what order should packages be installed?

DFS produces a **topological ordering**, which solves this problem.

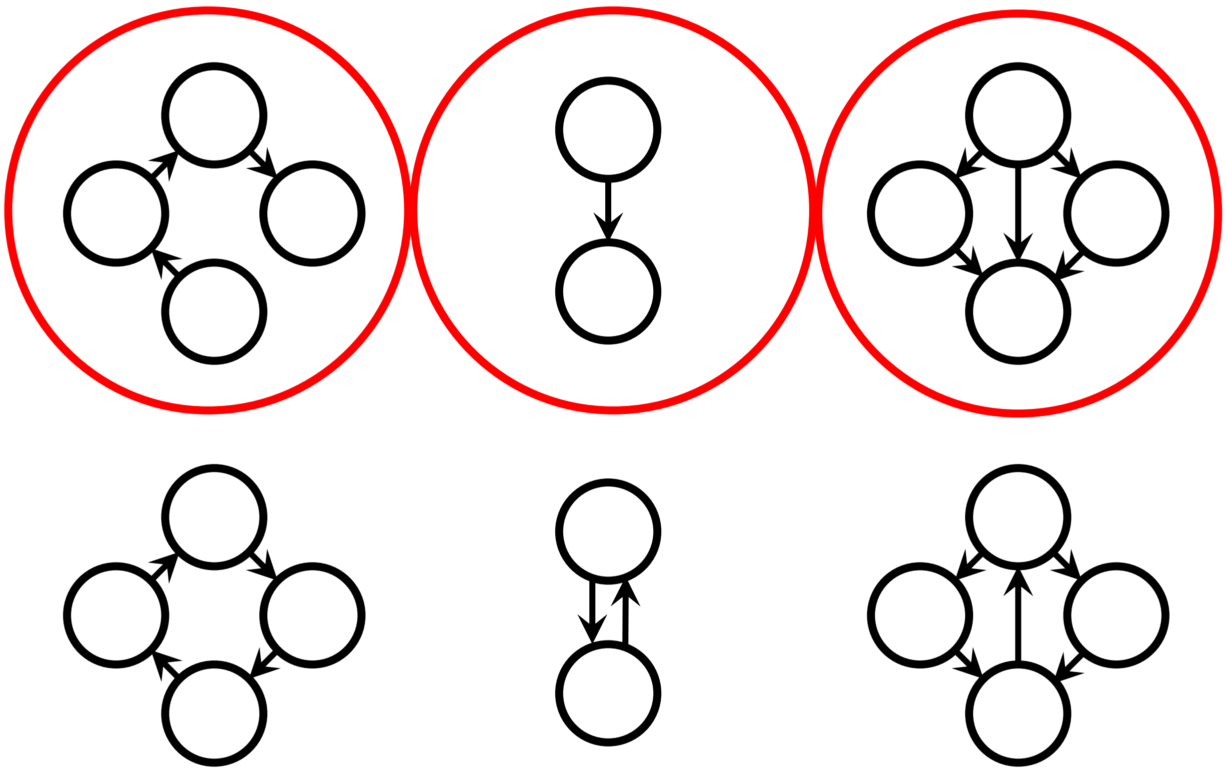
Definition: The topological ordering of a DAG is an ordering of its vertices such that for every directed edge (u, v) ∈ E, u precedes v in the ordering.



Aside: Directed Acyclic Graphs

A dependency graph is an instantiation of a directed acyclic graph (DAG) i.e. a directed graph with no directed cycles.

Which of these graphs are valid DAGs? 🤔

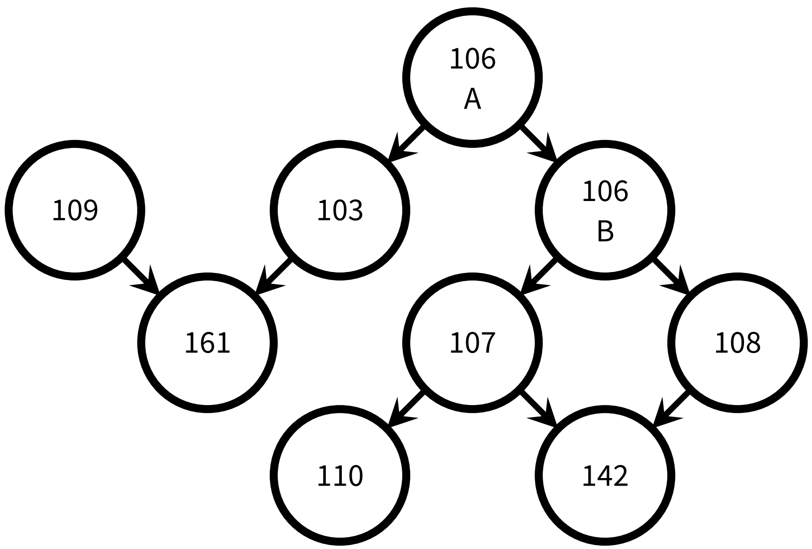


Topological Ordering

During DFS, once a node is closed, put it into reversed_topological_list.

To compute the topological ordering in general, reverse the order of `reversed_topological_list`.

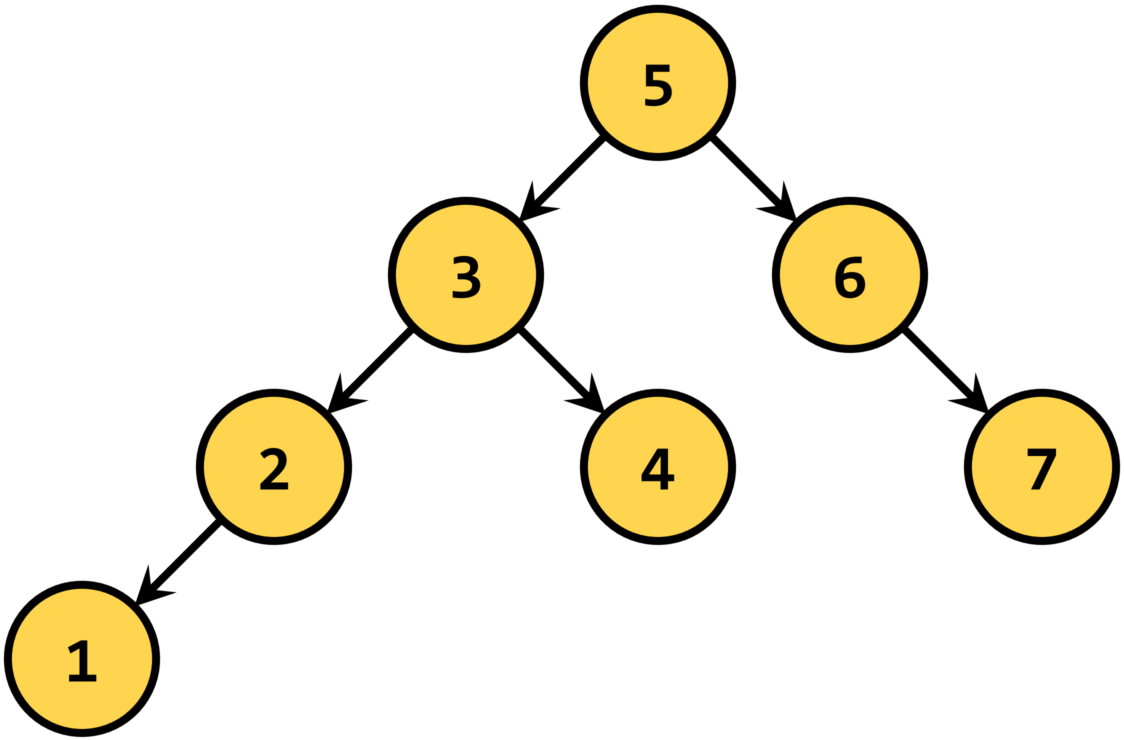
e.g. Finding an order to take courses that satisfies prerequisites.



Reversed_topological_list = 110, 142, 107, 108, 106B, 161, 103, 106A, 109

In-Order Traversal of BSTs

Application of DFS: Given a BST, output the vertices in order.

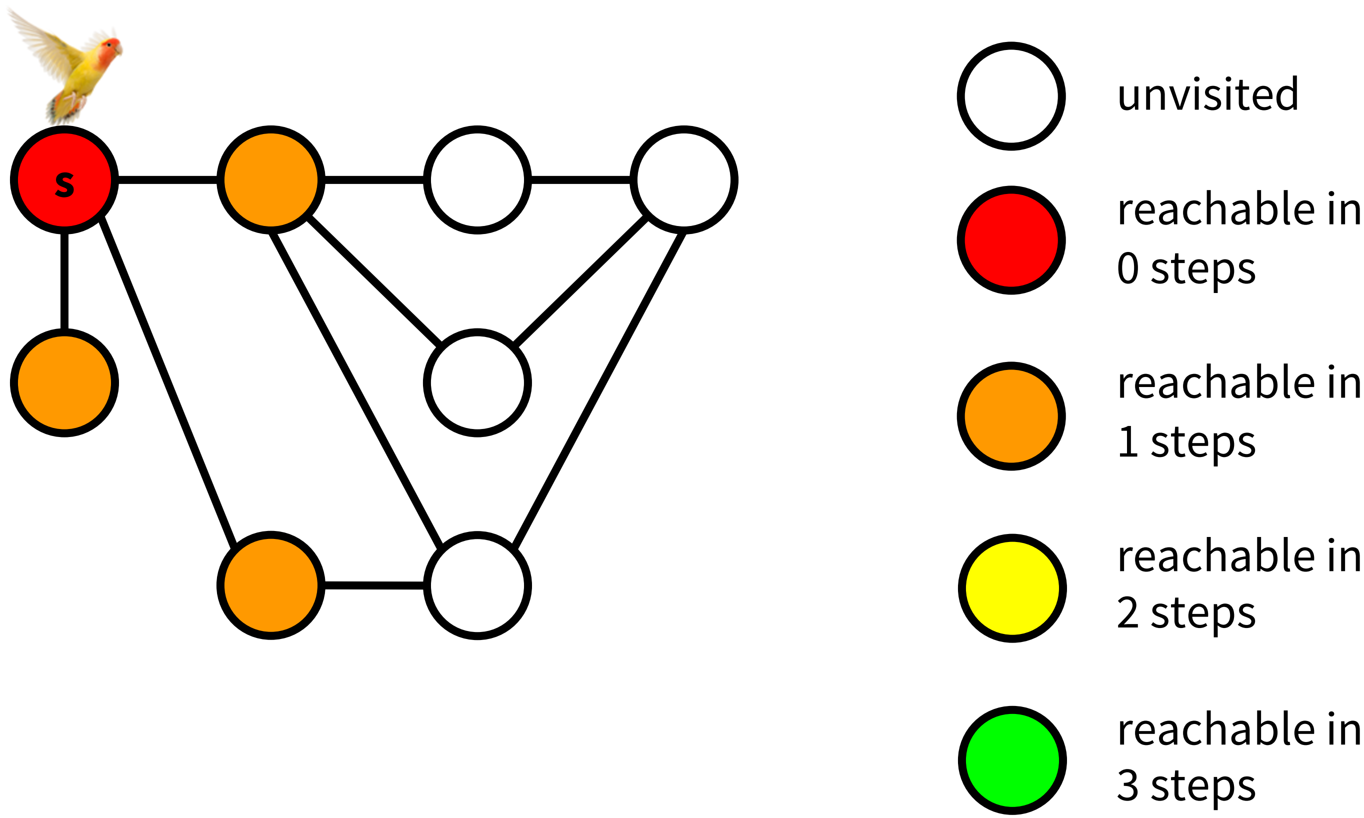


In-order traversal: visit left subtree -> visit the node -> visit the right tree

Breadth-First Search

An analogy

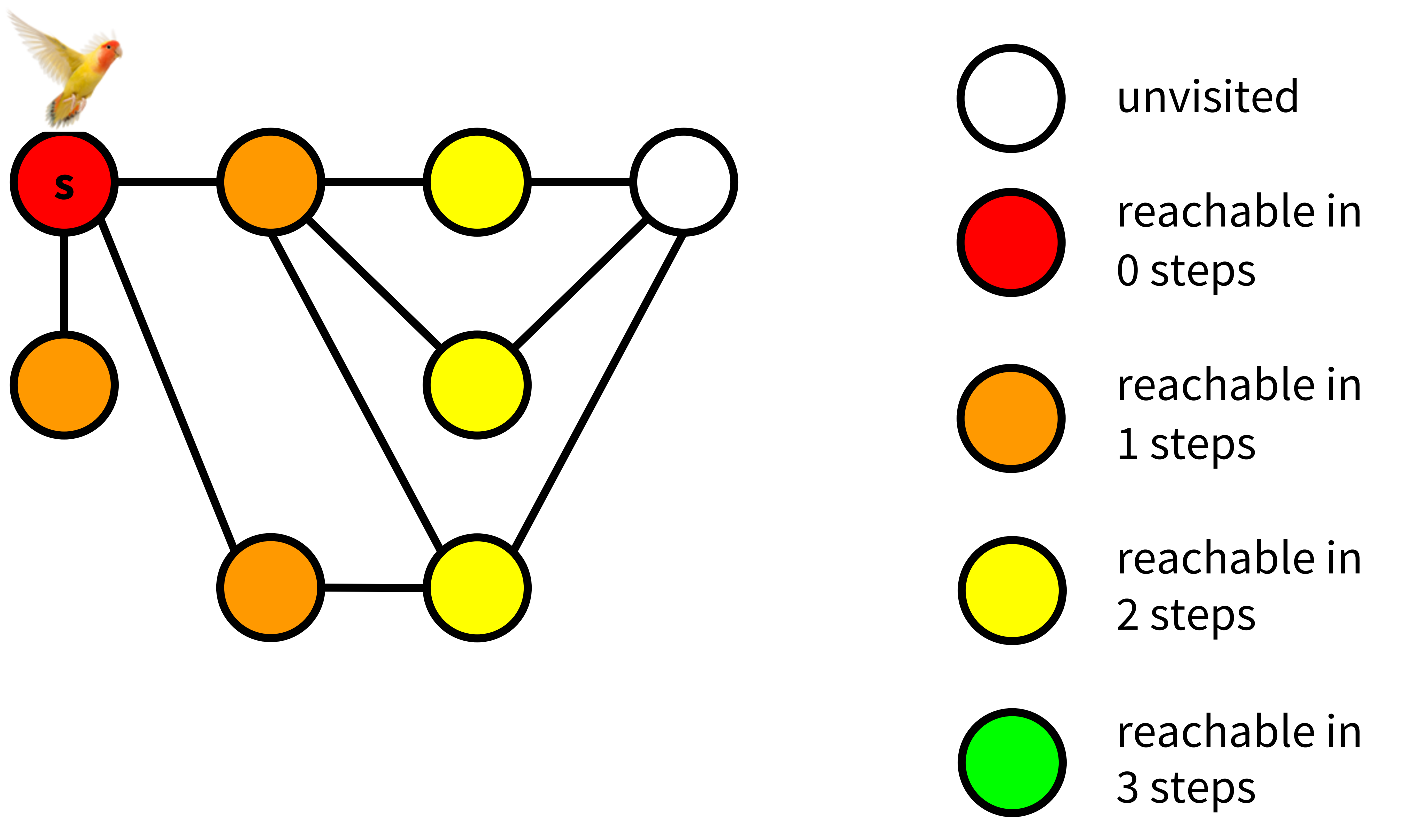
A bird exploring a labyrinth from above (with a bird's eye view).



