Graph Theory 1 (Alpha)

All Pairs Shortest Path (APSP): Floyd-Warshall

General problem: Given a graph with N nodes and E edges (the *i'th* edge has distance D_i), find the shortest path between any pair of nodes

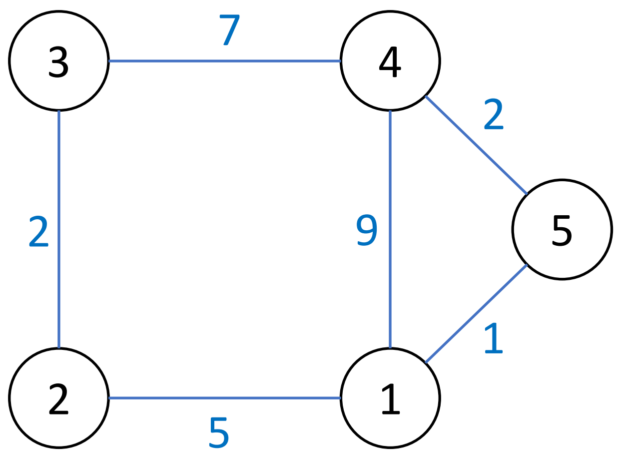
Floyd-Warshall's Algorithm – O$(\boldsymbol{N^3})$

Description

- Maintain a 2D distance array describing the distance any pair of nodes
- Start with only direct edge distances (∞ for no edge between a pair of nodes)
- Try each node as an "intermediate" node on the path between any pair of nodes and see if you can improve the distance between them

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm – O(N^3)



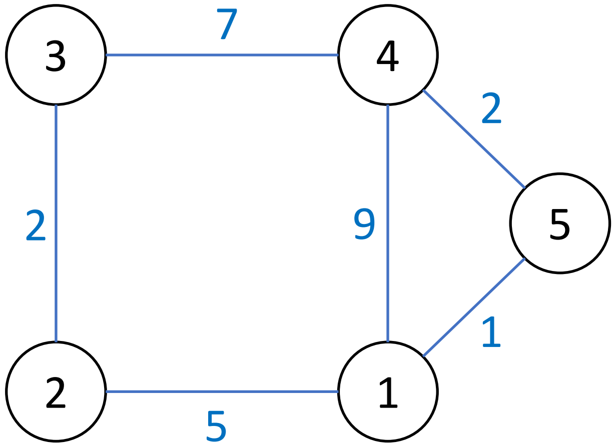
Initial distance array

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

- Just the edge distances in the graph
- ∞ for "no edge"
- 0 distance for node to itself

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm – O(N^3)



First round, intermediate node = 1

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

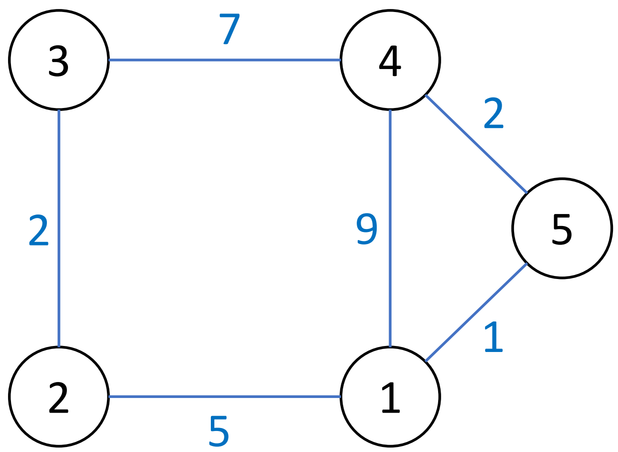
- On round 1, try using 1 as an "intermediate" node between every pair of nodes and see if we can improve distances

- New path between 2 and 4 and between 2 and 5. In both cases, we can improve our old distance of ∞ with the new distance (14 and 6 respectively).

- There is also a new path 4 → 1 → 5 which has cost 9 + 1 = 10, however, this does not improve the current distance between 4 and 5 which is 2, so we ignore it

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm – O(N^3)



Second round, intermediate node = 2

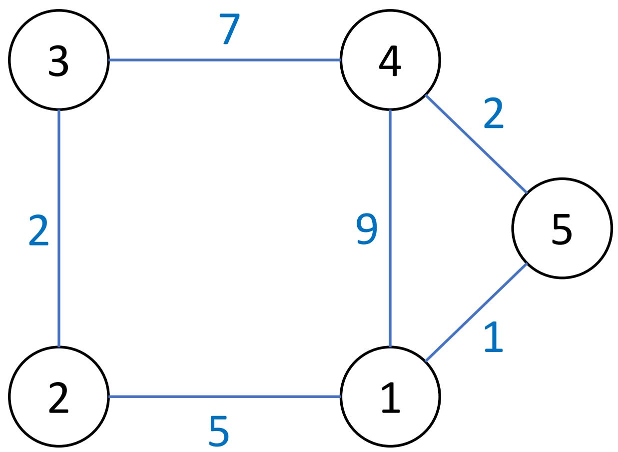
	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

- Try use node 2 as the "intermediate" node between every pair of nodes and see if we can improve distances

- New path between 1 and 3 and between 3 and 5 (3 →2 →1 →5). In both cases, we can improve our old distance of ∞ with the new distance (7 and 8 respectively).

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm – O(N^3)



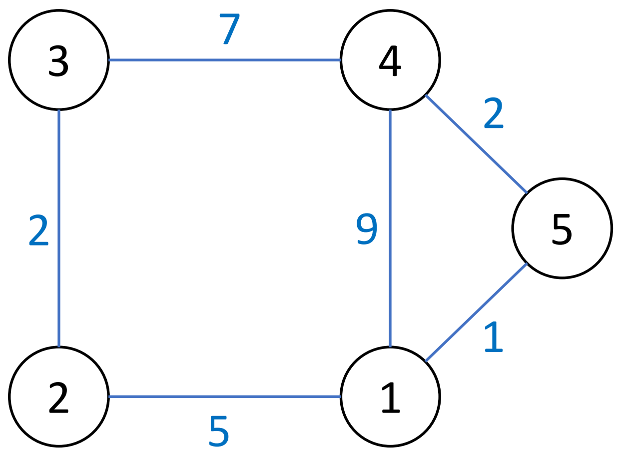
Third round, intermediate node = 3

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

- Try use node 3 as the "intermediate" node between every pair of nodes
- New path between 2 and 4 with distance 9 (better than old distance of 14)

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm – O(N^3)



After all rounds…

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

- Our distance array now gives the minimum distance between any pair of nodes

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm Notes

- After the *k'th* round, the distance matrix contains the shortest distance between any pair of nodes using *only nodes* $1, 2, \dots, k$ (because these are the nodes we have tried as intermediate nodes so far) – this can actually be thought of as a DP function:

  dp(k, i, j) = minimum distance from node I to node j using only nodes 1, 2, …, k

- Time complexity:

Try each node an intermediate node $\Rightarrow n$
For each intermediate node, try improve all pairs of distances $\Rightarrow n^2$
Total = $O(n^3)$

- Works for both directed graphs (distance matrix will not be symmetric) and also works with negative edges (but not negative cycles!)

All Pairs Shortest Path (APSP): Floyd-Warshall

Floyd-Warshall's Algorithm Implementation

```
for(int k = 0; k < N; k++){
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

- Outermost loop (k) is intermediate node
- Two inner loops (i, j) are checking all pairs
- Inner statement updates distance with intermediate node if it's better

- The order of the for-loops (KIJ) is important and will give wrong distances if in a different order
- If you forget this order, repeat all three for-loops 3 times (can be proven to give correct distances)

Single Source Shortest Path (SSSP): Bellman-Ford

General problem: Given a graph with N nodes and E edges (the *i'th* edge has distance D_i), find the shortest path from a source node S to every other node in the graph.

Bellman-Ford's Algorithm – O(NE)

- Works on all types of graphs (including those with negative edge weights unlike Dijkstra's)

- Can detect negative cycles (so can be used to ensure no negative cycles before something else which doesn't work with negative cycles)

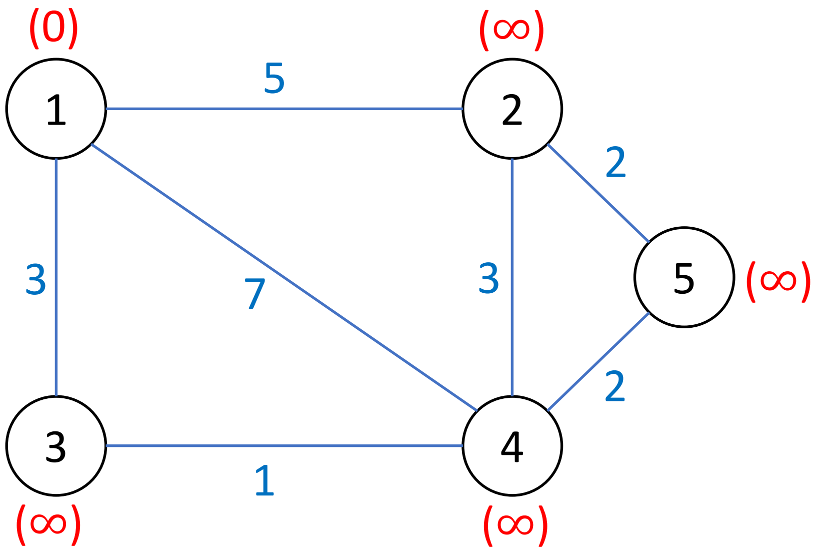
Single Source Shortest Path (SSSP): Bellman-Ford

Description

- Track distances from source node S to every other node
- Initially, every distance is ∞ (except the source which is 0)
- Repeat the following N-1 times: Try using each edge in the graph to reduce distances

Single Source Shortest Path (SSSP): Bellman-Ford

Bellman-Ford's Algorithm – $O(NE)$

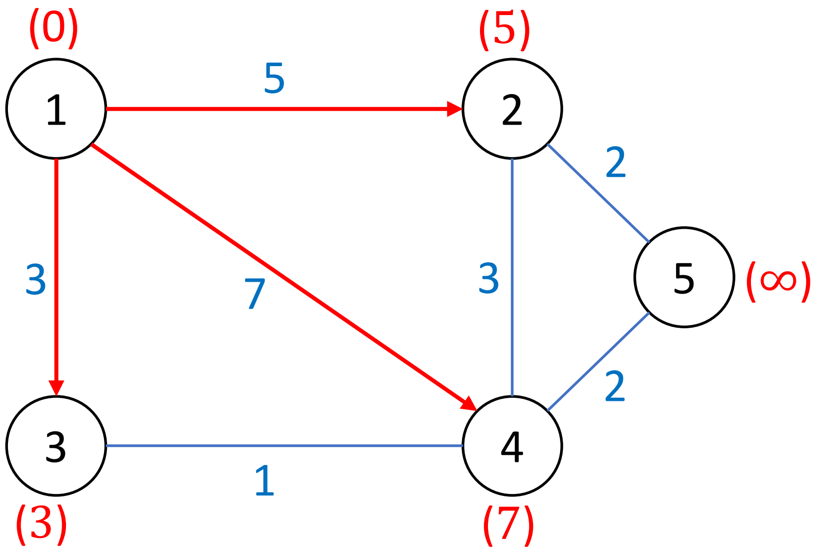


Initial distances

Node	1	2	3	4	5
Distance	0	∞	∞	∞	∞

Single Source Shortest Path (SSSP): Bellman-Ford

Bellman-Ford's Algorithm – $\mathbf{O}(\boldsymbol{NE})$



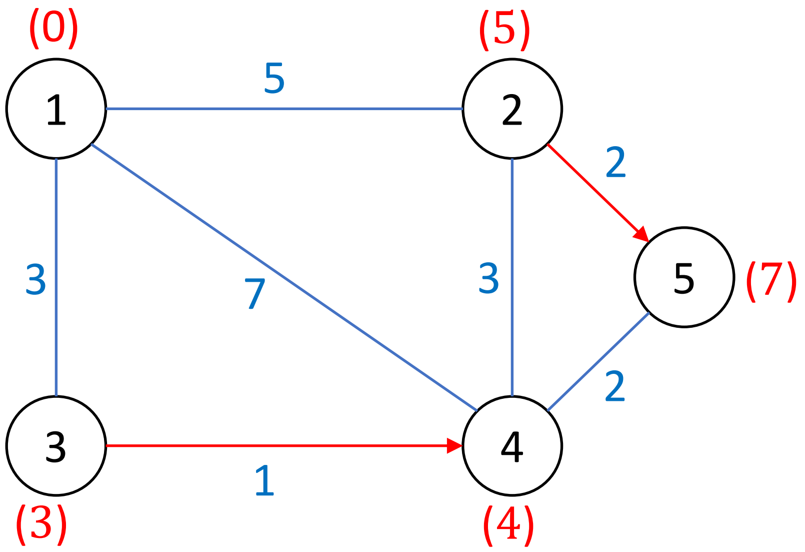
After first iteration

Node	1	2	3	4	5
Distance	0	5	3	7	∞

- Nodes 2, 3, 4 have distances updated coming from node 1
- Note that every other edge is still checked to see if it can improve any distances (but every other edge in this case had nodes with distance ∞ on both ends)

Single Source Shortest Path (SSSP): Bellman-Ford

Bellman-Ford's Algorithm – $\mathbf{O}(\boldsymbol{NE})$



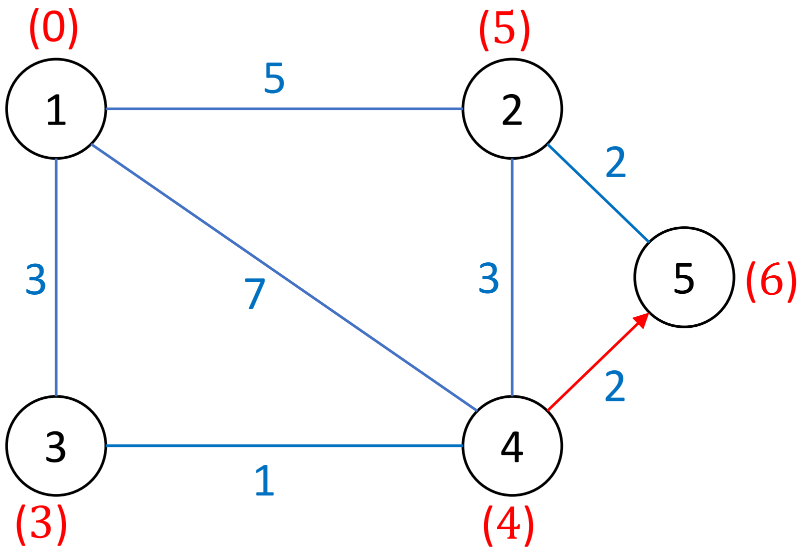
After second iteration

Node	1	2	3	4	5
Distance	0	5	3	4	7

- Nodes 4, 5 have distances updated coming from nodes 3 and 2 respectively

Single Source Shortest Path (SSSP): Bellman-Ford

Bellman-Ford's Algorithm – $O(NE)$



After third iteration

Node	1	2	3	4	5
Distance	0	5	3	4	6

- Node 5's distance gets improved from node 4
- After this, no further improvements can be made (and we have all shortest distances from source node 1)

Single Source Shortest Path (SSSP): Bellman-Ford

Implementation