Breadth-First Search

An analogy

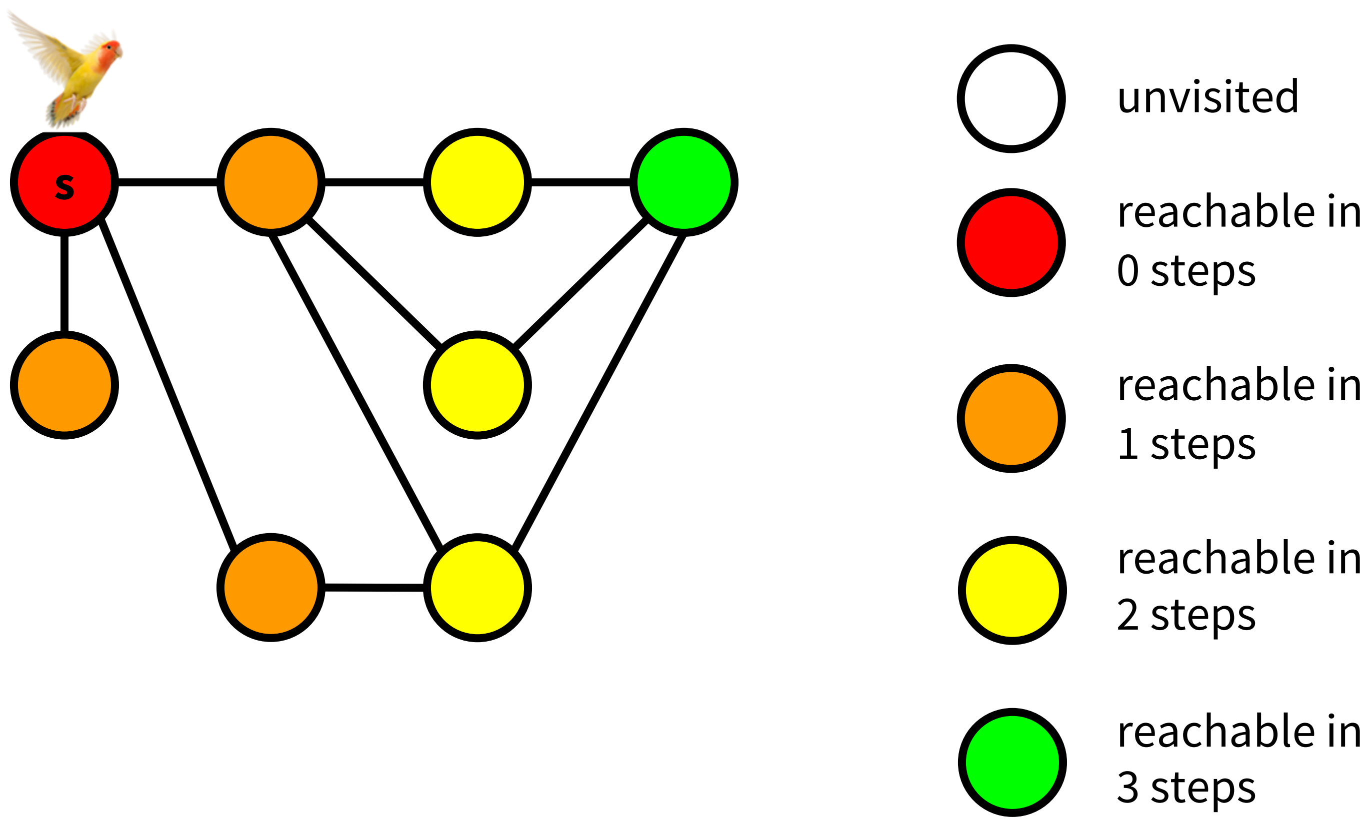
A bird exploring a labyrinth from above (with a bird's eye view).



Breadth-First Search

An analogy

A bird exploring a labyrinth from above (with a bird's eye view).



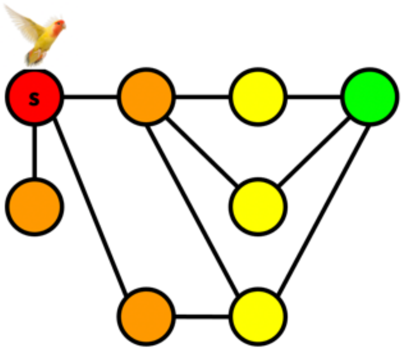
Shortest Path

Application of BFS: How long is the shortest path between vertices u and v?

Call `bfs(u)`.

For all vertices in `L[i]`, the shortest path between u and these vertices has length i.

If v isn't in `L[i]` for any i, then it's unreachable from u.



For example, by calling bfs(s) on node s, we have the following lists:

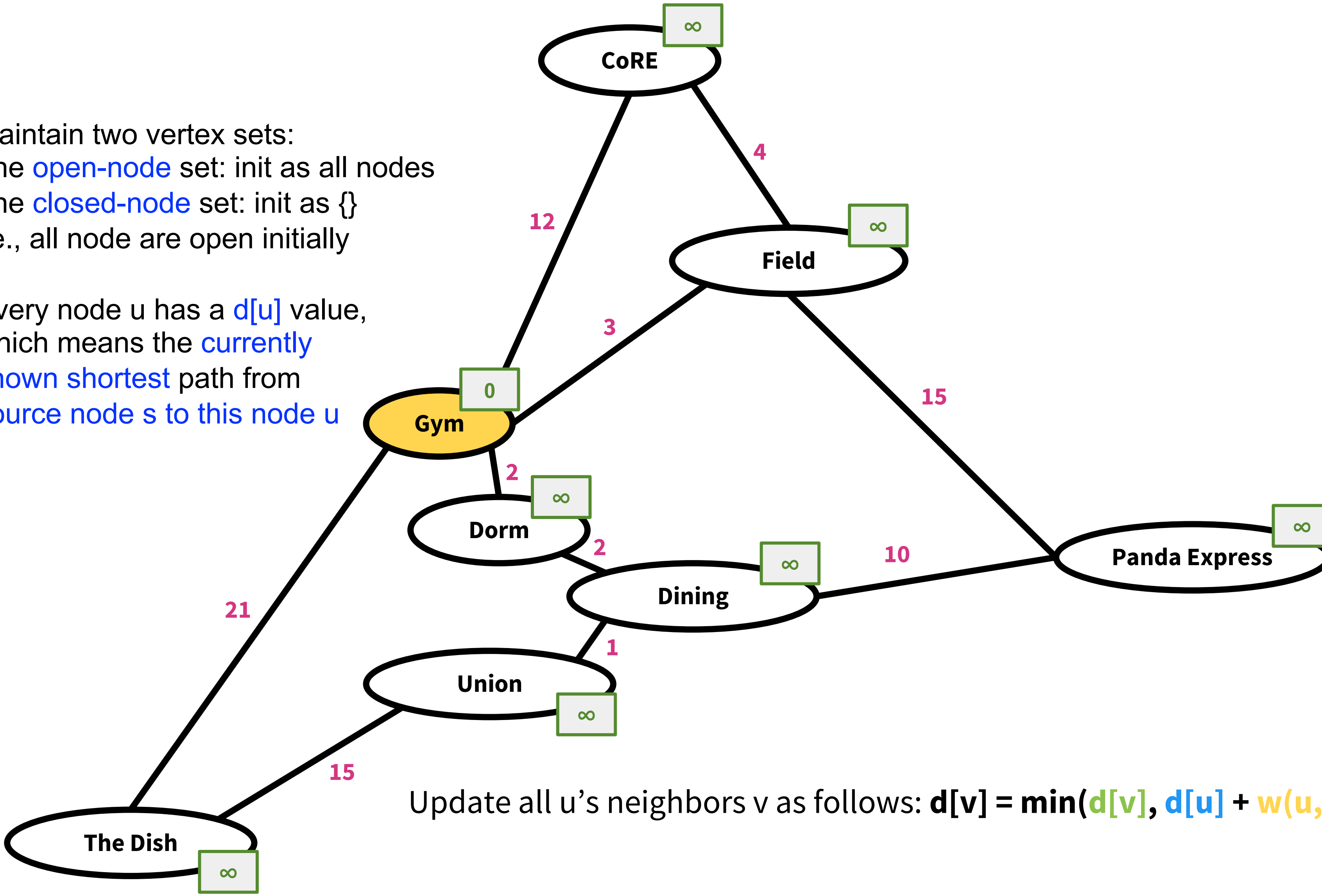
L[0] = {s} // Initialize
L[1] = {1, 2, 3} // Take out s from L[0], visit its (unvisited) neighbors and put them in L[1]
L[2] = {4, 5, 6} // Take out 1, 2, 3 from L[1], visit their (unvisited) neighbors and put them in L[2]
L[3] = {7} // Take out 4, 5, 6 from L[2], visit their (unvisited) neighbors and put them in L[3]
L[4] = {} // Take out 7 from L[3], visit its (unvisited) neighbors, but there is no unvisited neighbor anymore, stop.

We know the shortest path between s and node 5 has length 2, because node 5 appears in L[2].

Dijkstra's Algorithm

Maintain two vertex sets:
The open-node set: init as all nodes
The closed-node set: init as {}
i.e., all node are open initially

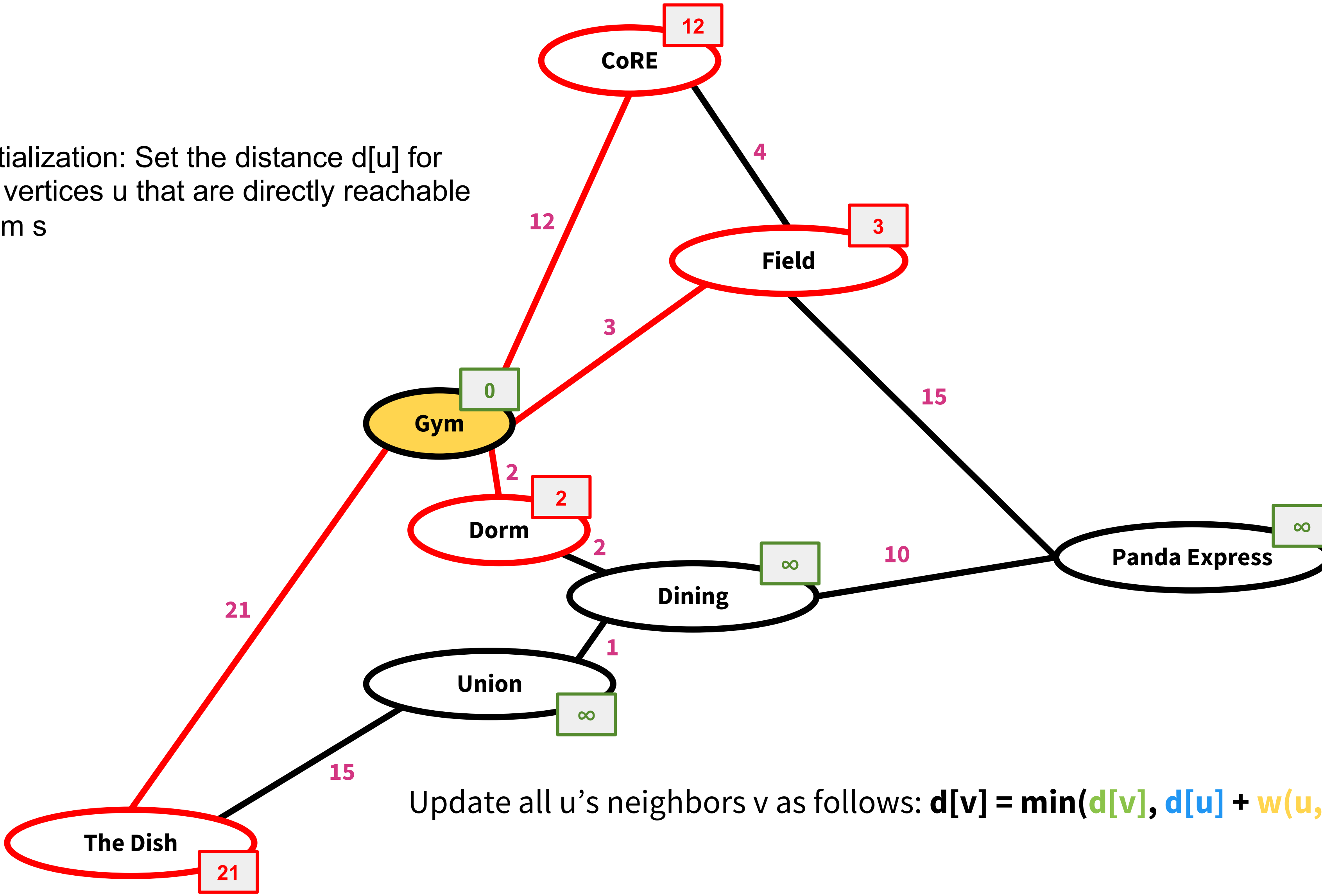
Every node u has a d[u] value, which means the currently known shortest path from source node s to this node u