```
// Initialise all distances to infinity, except the source which has distance 0
for (int i = 1; i <= N; i++) distance[i] = INF;
distance[S] = 0;

// Repeat procedure n-1 times
for (int i = 1; i <= N-1; i++){

        // Try use all edges to decrease distance
        for (auto e : edges){

                // edges are stored as tuples (a, b, w) indicates there's an edge from a to b with weight w
                int a, b, w;
                tie(a, b, w) = e;
                distance[b] = min(distance[b], distance[a]+w);
        }
}
```

Single Source Shortest Path (SSSP): Bellman-Ford

Why does it work? Proof by induction (1)

- Consider a shortest path from S to any node T, denoted: $Sx_1x_2 \dots T$

- Observe that every prefix of this path must also be a shortest path (S, Sx_1, Sx_1x_2, … are all shortest paths)

- On the first iteration of Bellman-Ford, we propagate all edges from the source S, meaning we find the shortest path Sx_1

- Assume after the n'th iteration of Bellman-Ford, we have found the shortest path $Sx_1x_2 \dots x_n$

- On the (n+1)'th iteration of Bellman-Ford, all edges from x_n are used to improve distances meaning $Sx_1x_2 \dots x_nx_{n+1}$ must be found.

- By induction, we conclude that on the *x*'th iteration of Bellman-Ford, we have found all shortest paths with (*x*+1) nodes

Single Source Shortest Path (SSSP): Bellman-Ford

Why does it work? Proof by induction (2)

- There can be at most *N* nodes in the shortest path from S to any node T as it will never be better to return to a node (that would mean a negative cycle).

- Therefore, by performing N-1 iterations of Bellman-Ford, we are guaranteed that the shortest path from the source to the destination will be achieved in a graph with no negative cycles

Single Source Shortest Path (SSSP): Bellman-Ford

Notes

- In practice, N-1 iterations aren't required, you can speed up the algorithm by terminating early if an iteration does not give any improvements to distances

- If after N-1 iterations, you do one more iteration and find an improvement, that means that a negative cycle must exist in the graph...

Why? We just proved that N-1 iterations is enough to find the shortest path from S to any node T for any graph with no negative cycles. Therefore, if one more iteration improves a path, we must have a negative cycle.

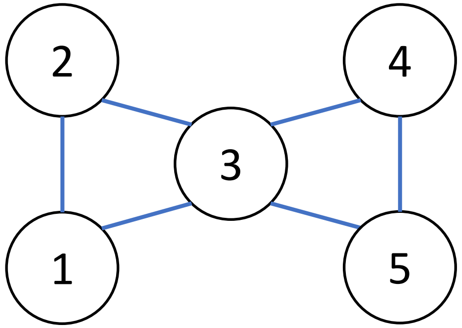
Eulerian Paths and Cycles

Definition

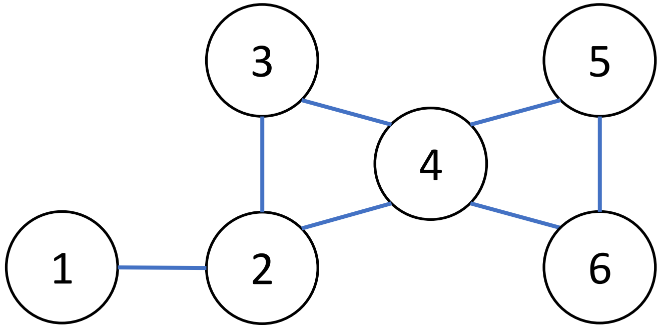
An **Eulerian path** in a graph is a path which visits every edge exactly once (vertices can be visited multiple times).

An **Eulerian cycle** is an Eulerian path which starts and ends at the same vertex.

Examples:



An Eulerian cycle of this graph is:
1→2→3→4→5→3→1
(It has no Eulerian paths)



An Eulerian path of this graph is:
1→2→3→4→5→6→4→2
(It has no Eulerian cycles)

Eulerian Paths and Cycles

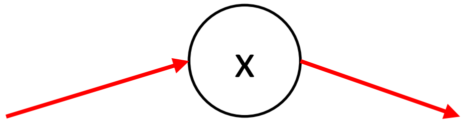
Existence Conditions

Definition 1: The degree of a node in a graph is the number of edges connected to that node.

Definition 2: The unvisited degree of a node in a graph with respect to a path, is the number of edges connected to that node which have not been visited by that path.

Theorem 1: If a graph has an Eulerian cycle, every node in the graph must have an *even* degree.

Proof: For every node that's visited in the Eulerian cycle, the path must enter on one edge and leave on a different edge decreasing the unvisited degree of that node by 2.



The initial unvisited degree of any node is equal to that node's degree. Once the Eulerian cycle is completed, every node should have an unvisited degree of 0. We can only decrease the unvisited degree of a node by 2 (entering and leaving), so for this number to reach 0, it must be *even* to begin with.

Eulerian Paths and Cycles

Existence Conditions

Theorem 2: If a graph has an Eulerian path, there must be exactly 2 nodes with *odd* degree.

Proof: (Note: sometimes Eulerian cycles are a subset of Eulerian paths – depending on convention, here an Eulerian path strictly *doesn't* start and end on the same node)