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

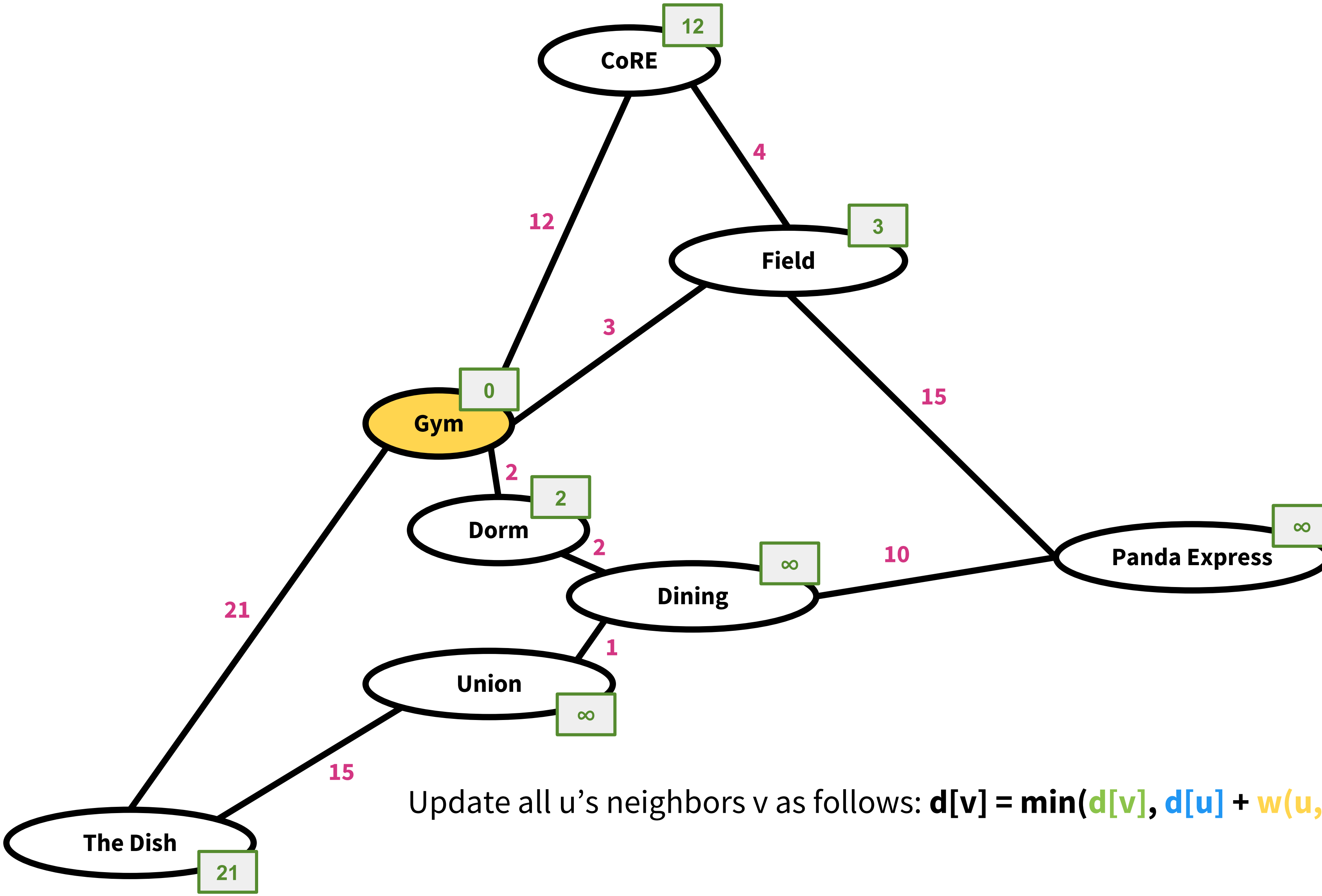
Dijkstra's Algorithm

Initialization: Set the distance d[u] for all vertices u that are directly reachable from s



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

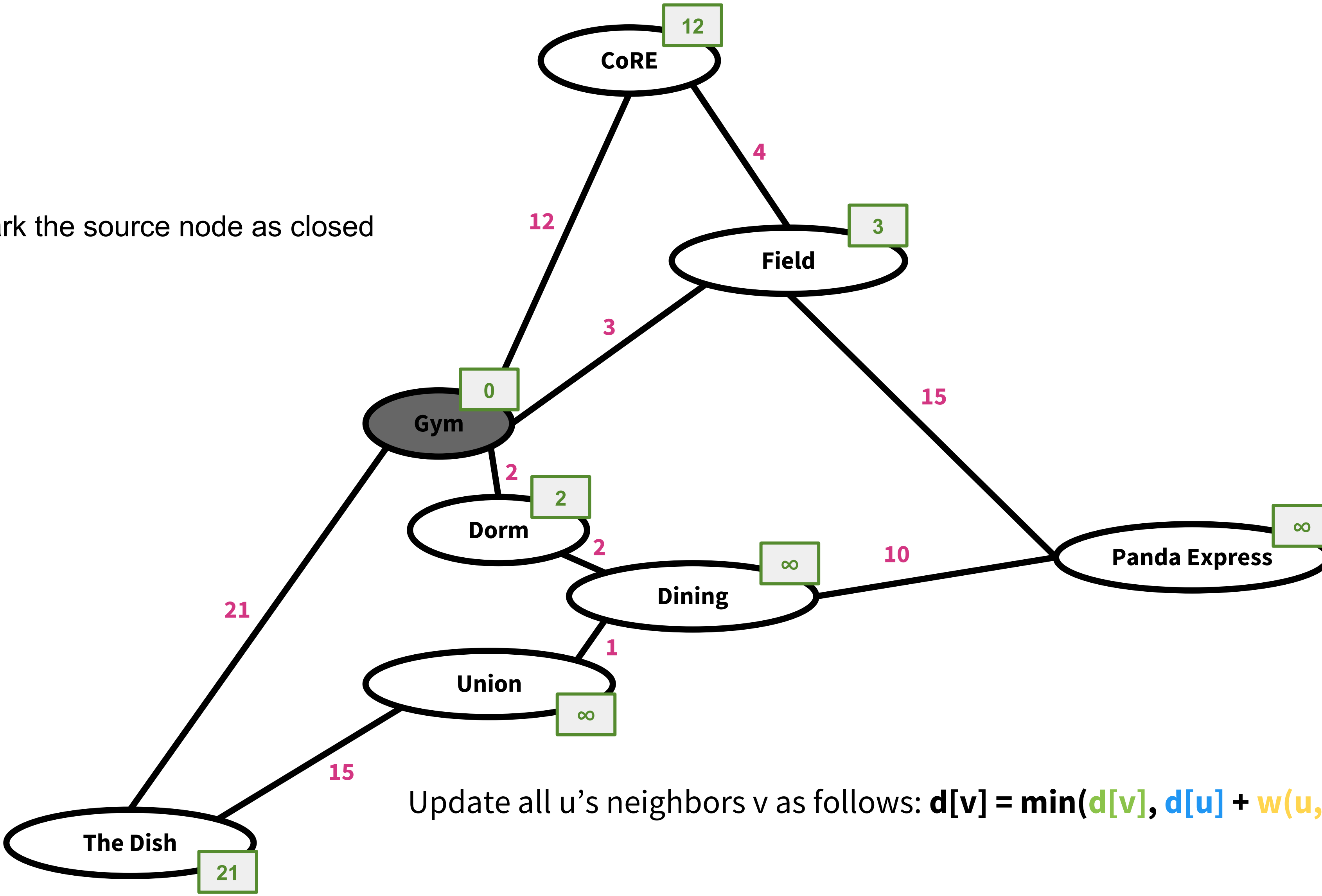
Dijkstra's Algorithm



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

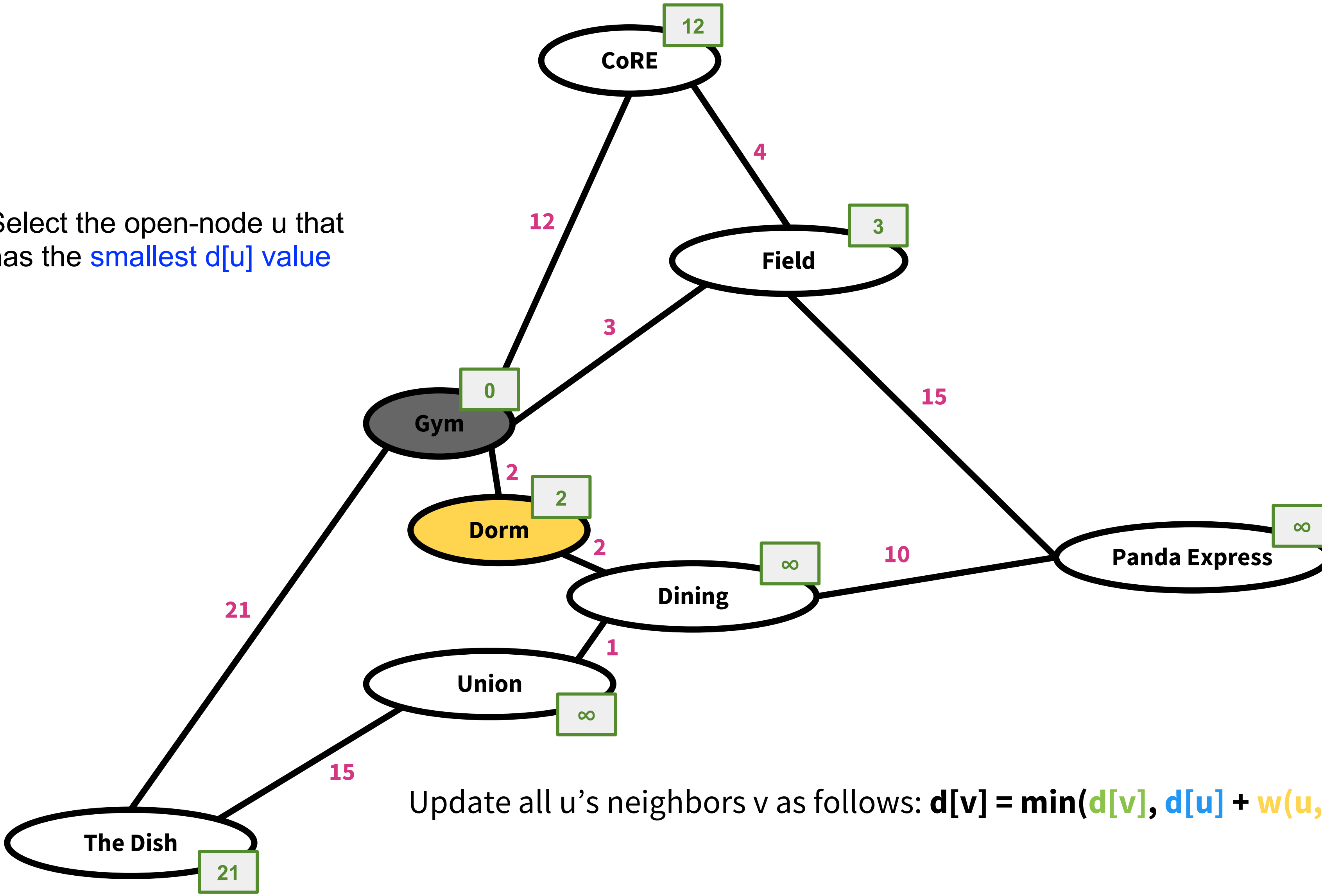
Mark the source node as closed



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

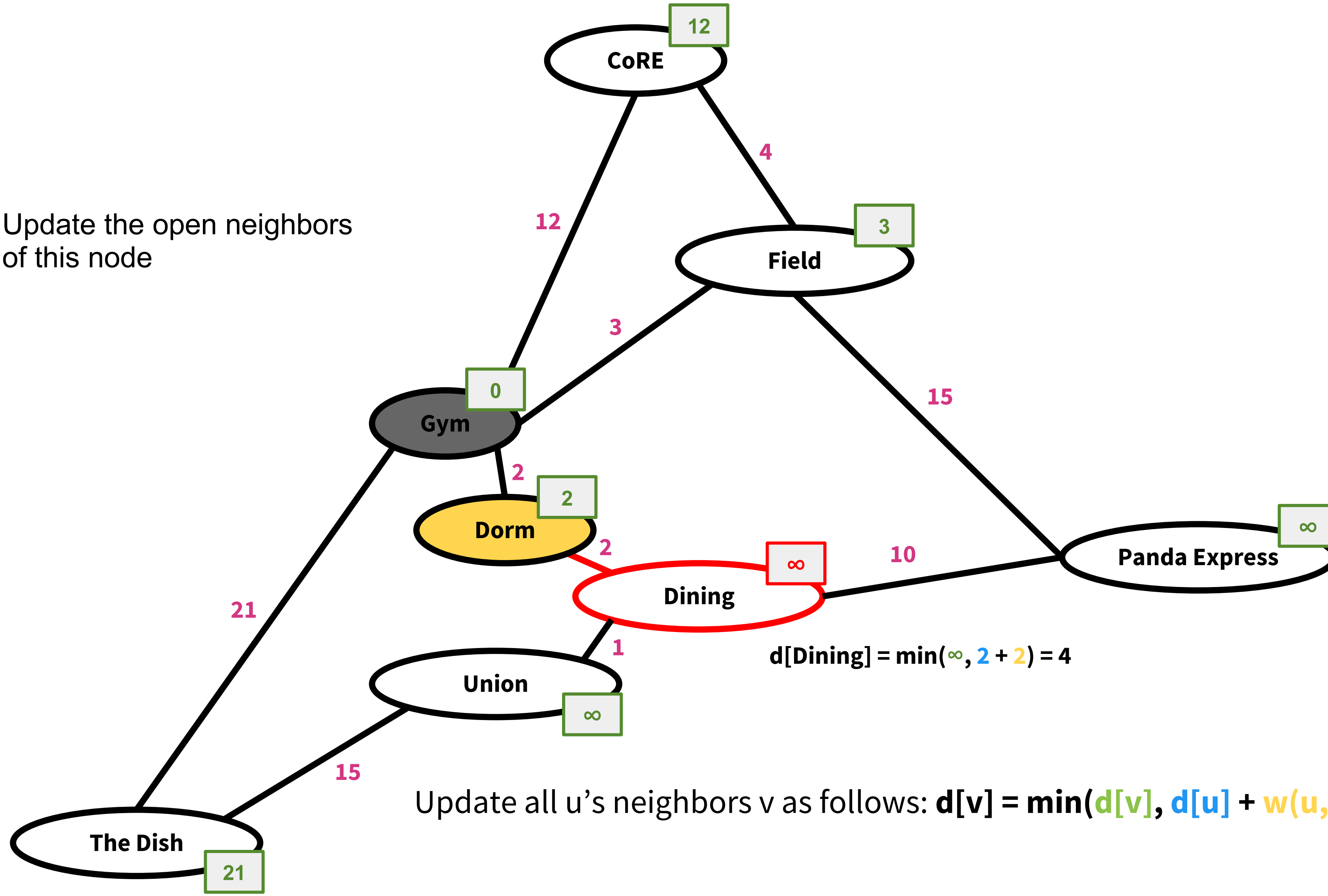
Select the open-node u that has the smallest d[u] value