As before, for every node that's visited in the Eulerian path **except the start and end node**, the path must enter on one edge and leave on a different edge decreasing the unvisited degree of that node by 2.

The start and end node may be revisited multiple times during the path, every time this happens their unvisited degree decreases by 2. However, at the start of the path, the first node has its unvisited degree decreased by just 1. Similarly, at the end of the path, the last node has its unvisited degree decreased by just 1.

Since the unvisited degrees of these two nodes are decreased by 2 upon every re-visit, and 1 at the start and the end, these nodes must have odd degree initially to have unvisited degree reduced to 0.

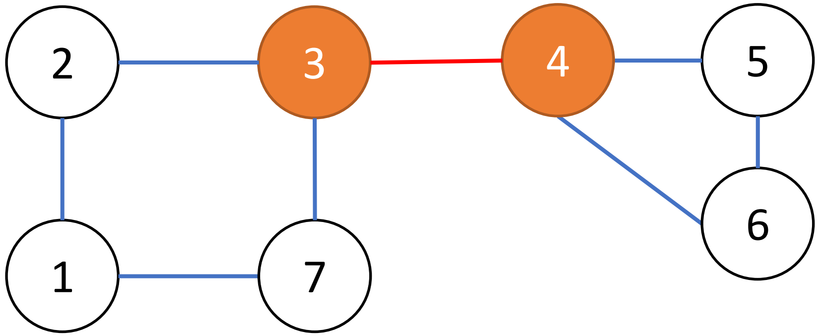
Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Fleury's algorithm – $O(E^2)$

- Start at one of the odd-degree nodes (if one exists) and start walking through the graph, delete each edge you traverse

- Don't ever take a bridge in the graph (an edge which is the only connection between two parts of the graph), like below:



If you start at node 3, the edge to node 4 is a bridge. If we take and delete it, then the graph is now disconnected (and we can no longer find an Eulerian path/cycle)

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Fleury's algorithm – $O(E^2)$

- Traversing is easy, bridge detection is more difficult (can be done in O(V+E) with Tarjan's bridge-finding algorithm – not covered in this lecture)

- There is a faster method to finding Eulerian paths and cycles…

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

- Strategy: Find all simple cycles and combine them into one which will be the Eulerian cycle/path

Recursively:

(starting at appropriate node V – an odd degree node if it exists)

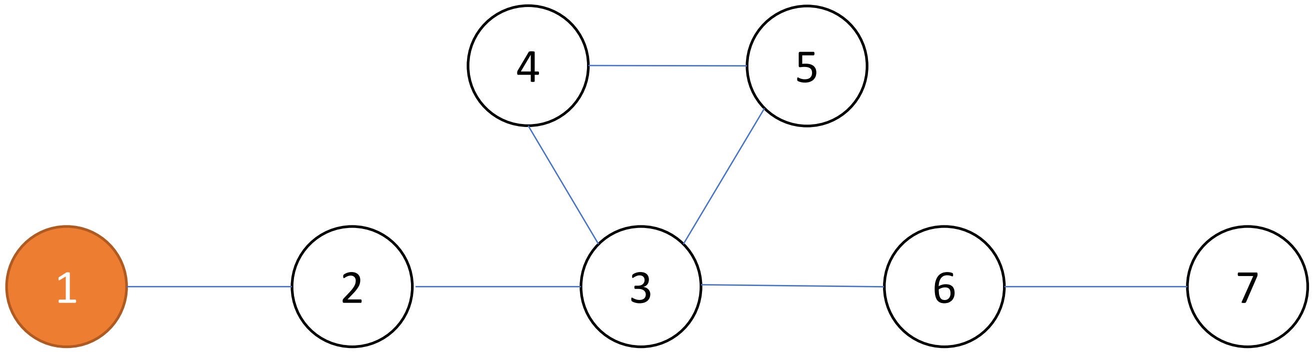
```
FindEulerPath(V):
       For every edge (V -> W) outgoing from V:
              Remove this edge
              FindEulerPath(W)
       Add V to answer
```

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {}

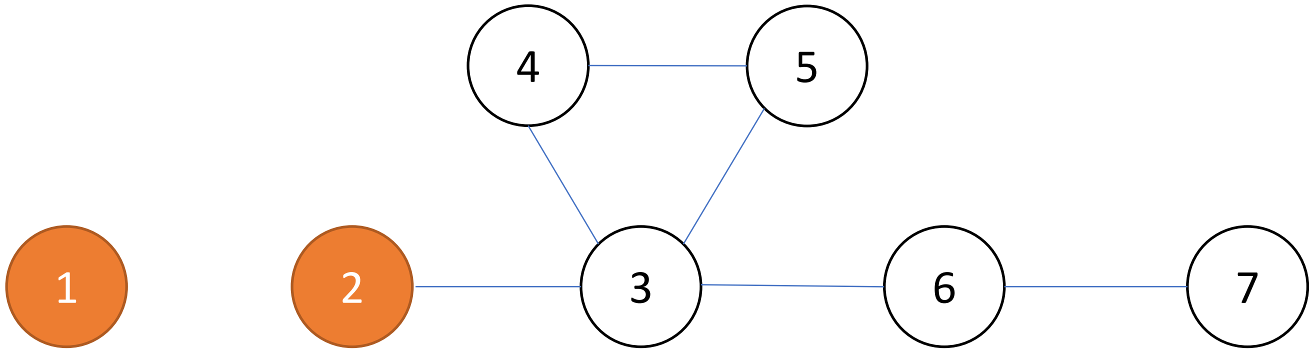


Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

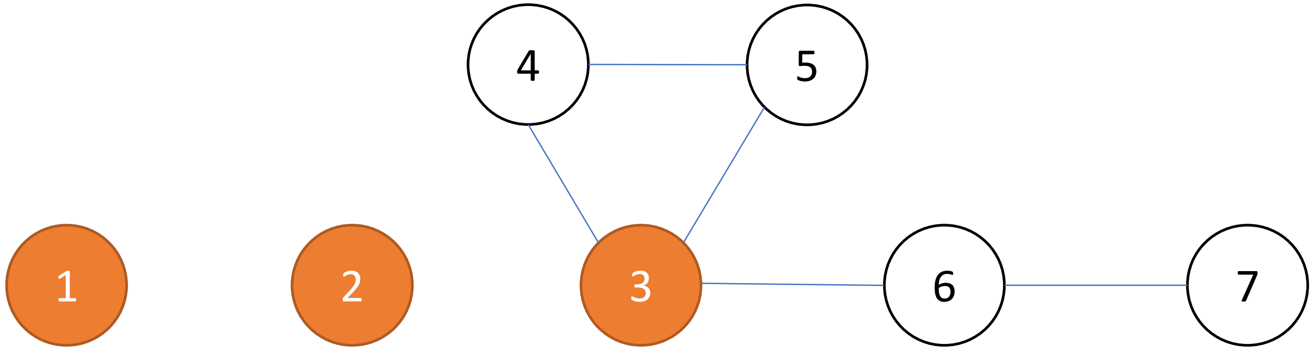
Solution ("in reverse"): {}



Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$



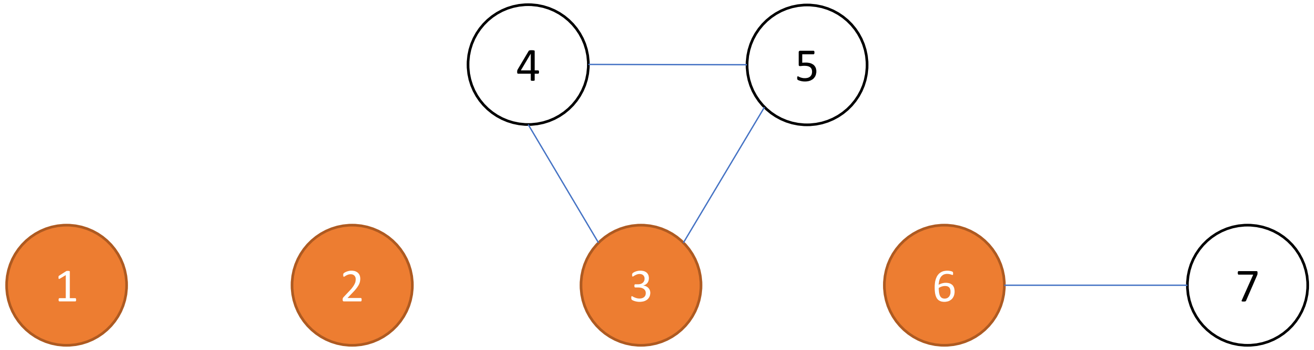
Solution ("in reverse"): {}

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {}

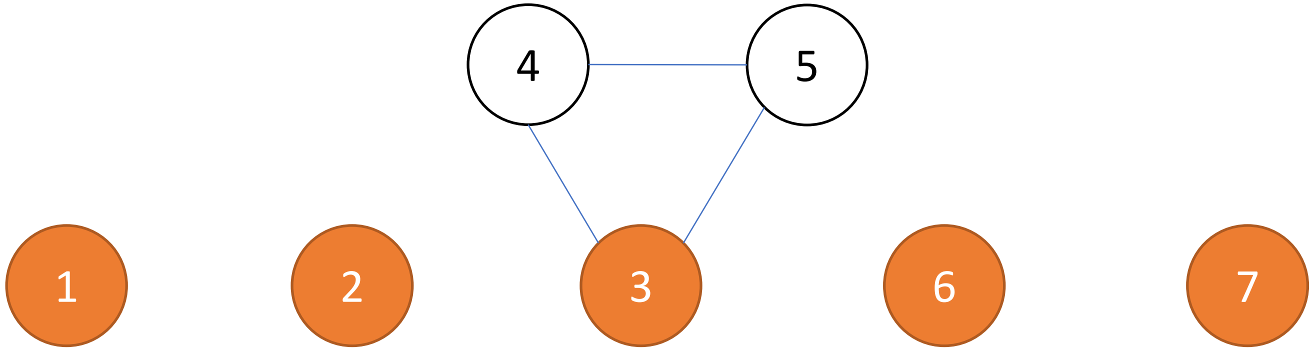


Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7}



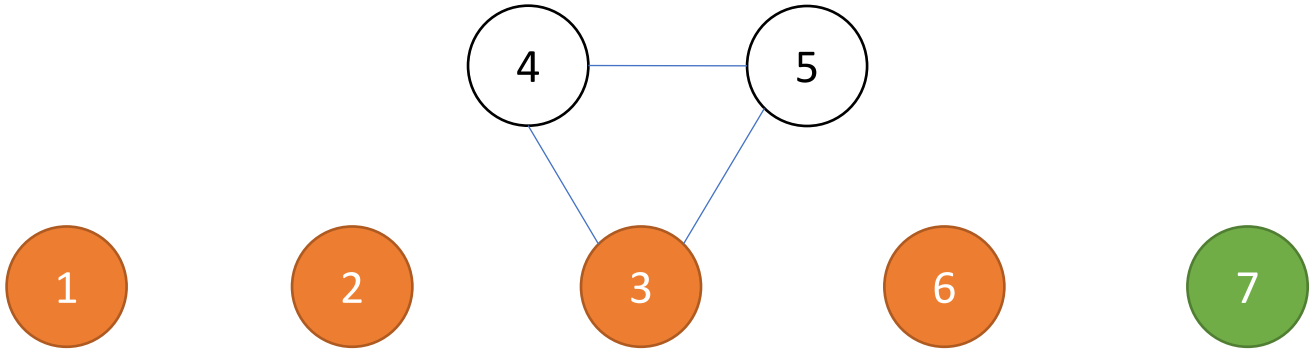
No more edges adjacent to 7,
add to solution and return

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6}



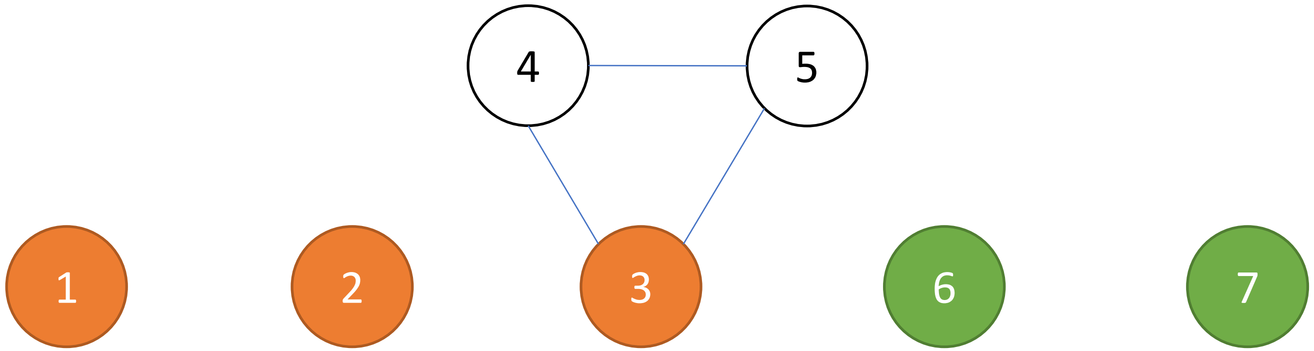
No more edges adjacent to 6,
add to solution and return

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6}



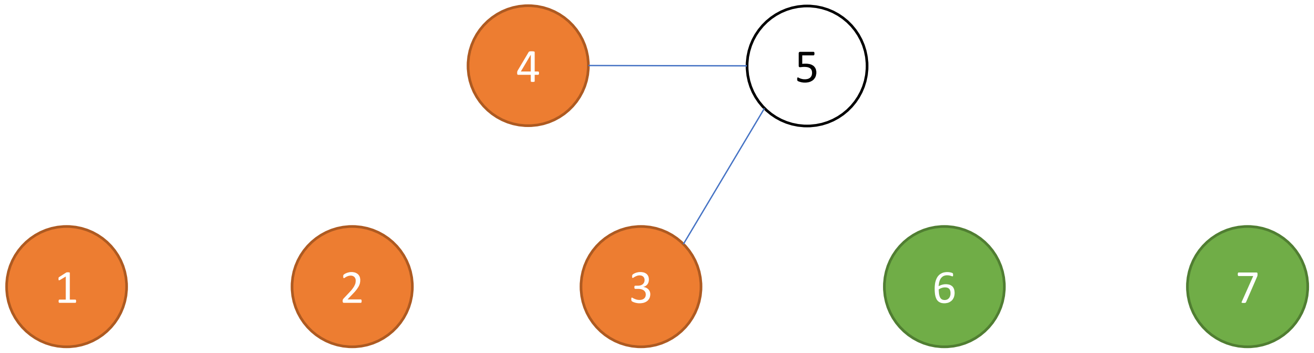
3 still has unexplored
edges, explore them first…

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $\boldsymbol{O(E)}$

Solution ("in reverse"): {7, 6}



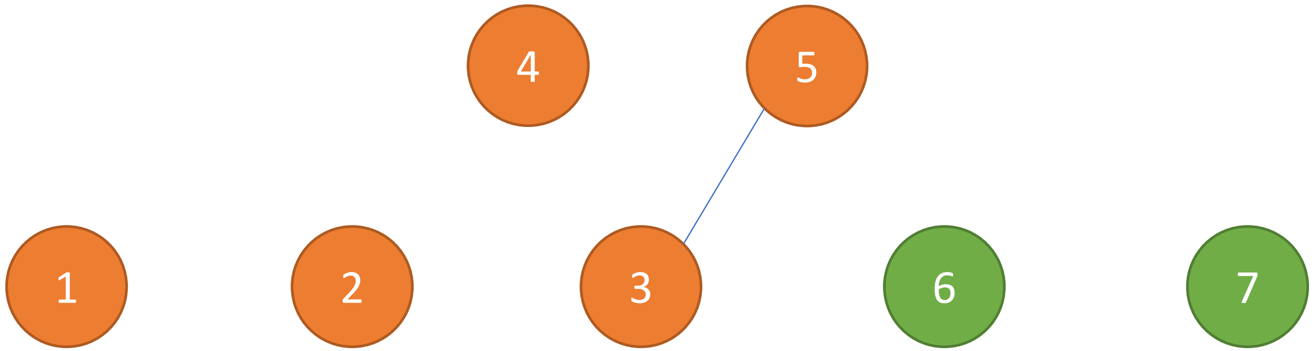
3 still has unexplored
edges, explore them first…

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6}



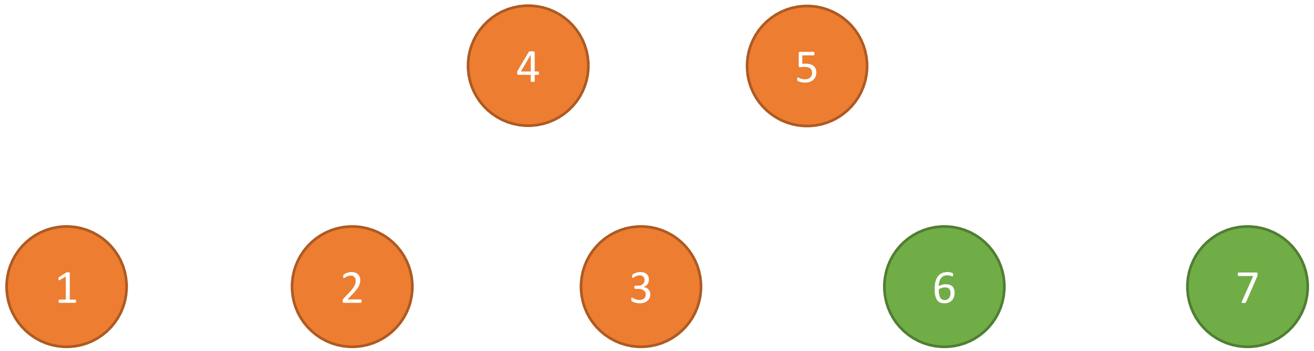
3 still has unexplored
edges, explore them first…

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6, 3}



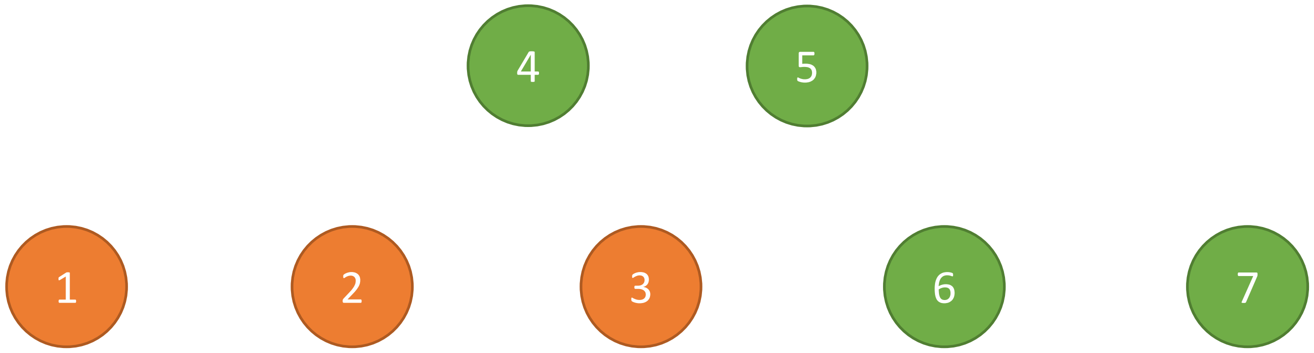
Return to 3 from 5, no
unexplored edges,
add 3 to solution

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6, 3, 5, 4}



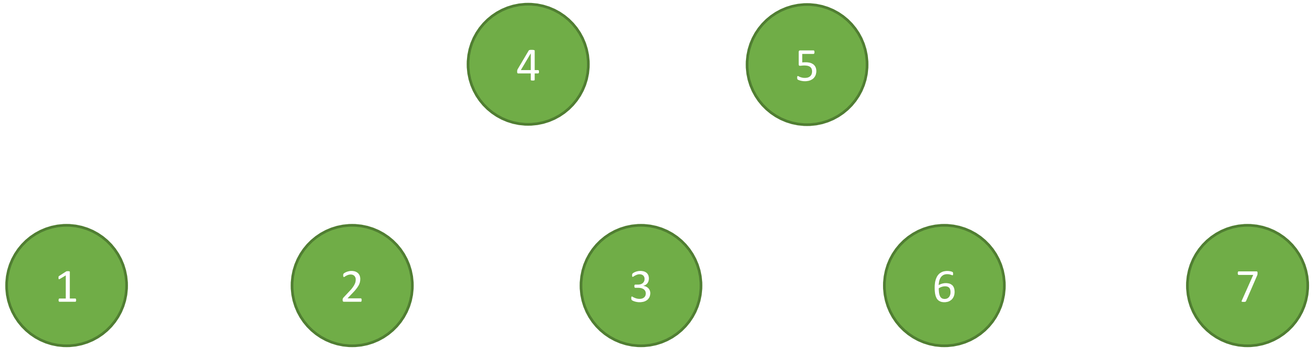
3 still has unexplored edges, explore them first…

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$

Solution ("in reverse"): {7, 6, 3, 5, 4, 3, 2, 1}



Returning to 3, it still has no more unexplored edges, add to solution again, return to 2 & 1

Eulerian Paths and Cycles

Finding Eulerian paths and cycles (once you know they exist)

Finding Eulerian cycles/paths in $O(E)$ Extension

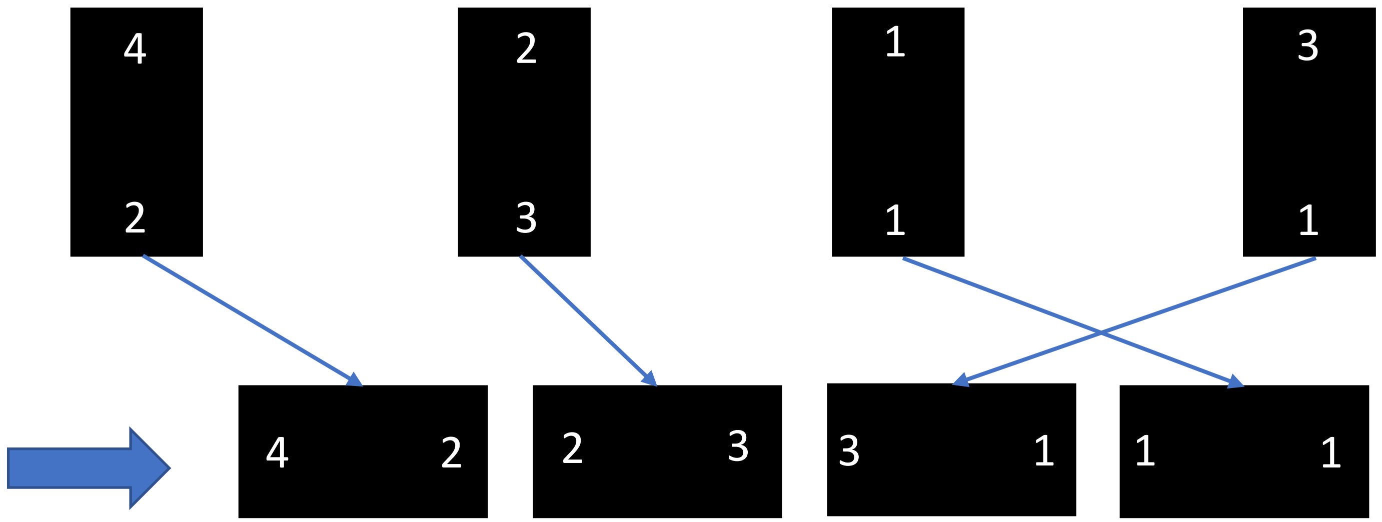
- Try coding this process iteratively (simulate the recursion with a stack) – it's not too difficult!

Eulerian Paths and Cycles

A classical example

Given N dominoes with each having a number written on the bottom and the top, how can you put all the dominoes in a row so that numbers on any two adjacent dominoes, written on their common side, coincide. Dominoes are allowed to turn.

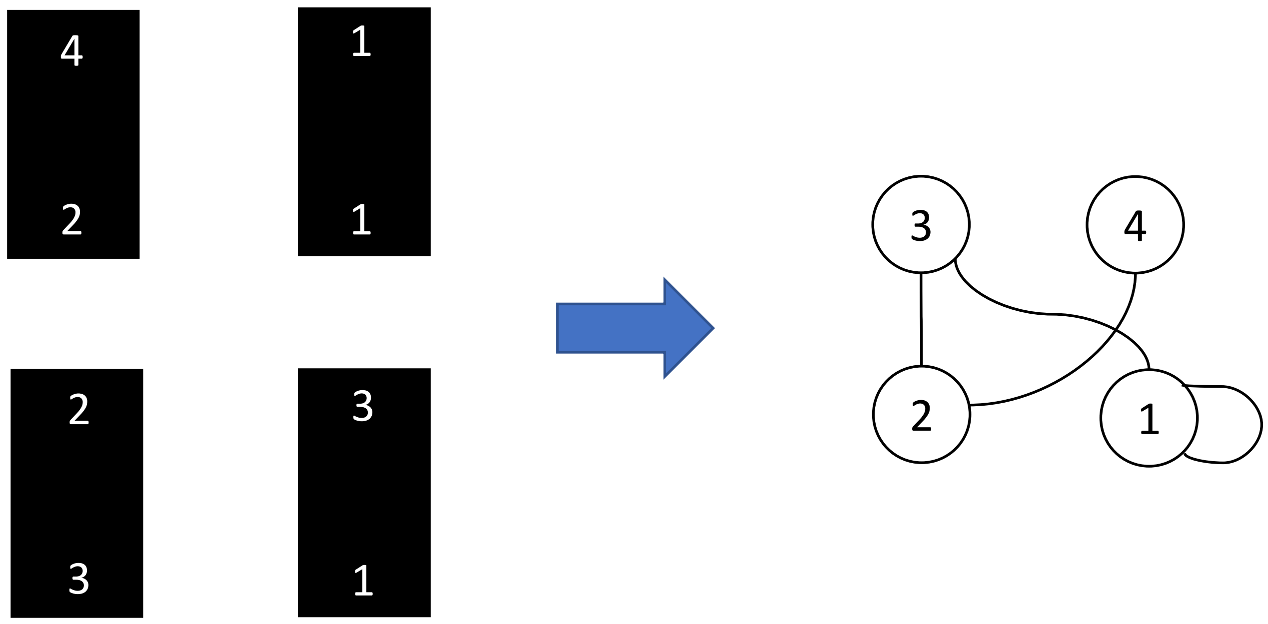
Sample: (N = 4)



Eulerian Paths and Cycles

A classical example

Let the numbers on each domino be a node in a graph, and the domino itself an edge between the bottom number and top number, then our problem is reduced to finding an Eulerian path in this new graph…

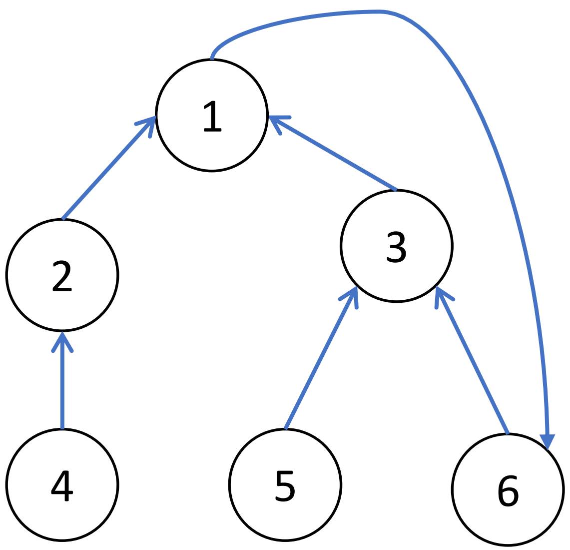


Functional Graphs

Definition: Directed graphs where the outdegree (number of edges leaving a node) of every node is 1.

- Called functional because graph corresponds to a function that defines edges of graph (the “successor function”)
- Can think of it as a rooted tree where every node is directed to its parent, and there’s one additional edge coming out of root

Example:



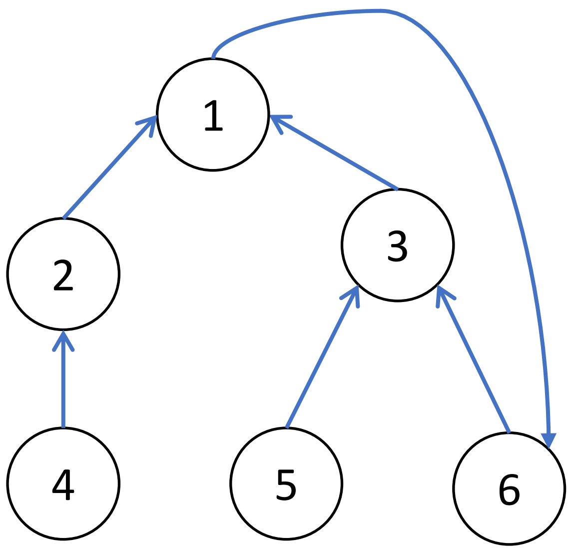
Node *x*	1	2	3	4	5	6
succ(x)	6	1	1	2	3	3

Functional Graphs

Definition: Directed graphs where the outdegree (number of edges leaving a node) of every node is 1.

- Can determine k'th successor in logarithmic time by doing "binary jumps" (you'll see more of these later)
- succ(x, k) = 2^k'th successor of x = $succ(succ(x, k-1), k-1)$ (do two 2^{k-1} jumps)

Example:



Node *x*	**1**	**2**	**3**	**4**	**5**	**6**
succ(x,0)	6	1	1	2	3	3
succ(x,1)	3	6	6	1	1	1
succ(x,2)	6	1	1	3	3	3
succ(x,3)	*a*	6	6	*b*	*b*	*b*

Functional Graphs

Cycle finding in Functional Graphs – Floyd's Algorithm (aka Tortoise and the Hare algorithm)

- Use two pointers *a* and *b*, both pointers begin at some node *x* (starting point of graph)

- On each iteration, *a* walks forward once, *b* walks forward twice. Repeat until two pointers meet each other again.