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

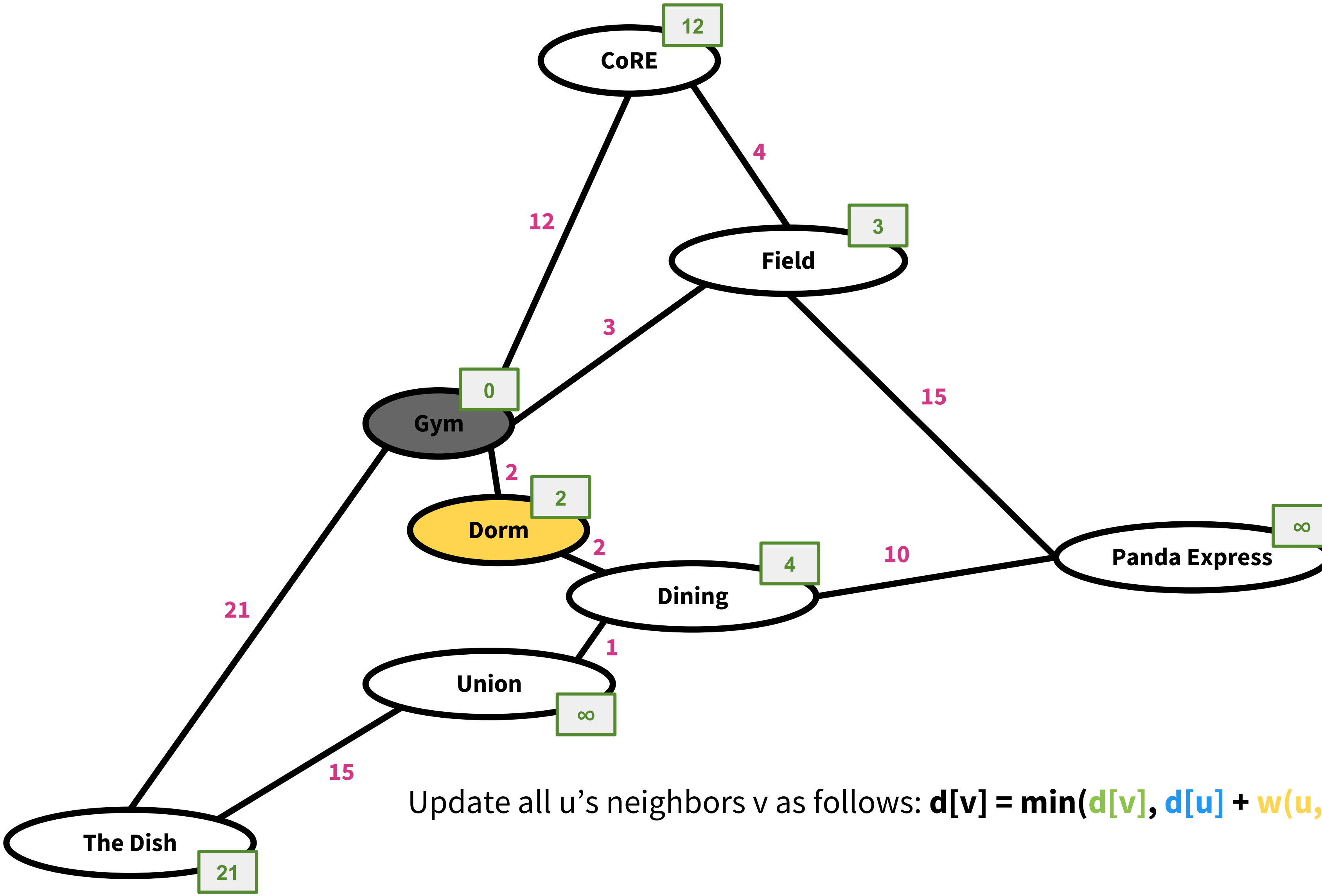
Dijkstra's Algorithm

Update the open neighbors of this node



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

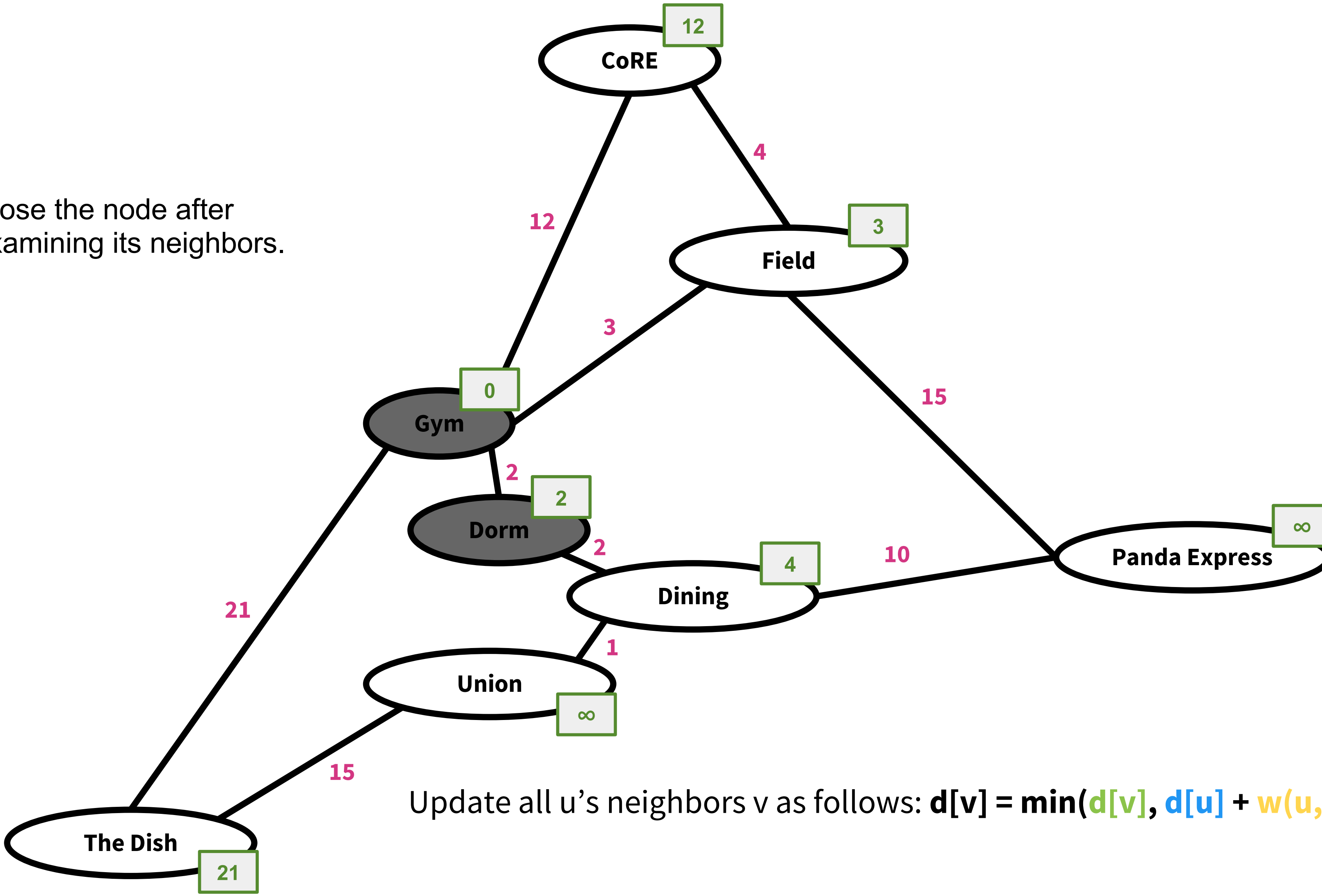
Dijkstra's Algorithm



Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Dijkstra's Algorithm

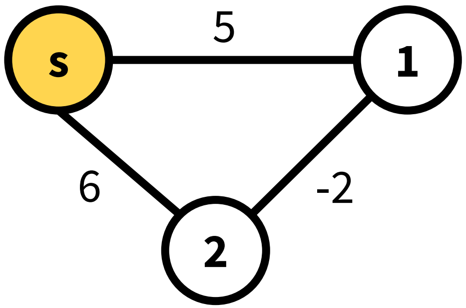
Close the node after examining its neighbors.



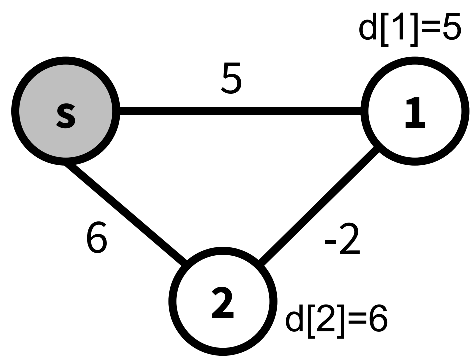
Update all u's neighbors v as follows: **d[v] = min(d[v], d[u] + w(u, v))**

Problem of Dijkstra’s Algorithm

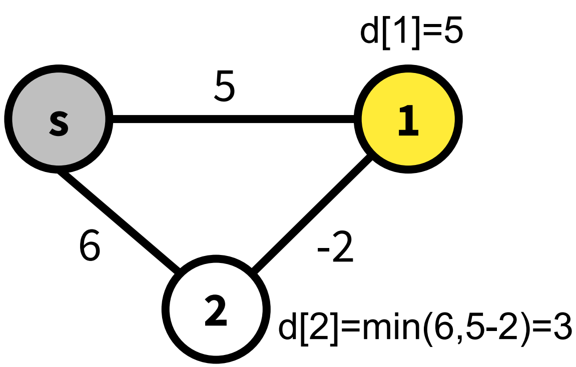
Can not handle negative edge weights properly.



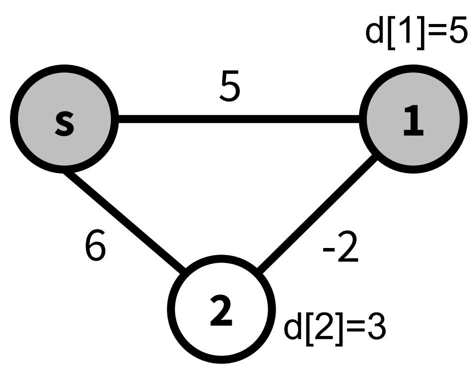
0. Original graph



1. Initialization



3. Select the smallest open node
and update its open neighbors



4. Close the node

Once a node is closed, its distance to source will never be updated anymore.

Bellman-Ford Algorithm for SSSP

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

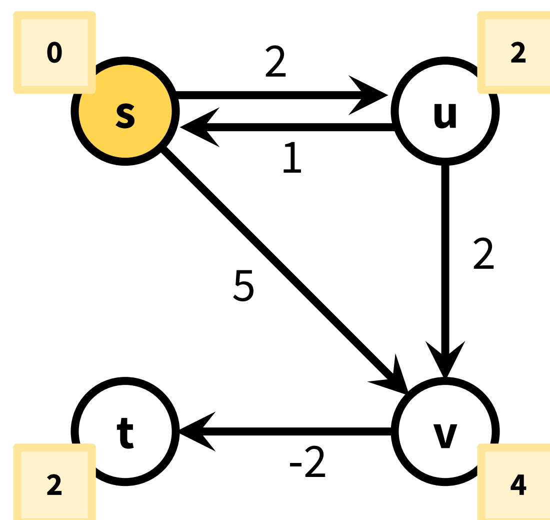
Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with at most 1 edge has cost **∞** (no path exists).

The shortest path from s to t with at most 2 edges has cost **3** (s-v-t).

The shortest path from s to t with at most 3 edges has cost **2** (s-u-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



$d^{(k)}[b] = \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$

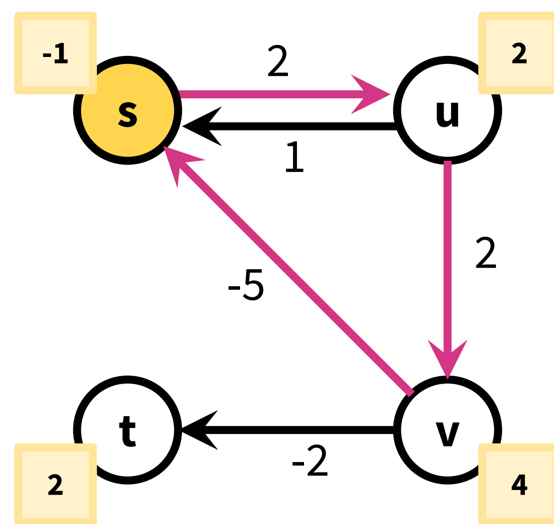
Use BF to Detect Negative Cycle

Basic idea: perform an extra iteration.

If there is no negative cycle, then BF algorithm only needs |V|-1 iterations.

If the |V|-th iteration changed the shortest distance of a vertex, then there must exist a negative cycle.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	∞	∞
$d^{(2)}$	0	2	4	∞
$d^{(3)}$	-1	2	4	2
$d^{(4)}$	-1	1	4	2



Floyd-Warshall Algorithm for APSP*

$D^{(k)}[u, v] = \min\{D^{(k-1)}[u, v], D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]\}$

“to”

“from”

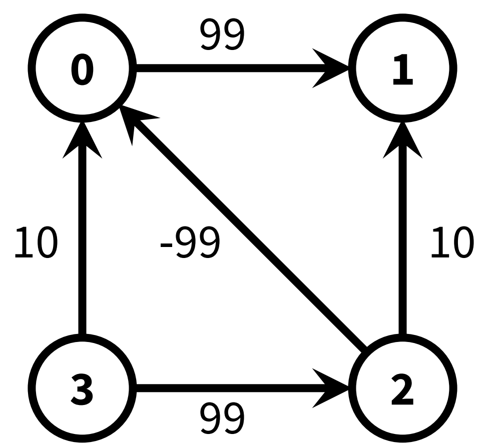
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	0	0	∞
3	10	109	99	0

$D^{(1)}[u, v]$

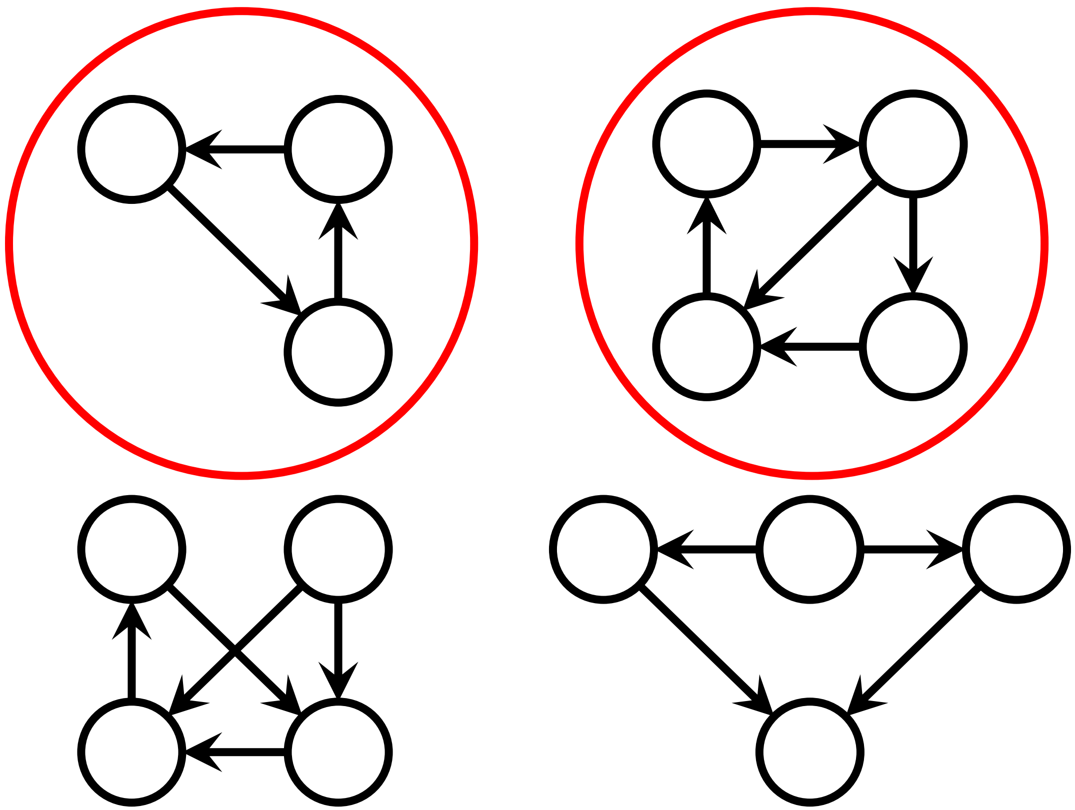
$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



(* Advanced Optional Topic)

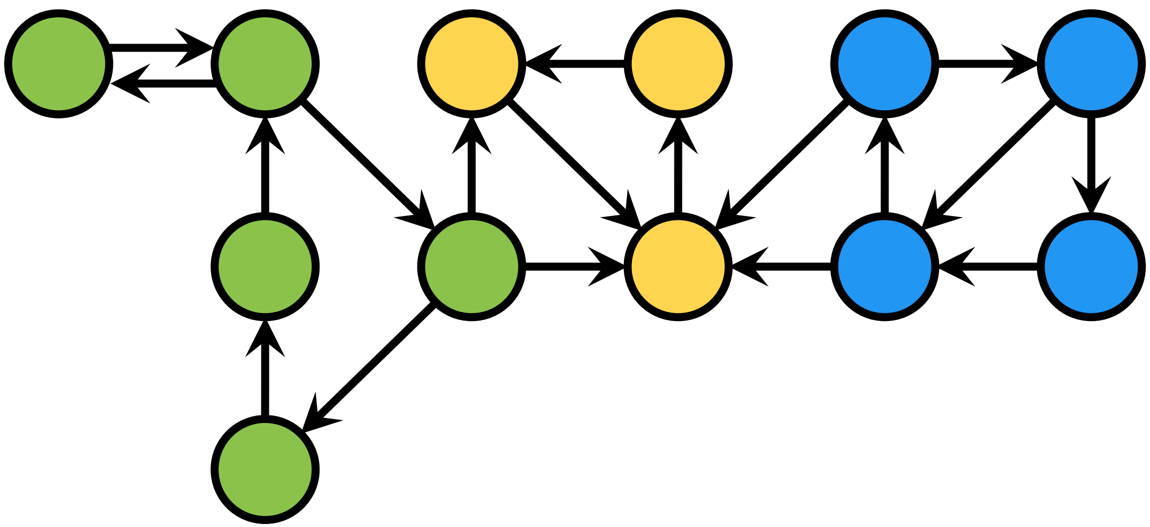
Strongly Connected Components

A directed graph G = (V, E) is strongly connected if,
for all pairs of vertices u and v, there's a path from u to v
and a path from v to u.



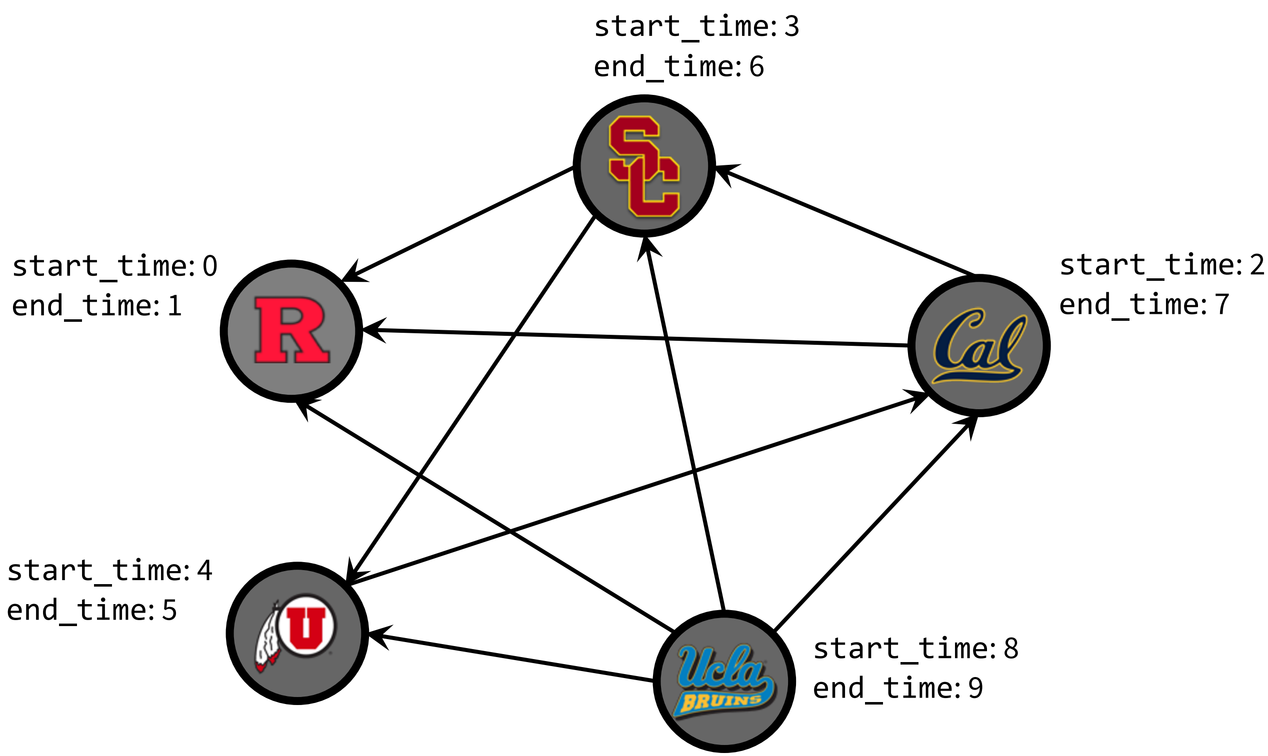
Strongly Connected Components

We can decompose a graph into its strongly connected components (SCCs).