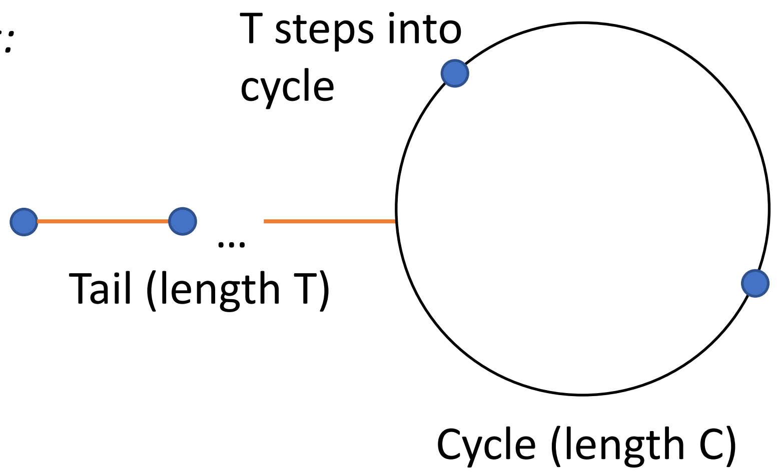
- At this point, *a* has walked forward *k* steps and *b* has walked forward *2k* steps. Since they've met again, this means that the length of the cycle must divide $2k - k = k$…

Functional Graphs

Cycle finding in Functional Graphs – Floyd's Algorithm (aka Tortoise and the Hare algorithm)

- At this point, *a* has walked forward *k* steps and *b* has walked forward *2k* steps. Since they've met again, this means that the length of the cycle must divide *2k – k = k*…

*Hopefully this is intuitive, we can prove it **somewhat** more formally as follows:*



Assume T < C:
After T iterations, *a* enters cycle, *b* is 2T steps into cycle
Right now, *b* is C-T steps behind *a* and get 1 step closer each iteration
After C-T iterations, they will meet
At which point, *b* has travelled 2T + 2(C-T) = 2C steps
and *a* has travelled T + (C-T) = C steps
The difference in their number of steps = 2C – C = C

This is not the full proof because it's possible that C < T (in which case redo the above with modular arithmetic)

Functional Graphs

Cycle finding in Functional Graphs – Floyd's Algorithm (aka Tortoise and the Hare algorithm)

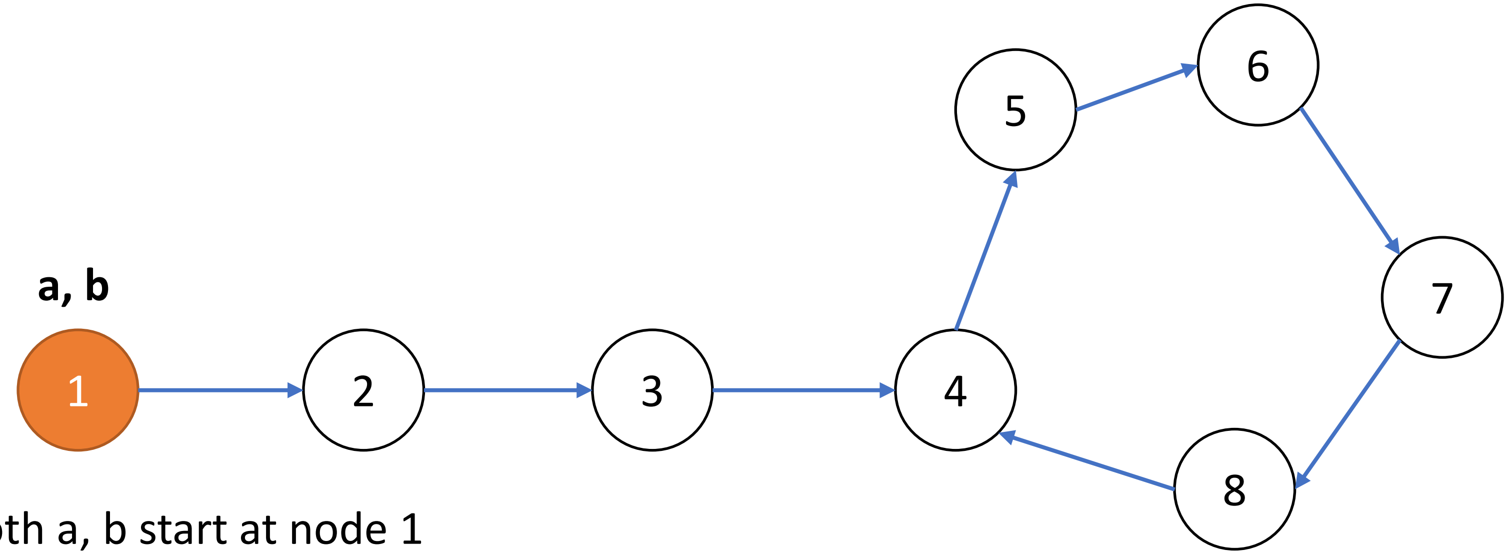
- Now *b* is *2k* steps into the graph and *a* is *k* steps (they are at the same location). Since the length of the cycle divides *k*, we can move *a* back to the starting point *x* and start moving both pointers at 1 step per iteration.

- In *k* more steps after this, they will both be back to their original meeting position. Since they've "walked together" to the meeting position, the first place they meet will be the first node in the cycle.

- After this, we can calculate the length of the cycle simply by moving one pointer around until it returns to the start again

Functional Graphs

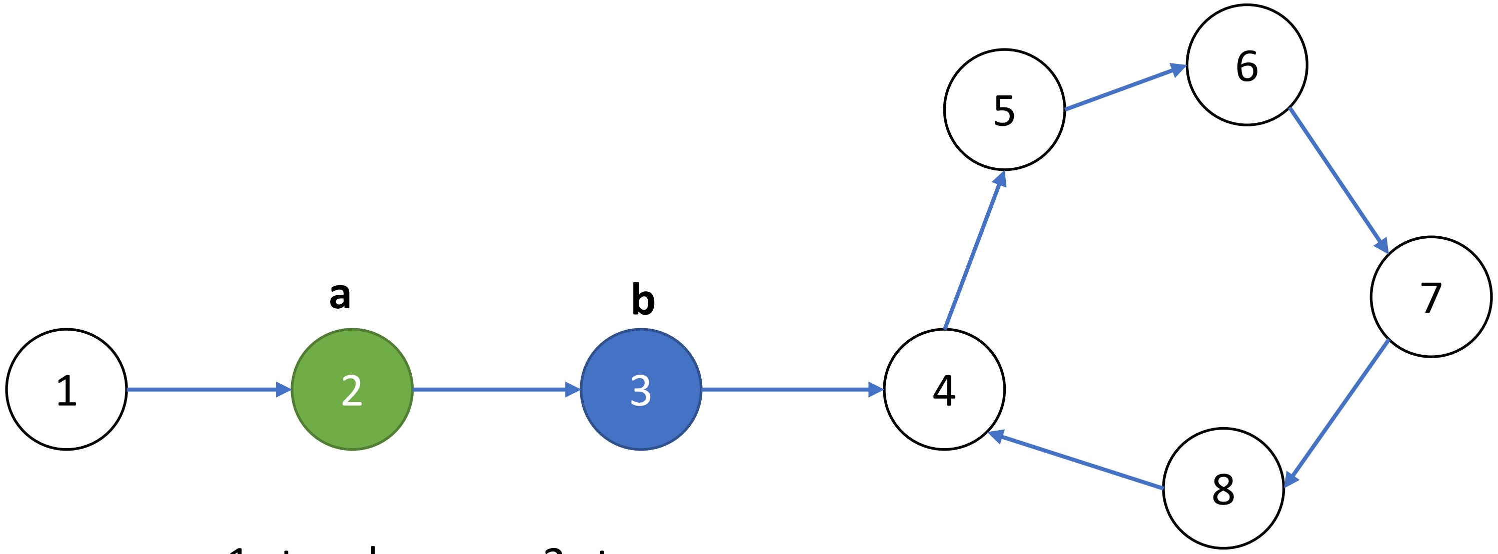
Tortoise and the Hare algorithm – stage 1



Both a, b start at node 1

Functional Graphs