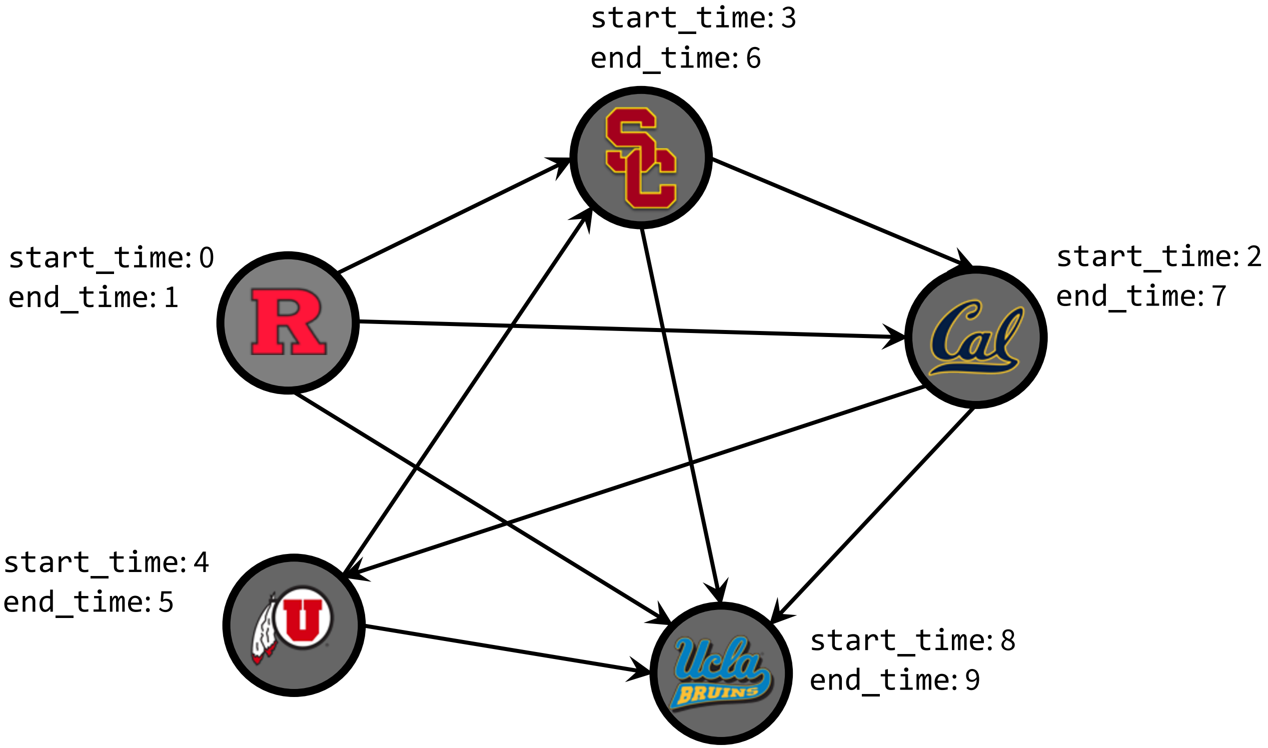
Kosaraju's Algorithm

1. Repeat `dfs` from an arbitrary vertex until done.



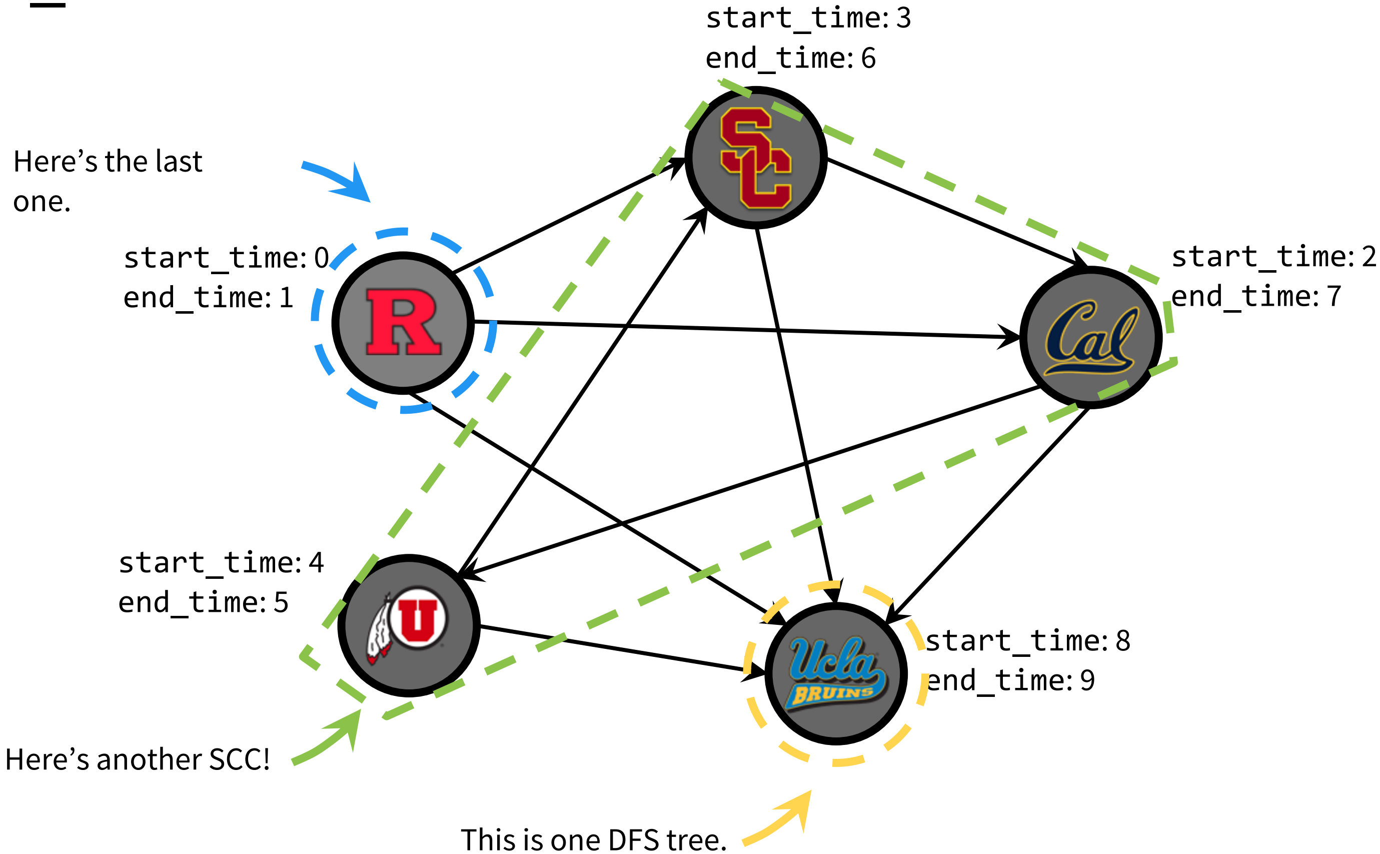
Kosaraju’s Algorithm

2. Reverse all of the edges.



Kosaraju's Algorithm

3. Repeat `dfs` again, starting with vertices with the largest `end_time`.

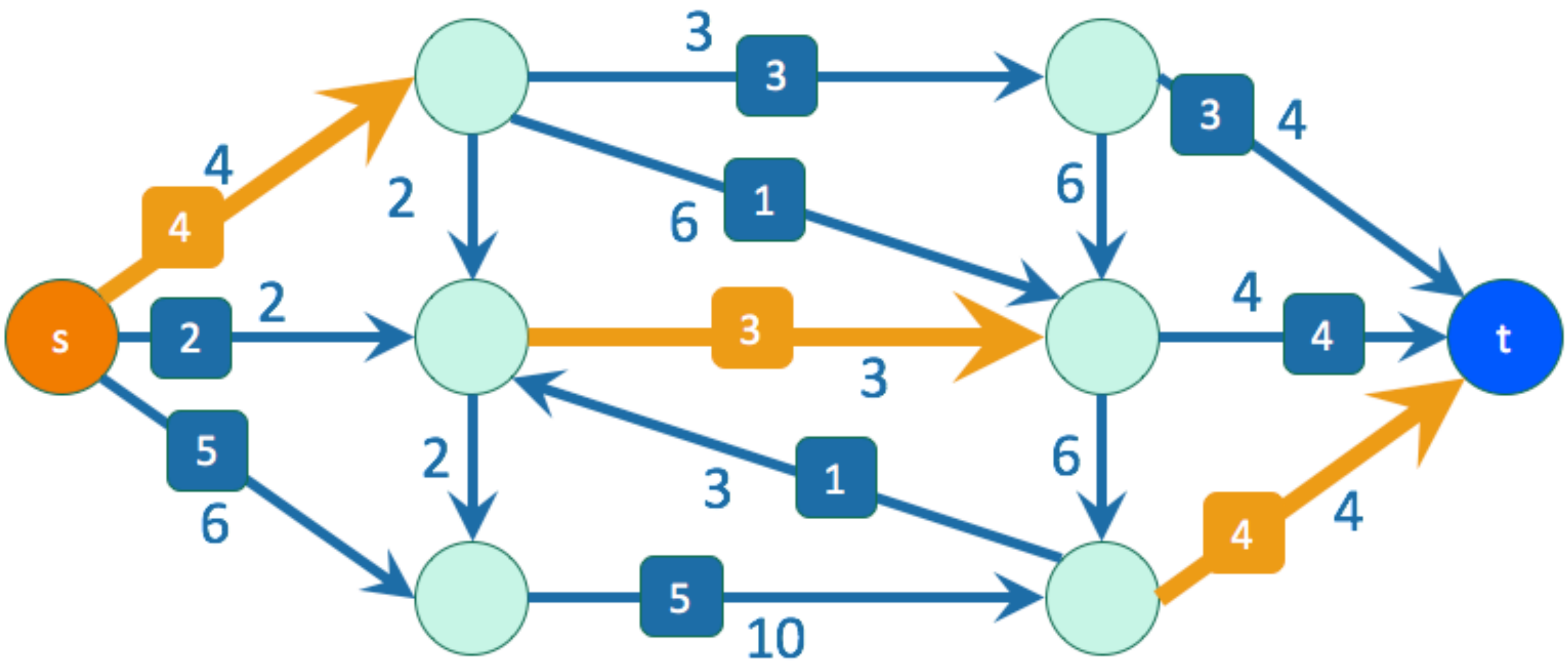


Theorem

Max-flow min-cut theorem

The value of a max flow from s to t
is equal to
the cost of a min s-t cut.

Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.



Ford-Fulkerson algorithm

- Usually we state the algorithm first and then prove that it works.
- Today we're going to just start with the proof, and this will inspire the algorithm.

Outline of algorithm:

- Start with zero flow
- We will maintain a **"residual graph"** G_f
- A path from s to t in G_f will give us a way to improve our flow.
- We will continue until there are no s-t paths left.

Greedy Algorithms

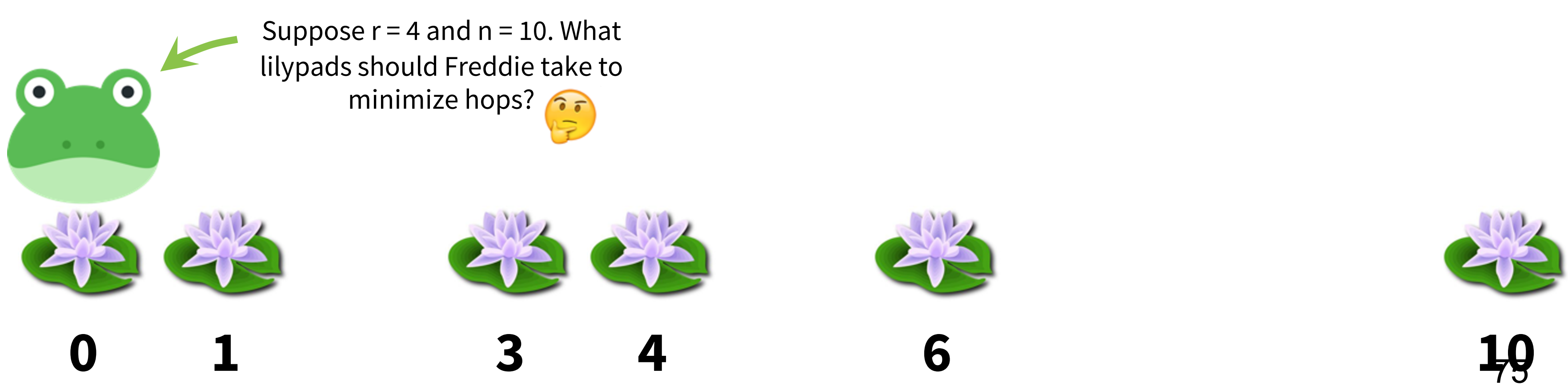
Freddie the Frog

Freddie the Frog starts at position 0 along a river. His goal is to reach position n.

There are lilypads at various positions, including at position 0 and position n.

Freddie can hop at most r units at a time.

Task: Find the path Freddie should take to minimize hops, assuming such a path exists.



Frog Hopping Correctness

We need to prove two properties about the algorithm to guarantee correctness.

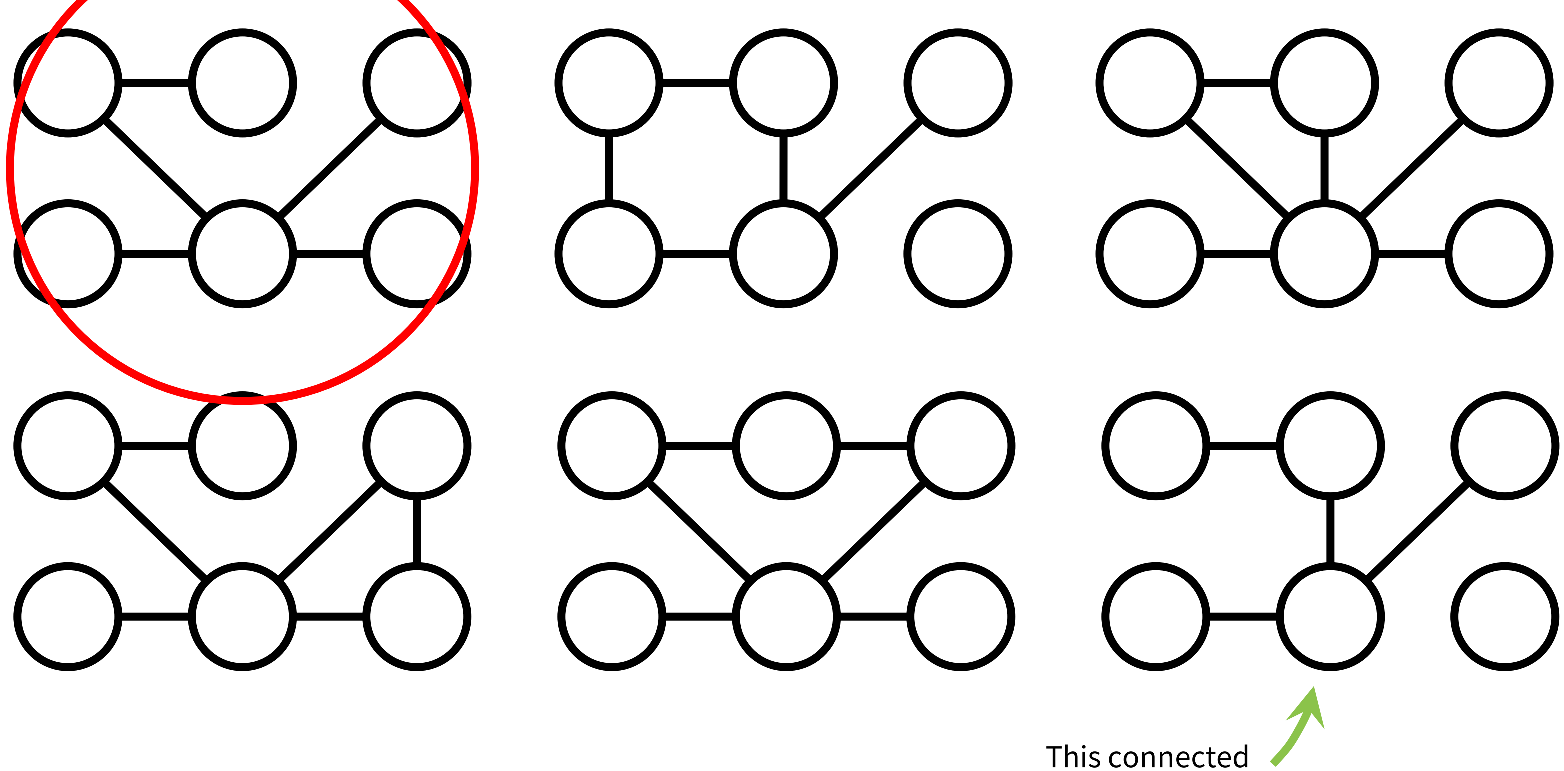
(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules).

(2) **Optimality.** The algorithm finds an optimal series of hops (i.e. there isn't a better path available).

MSTs

A spanning tree is a tree that connects all of the vertices.

Which of these graphs are spanning trees? 🤔

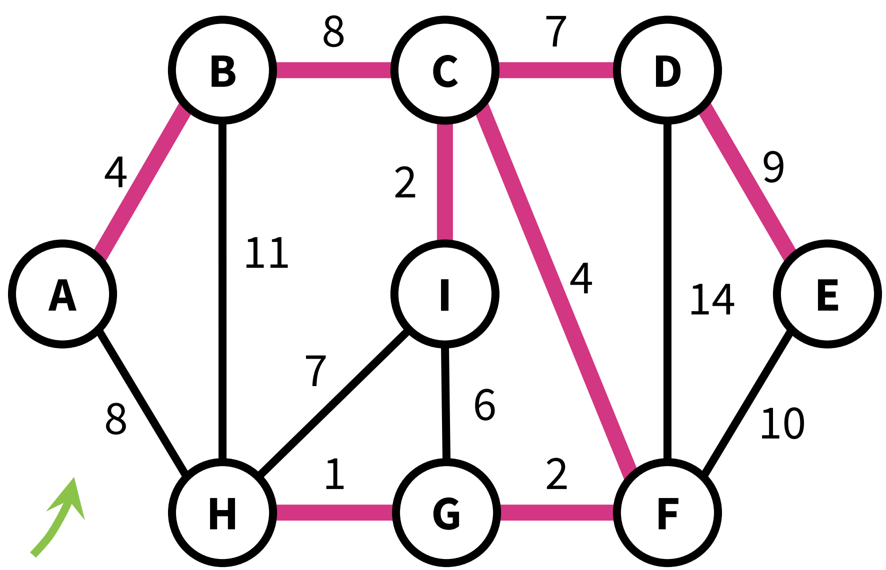


MSTs

mininmum **of minimal cost**

A spanning tree is a tree that connects all of the vertices.

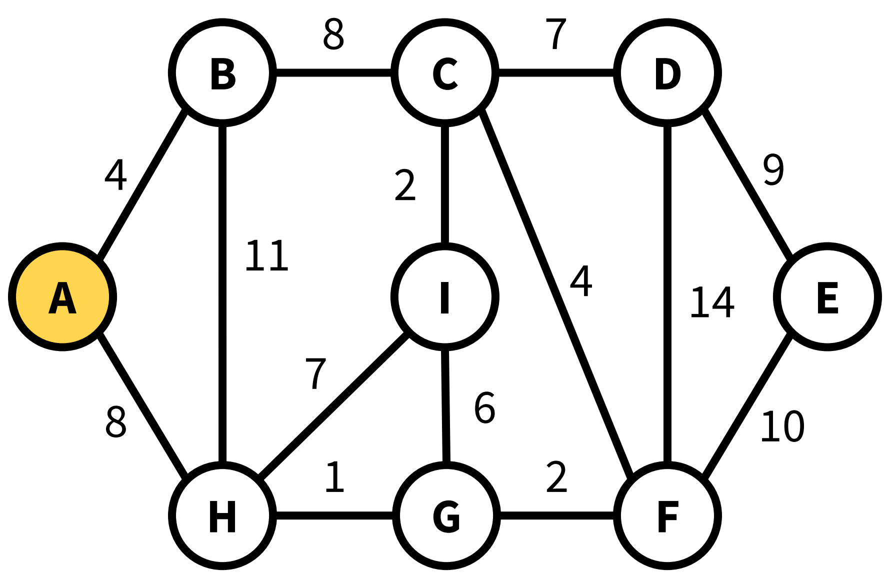
The cost of a spanning tree is the sum of the weights on the edges.



This spanning tree has a cost of 37. This is a minimum spanning tree.

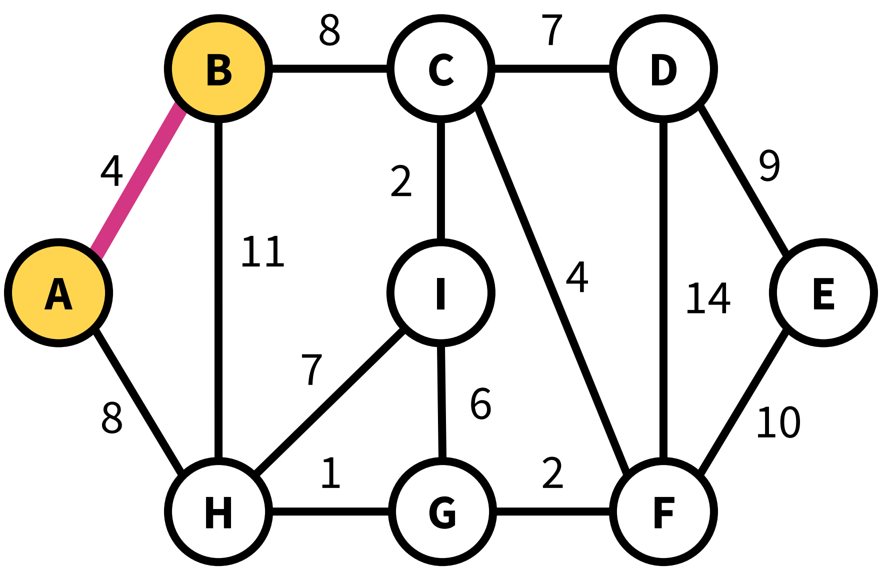
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex of the lightest edge.



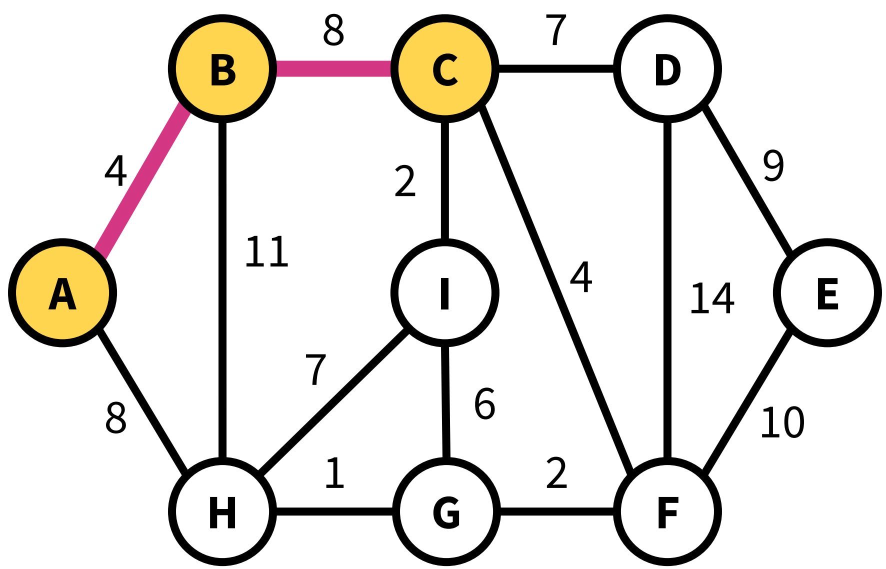
Prim’s Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex of the lightest edge.



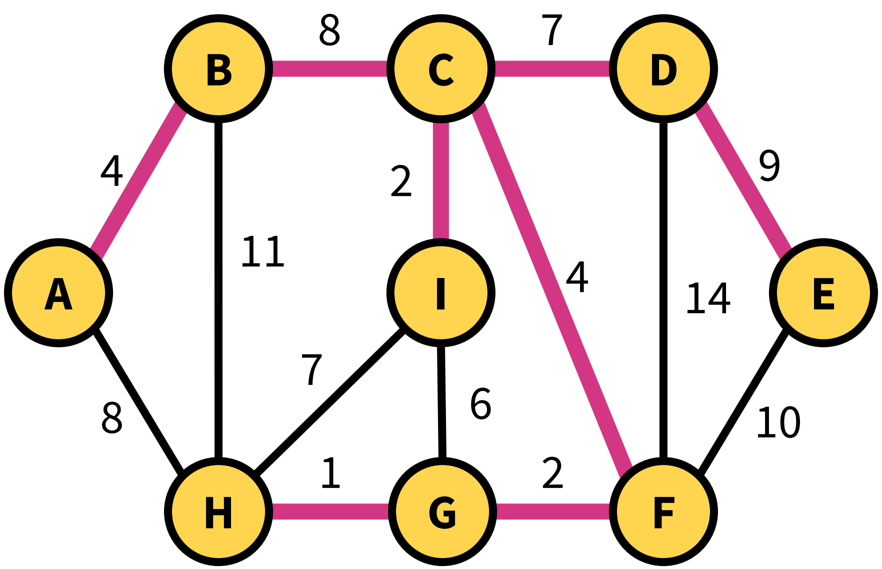
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex of the lightest edge.



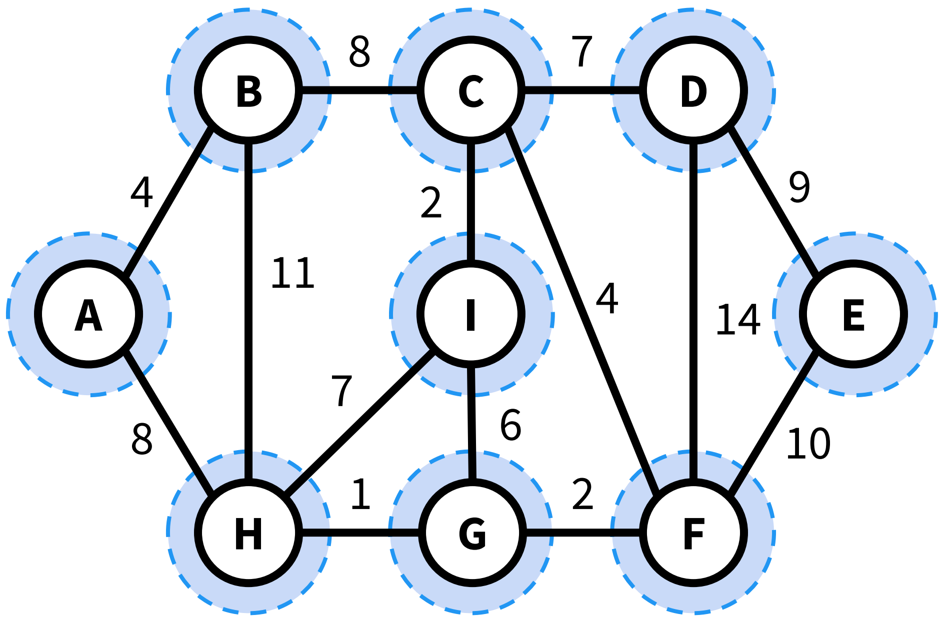
Prim's Algorithm

Main idea: Extend a single tree of visited vertices by greedily adding the closest vertex of the lightest edge.



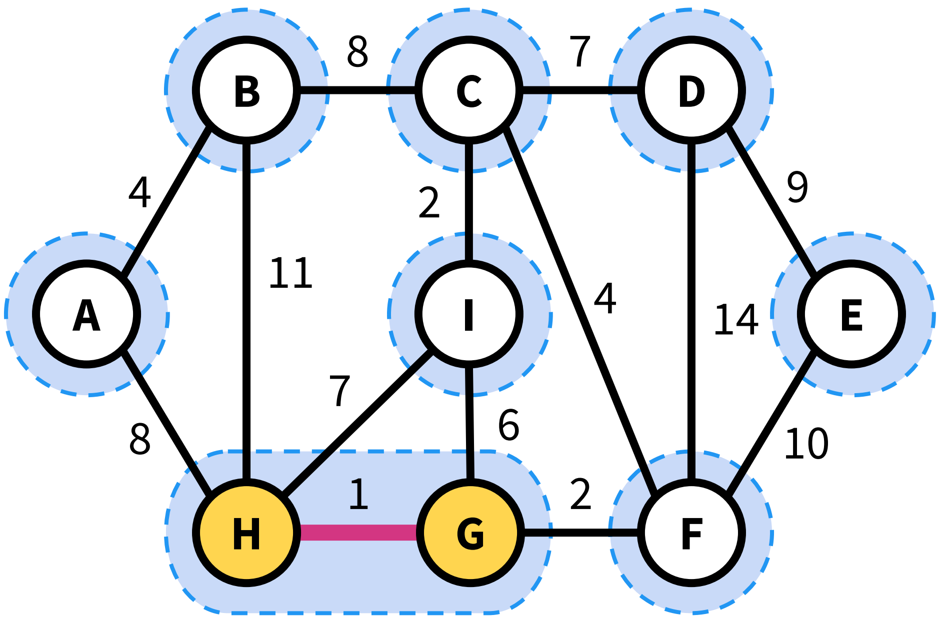
Kruskal’s Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



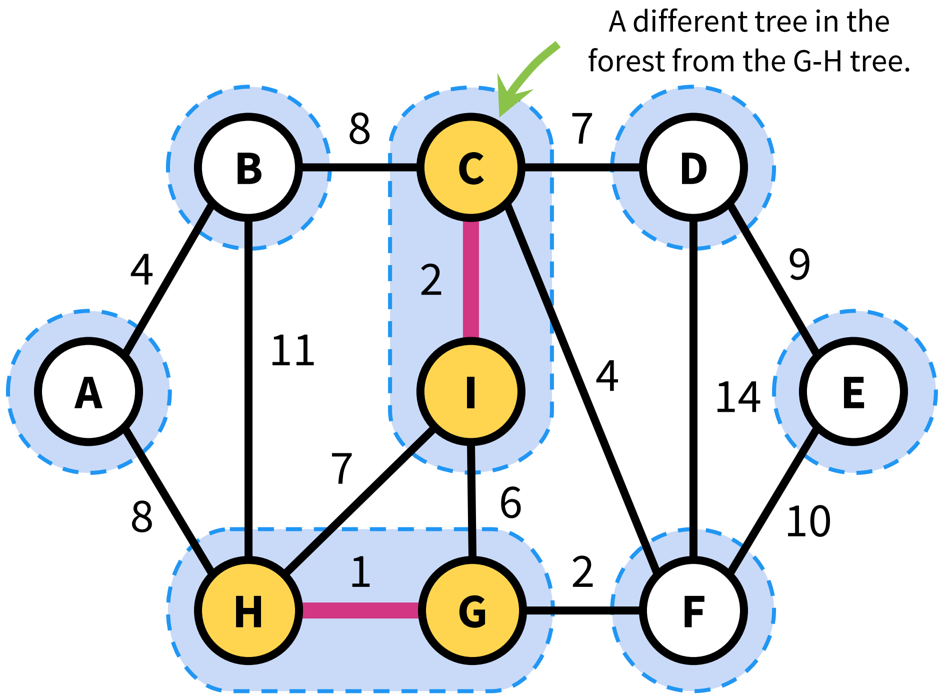
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

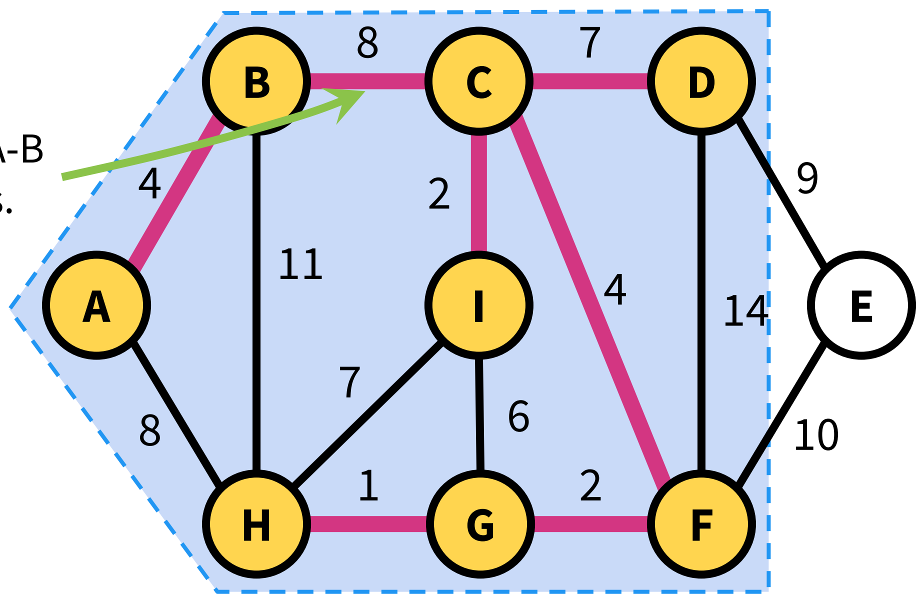
Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

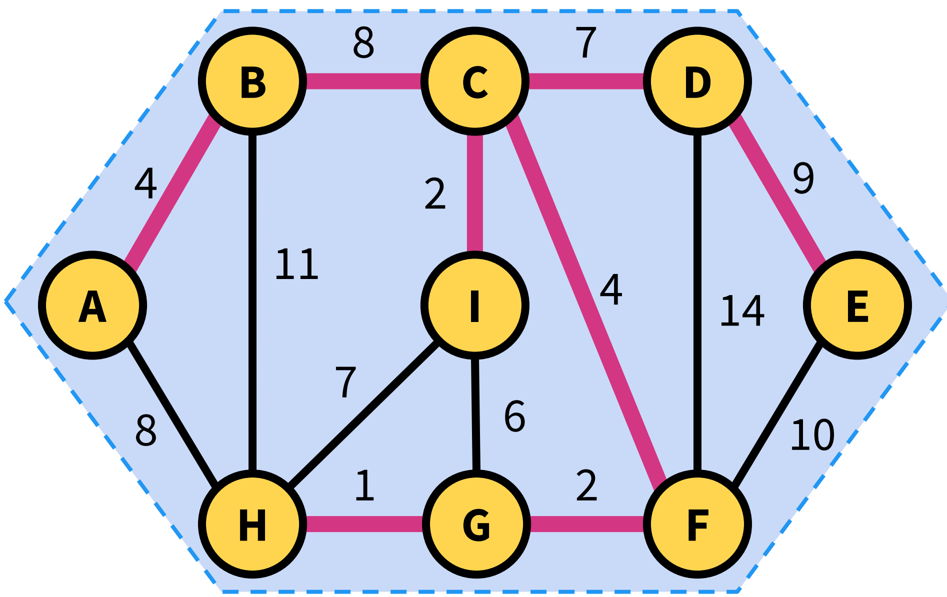
Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.

This edge merges the A-B and C-D-F-G-H-I trees.



Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Dynamic Programming

Four Steps

Steps of dynamic programming

(1) Identify optimal substructure with overlapping subproblems.

(2) Define a recursive formulation.

(3) Use dynamic programming to solve the problem.

(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Bellman-Ford Algorithm for SSSP

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

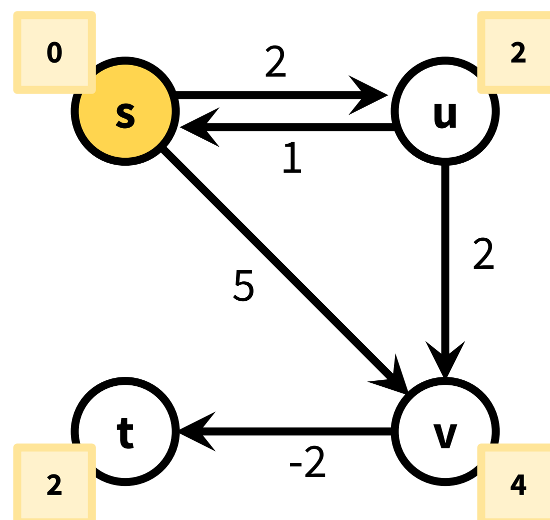
Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with at most 1 edge has cost **∞** (no path exists).

The shortest path from s to t with at most 2 edges has cost **3** (s-v-t).

The shortest path from s to t with at most 3 edges has cost **2** (s-u-v-t).

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	0	∞	∞	∞
$\mathbf{d^{(1)}}$	0	2	5	∞
$\mathbf{d^{(2)}}$	0	2	4	3
$\mathbf{d^{(3)}}$	0	2	4	2



$$d^{(k)}[b] = \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$$

Floyd-Warshall Algorithm for APSP*

$D^{(k)}[u, v] = \min\{D^{(k-1)}[u, v], D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]\}$

“to”

“from”

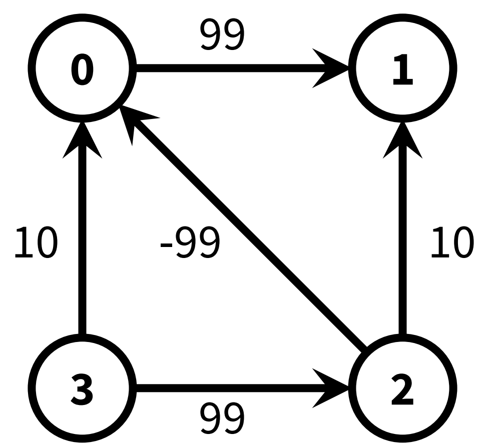
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	0	0	∞
3	10	109	99	0

$D^{(1)}[u, v]$

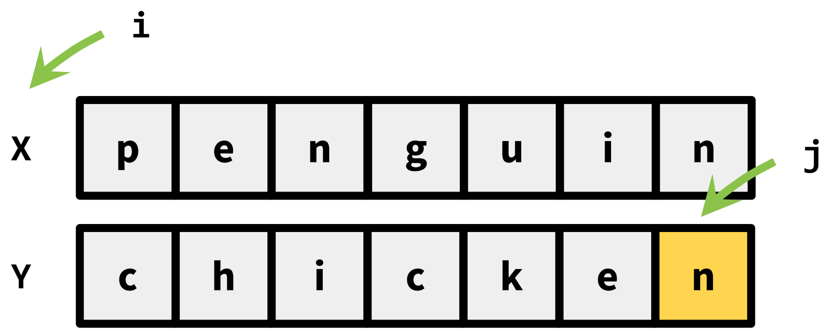
$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



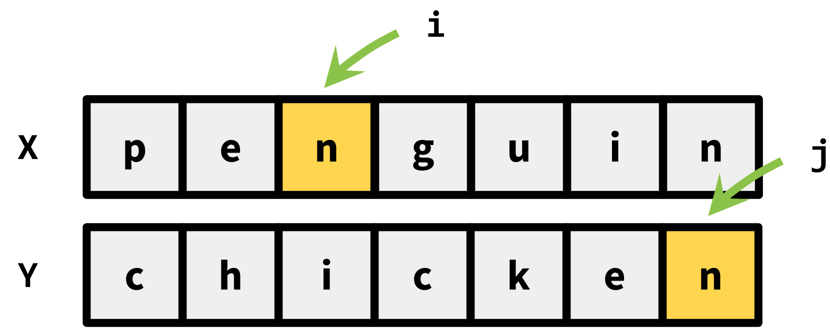
(* Advanced Optional Topic)

Longest Common Subsequence

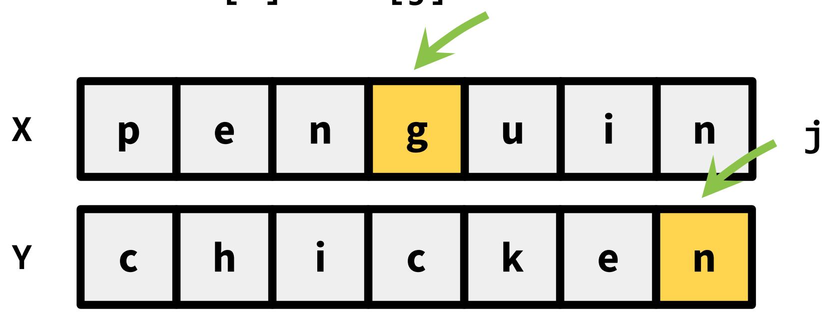
Base case (Case 0): `i` or `j` is -1

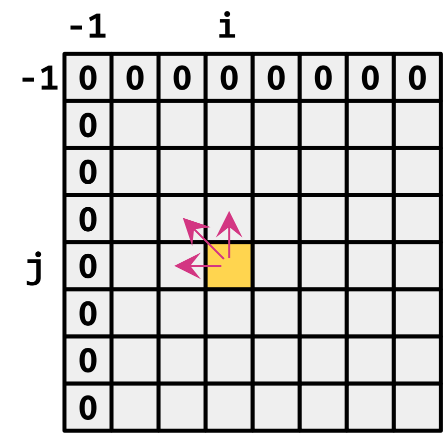


Case 1 `X[i] = Y[j]`



Case 2 `X[i] != Y[j]`





An element at position (i, j) only depends on elements at positions (i-1, j), (i, j-1), and (i-1, j-1).

$$
T(i,\ j) = \begin{cases} 0 & \text{if } \mathbf{i} \text{ or } \mathbf{j} \text{ is -1} \\ 1 + T(i\text{-}1,\ j\text{-}1) & \text{if } \mathbf{X[i]} = \mathbf{Y[j]} \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \\ \max\{T(i\text{-}1,\ j),\ T(i,\ j\text{-}1)\} & \text{if } \mathbf{X[i]} \neq \mathbf{Y[j]} \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \end{cases}
$$

Longest Common Subsequence

For example, consider `lcs_helper("ACGGA", "ACTG")`.

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

That **1** must have come from this **0** since **A**'s match.

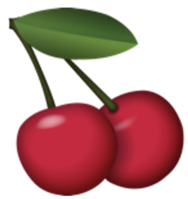
LCS A C G

$$T(i, j) = \begin{cases} 0 & \text{if } \mathbf{i} \text{ or } \mathbf{j} \text{ is -1} \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \end{cases}$$

Knapsack

Which items should I cram inside my knapsack?

We have n items with weights and values.

item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

And we have a knapsack that can only carry so much weight.

capacity: 10

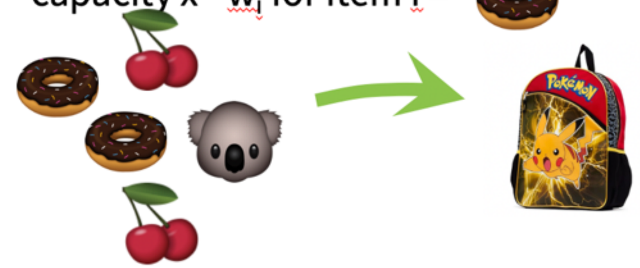
Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

If this is an optimal solution for capacity x

capacity x
value V

Then this must be an optimal solution for capacity x - w_i for item i = 



capacity x - w_i
value V - v_i

Define a recursive formulation. Let **V[x]** be the optimal value for capacity x.

$V[x] = \max_i\{$  $+$  $\}$

The maximum over all item i such that $w_i \leq x$.

The optimal way to fill the smaller knapsack

The value of item i.

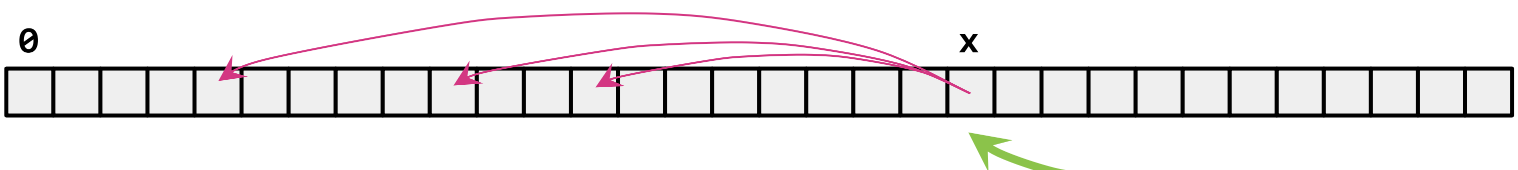
$$V[x] = \begin{cases} 0 & \text{if there are no i where } w_i \leq x \\ \max_i\{V[x-w_i] + v_i\} & \text{otherwise} \end{cases}$$

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

Use dynamic programming to solve the problem.

In what order do we need to fill our table according to the formulation from (2)?



An element at position x in the table depends on elements at positions $x - w_i$ for all i. So we want to fill out the values at these positions before x.

$$V[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i\{V[x-w_i] + v_i\} & \text{otherwise} \end{cases}$$

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

We reason that we must solve the problem for a smaller number of items **and** for smaller knapsacks.

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

First solve the problem for few items

Then more items

Then more items

We need a two-dimensional table!

0/1 Knapsack

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1 🍩	1 🍩	1 🍩
j=2	0	1 🍩	4 🍒	5 🍩🍒
j=3	0	1 🍩	4 🍒	6 🌮

current entry

relevant previous entry

So the optimal solution is to put one taco in your knapsack!

	🍩	🍒	🌮
weight	1	2	3
value	1	4	6

capacity: 3

$$
\mathtt{V[x,j]} = \begin{cases} \mathtt{0} & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \mathtt{max\{V[x,j-1],\ V[x-w_j,j-1] + v_j\}} & \text{otherwise} \end{cases}
$$

A High-level Picture

What we have learned

Basic Techniques for algorithmic analysis

Asymptotic analysis (big-O notation), proofs of correctness, runtime analysis
Solving recurrences: Recursion tree method, iteration method, master theorem

Sorting Algorithms

Insertion sort, Merge sort, Quick sort, Sorting lower bound, linear sorting algorithms,
Sorting data structures (Binary search tree and Red black tree)

A High-level Picture

What we have learned (cont.) 5 algorithmic paradigms

Divide and Conquer: Merge sort, Quick sort, Integer multiplication, Select_k

Randomized Algorithm:

- Las Vegas: Quick sort, Quick select, Majority element, Hash tables, Expected runtime analysis
- Monte-Carlo: Karger's Algorithm for finding minimum cut, Probability of success analysis

Graph Algorithm:

- Graph Basics: Graph representation, DAG, DFS, BFS, Topological Ordering, In-order traversal of BST
- Shortest Path: Using BFS, Dijkstra's Algorithm (SSSP), Bellman-Ford (SSSP), Floyd-Warshall (APSP)
- SCC: Kosaraju's Algorithm
- Global Minimum Cut: Karger's Algrothm
- Maxflow-Mincut: Ford-Fulkerson Algorithm

Greedy Algorithm

- Frog Hopping, Proof of correctness (not required)
- Minimum Spanning Tree: Prim's Algorithm (lightest edge), Kruskal's Algorithm (cheapest edge)

Dynamic Programming

- Four steps of designing dynamic programming algorithm
- Bellman-Ford Algorithm (SSSP), Floyd-Warshall (APSP)
- Longest Common Subsequence, 0/1 and Unbounded Knapsack

Thanks for the Journey!

Wish you a lot of a happiness and success in your future life and career ☺