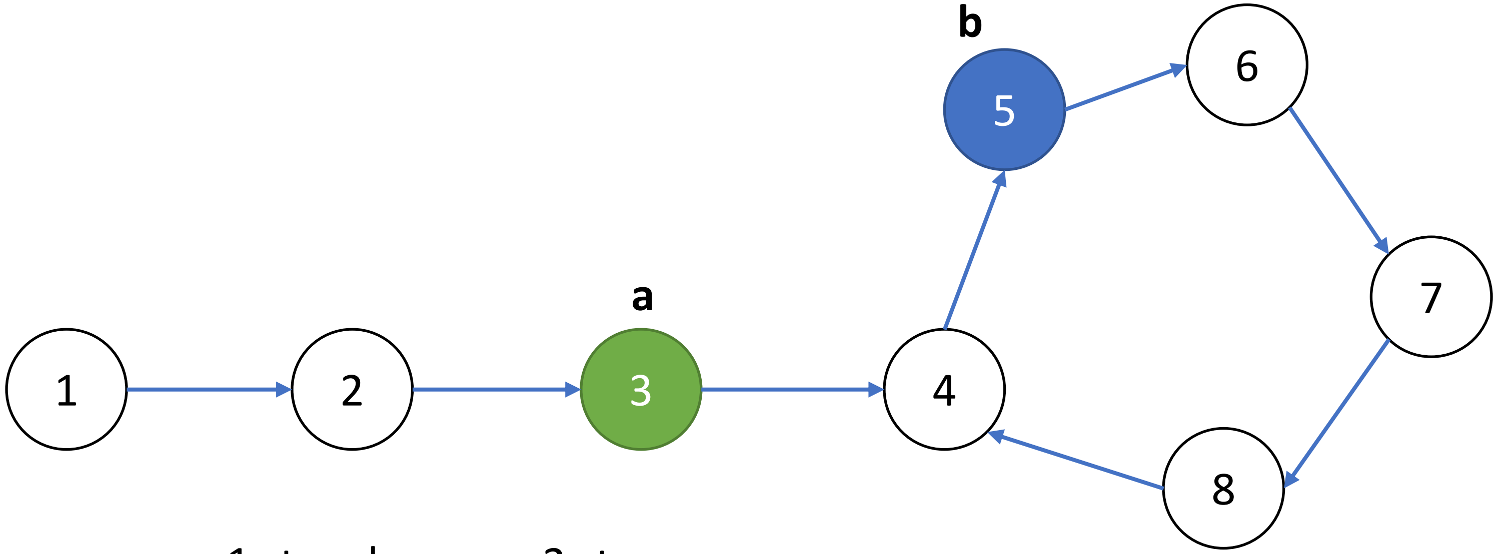
Tortoise and the Hare algorithm – stage 1



a moves 1 step, b moves 2 steps

Functional Graphs

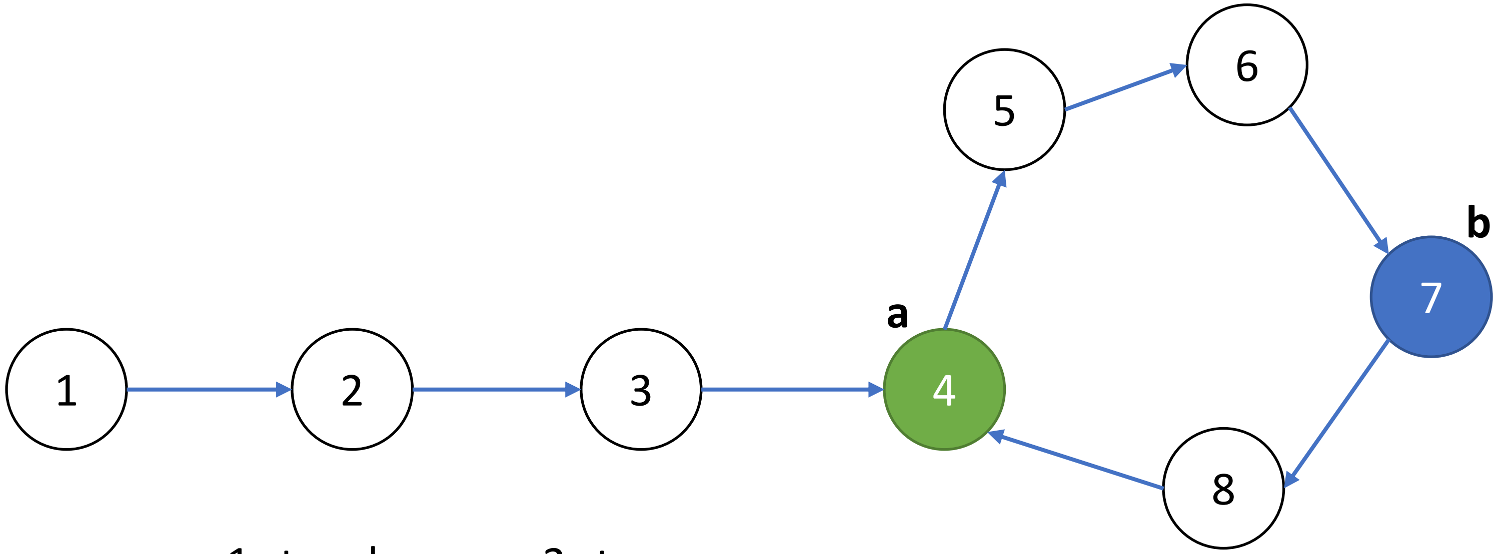
Tortoise and the Hare algorithm – stage 1



a moves 1 step, b moves 2 steps

Functional Graphs

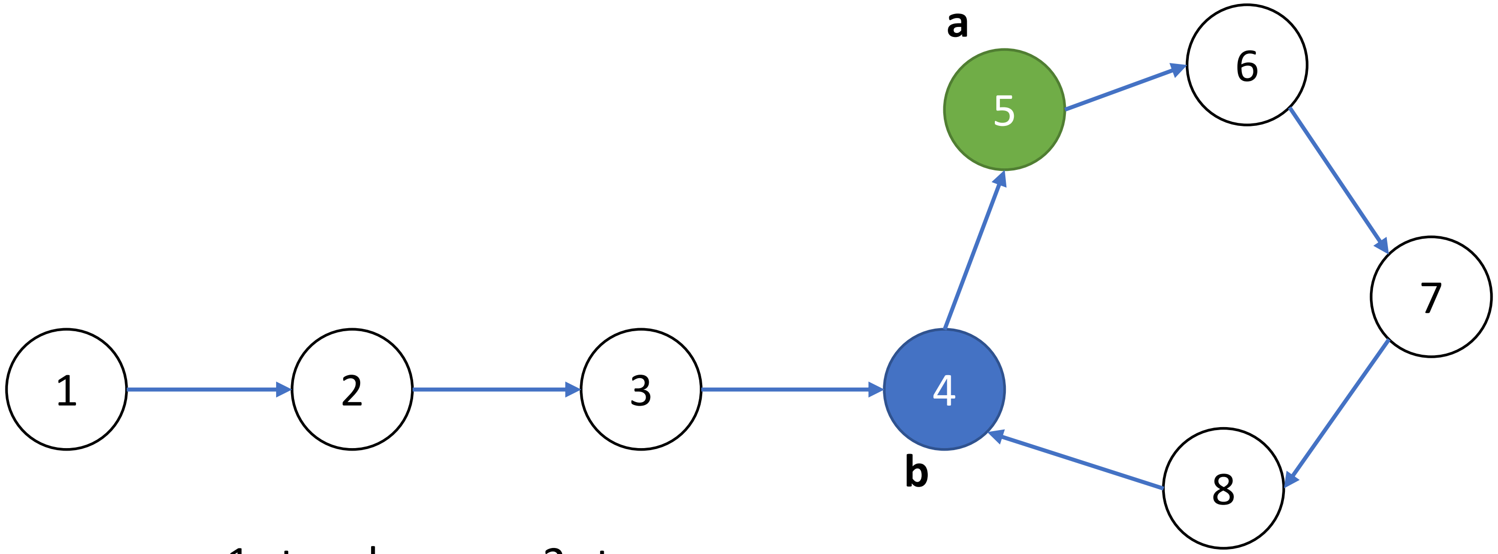
Tortoise and the Hare algorithm – stage 1



a moves 1 step, b moves 2 steps

Functional Graphs

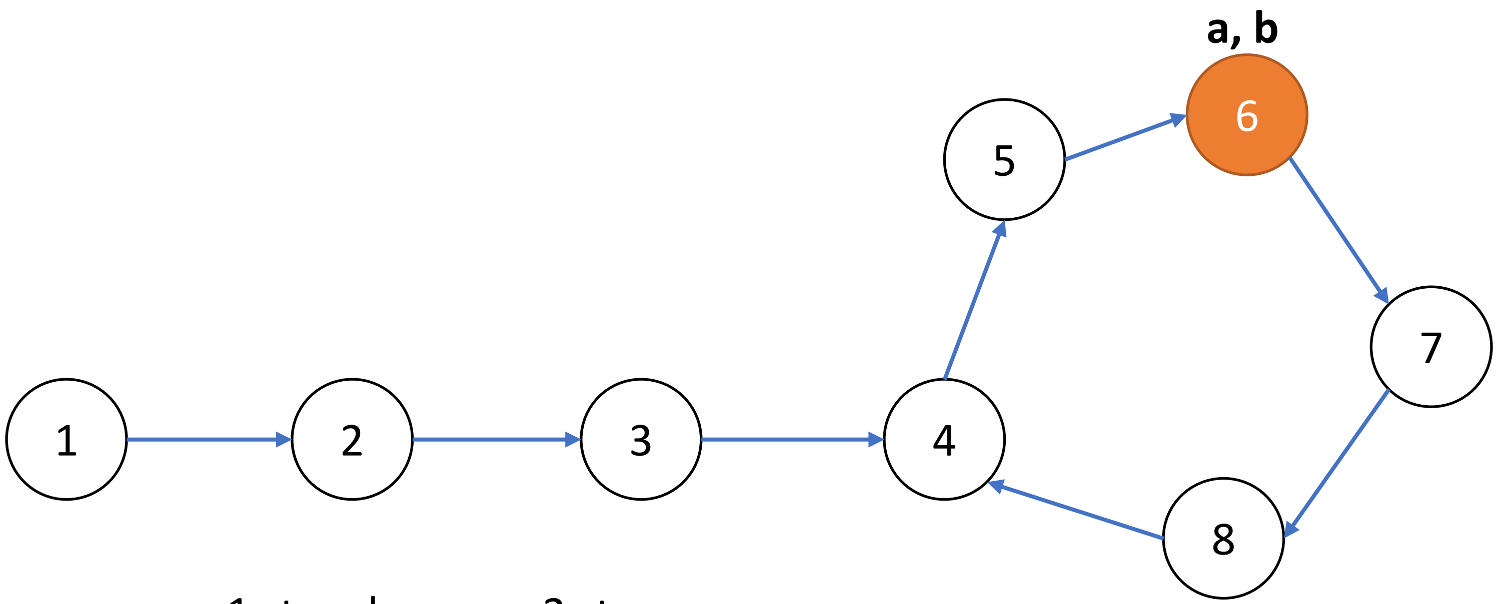
Tortoise and the Hare algorithm – stage 1



a moves 1 step, b moves 2 steps

Functional Graphs

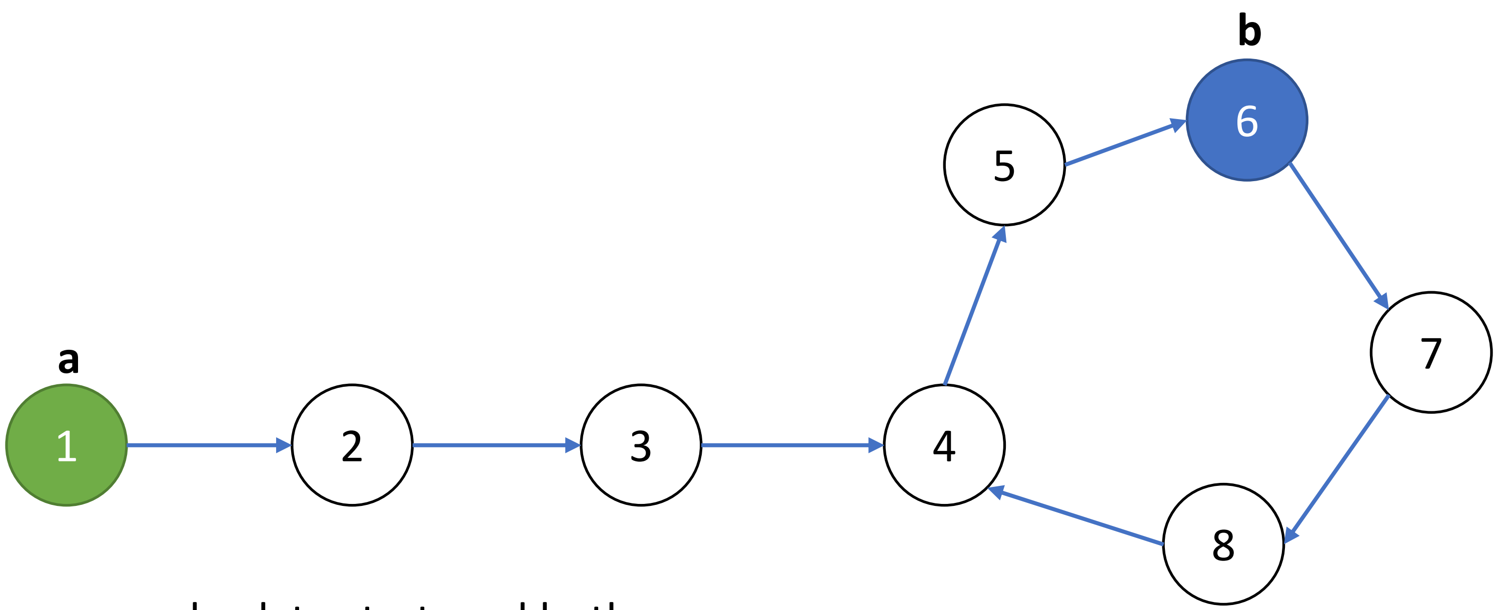
Tortoise and the Hare algorithm – stage 1



a moves 1 step, b moves 2 steps

Functional Graphs

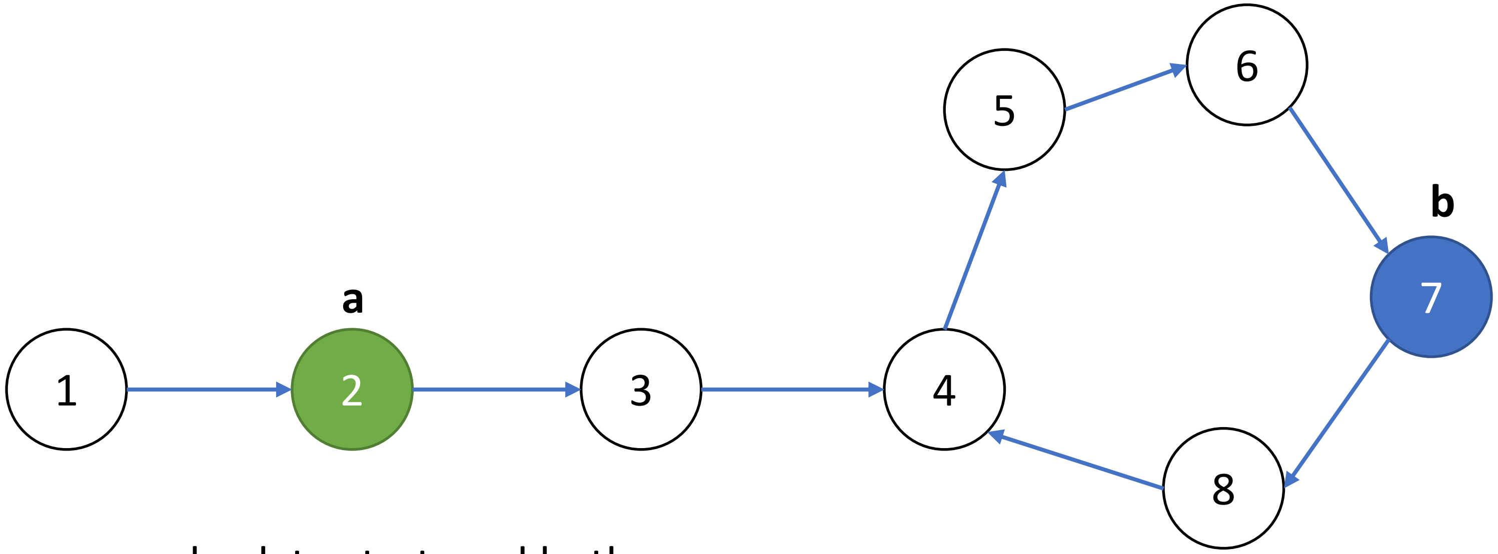
Tortoise and the Hare algorithm – stage 2



a goes back to start, and both
nodes move 1 step per iteration

Functional Graphs

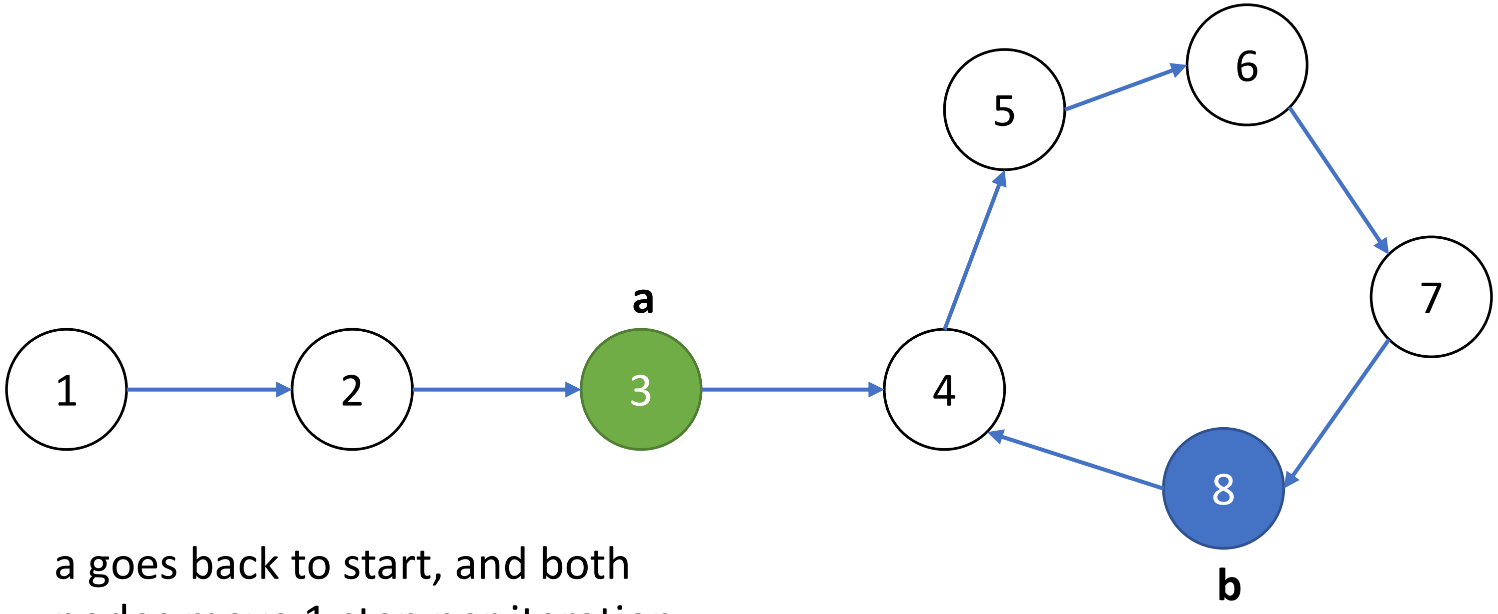
Tortoise and the Hare algorithm – stage 2



a goes back to start, and both
nodes move 1 step per iteration

Functional Graphs

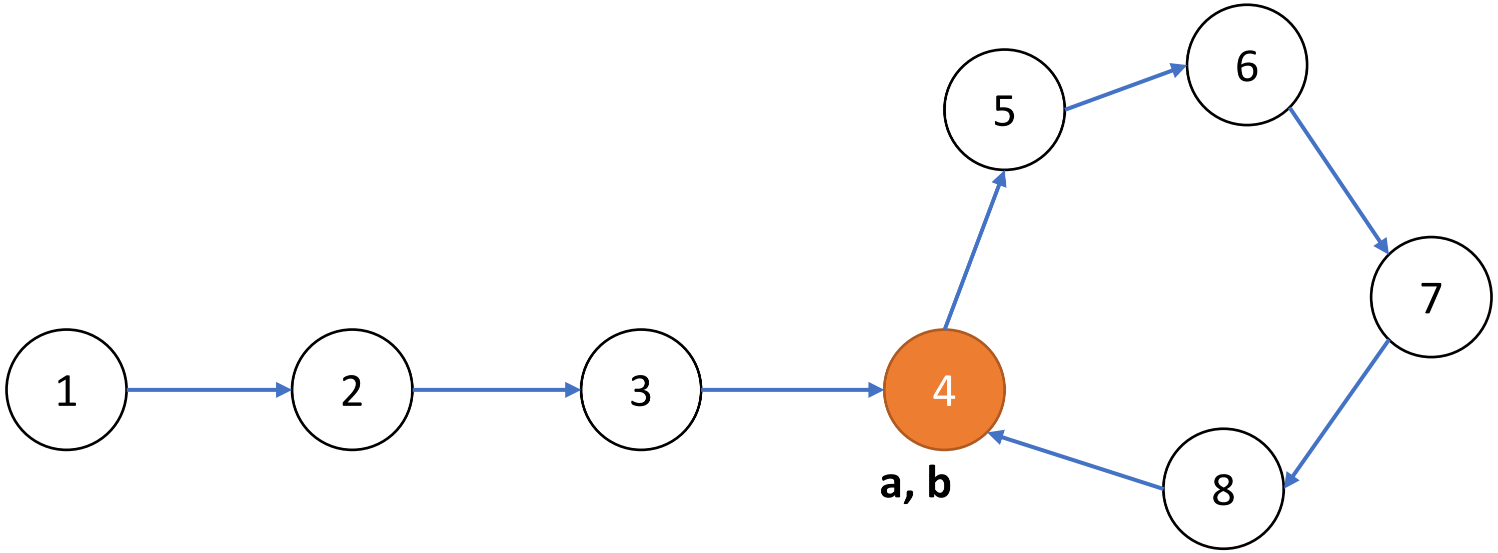
Tortoise and the Hare algorithm – stage 2



a goes back to start, and both
nodes move 1 step per iteration

Functional Graphs

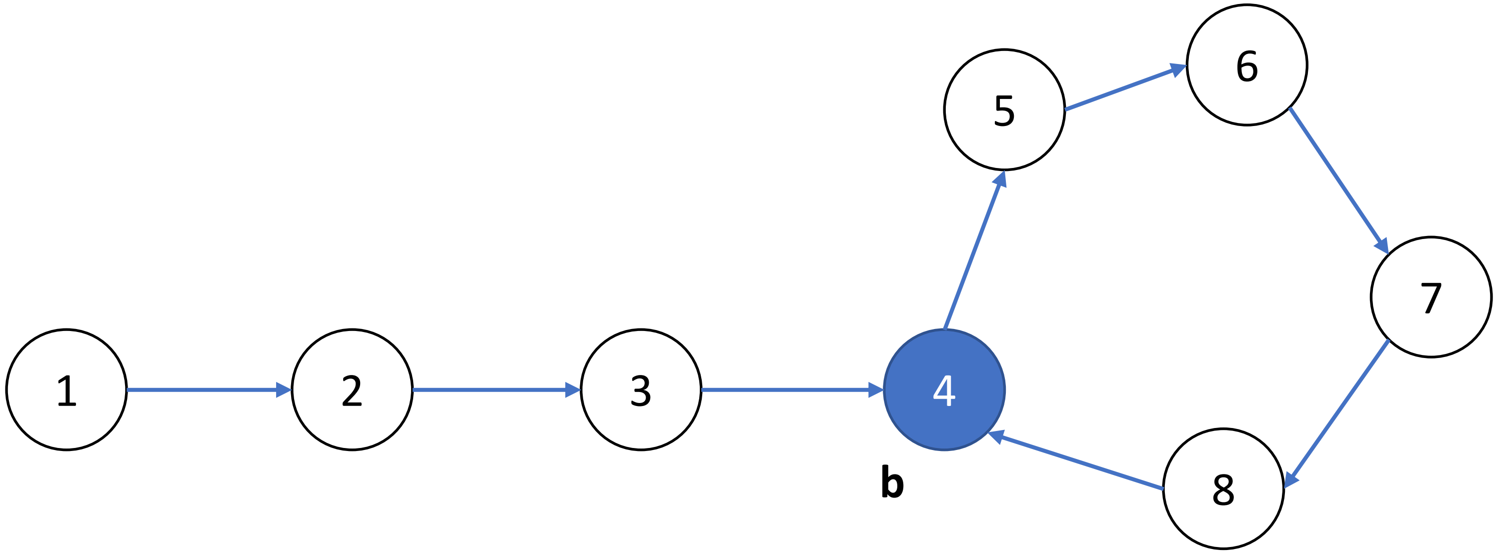
Tortoise and the Hare algorithm – stage 2



The common point (node 4) is the first node in the cycle

Functional Graphs

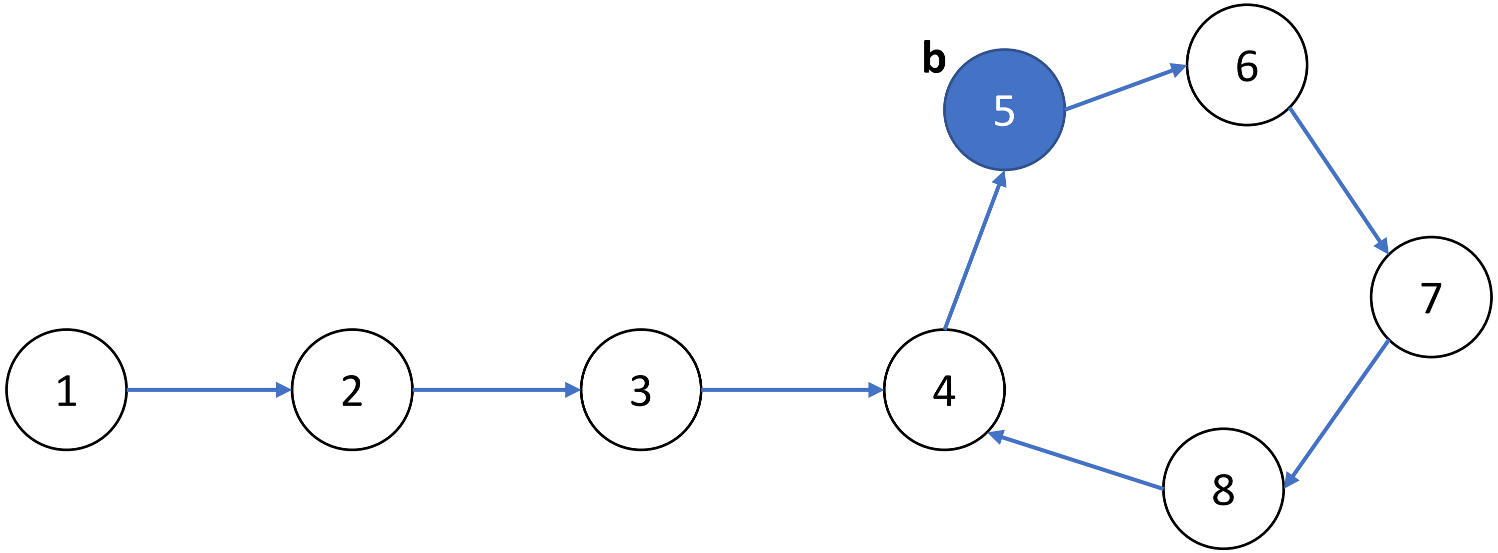
Tortoise and the Hare algorithm – stage 3



We send b off to find the length
of the cycle on its own

Functional Graphs

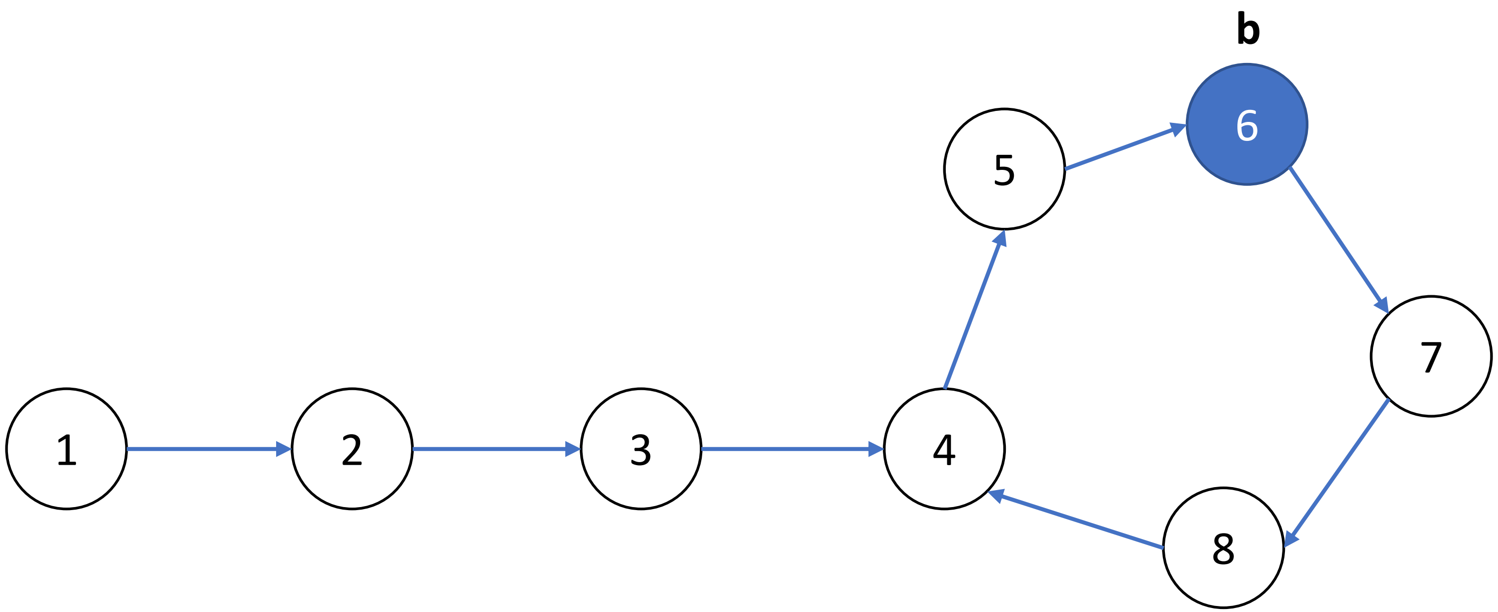
Tortoise and the Hare algorithm – stage 3



We send b off to find the length
of the cycle on its own

Functional Graphs

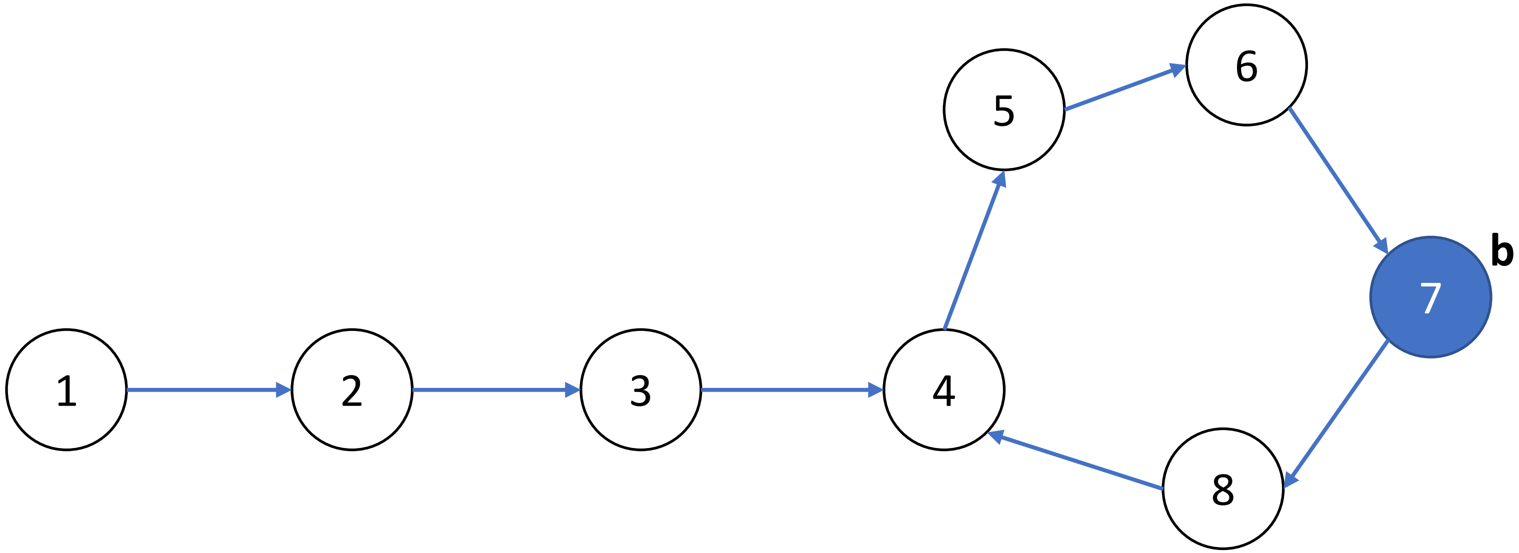
Tortoise and the Hare algorithm – stage 3



We send b off to find the length
of the cycle on its own

Functional Graphs

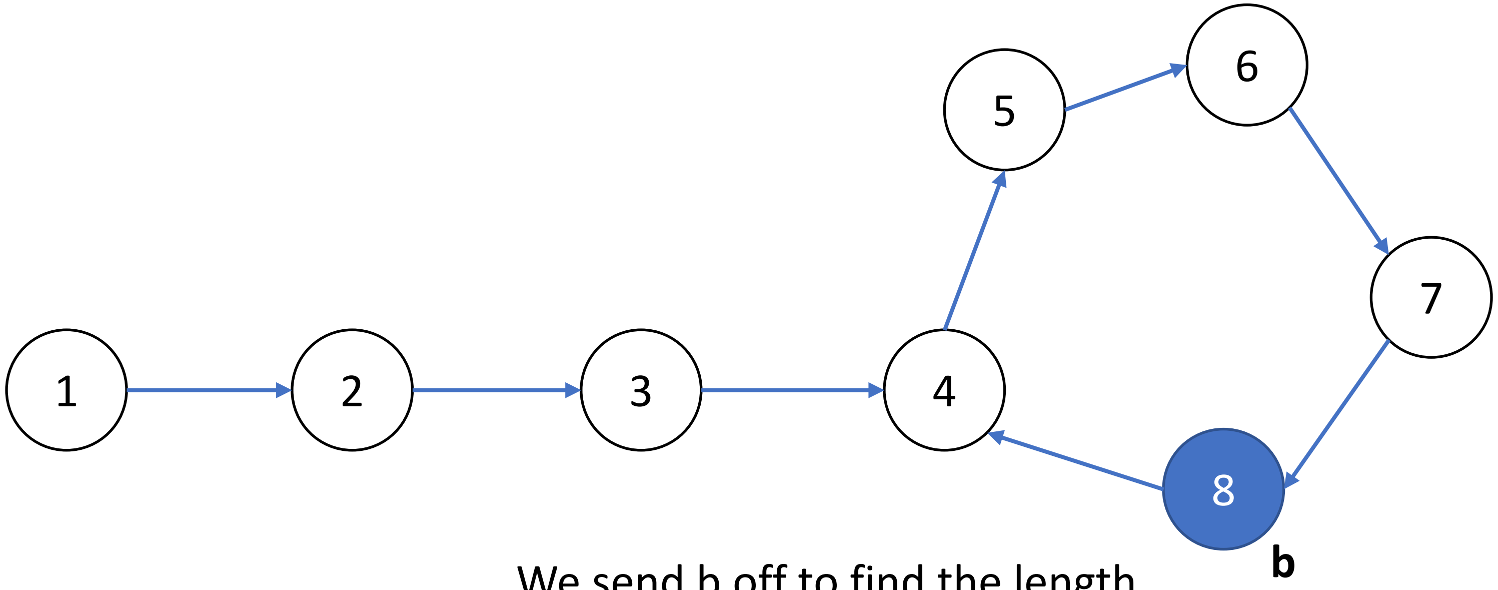
Tortoise and the Hare algorithm – stage 3



We send b off to find the length of the cycle on its own

Functional Graphs

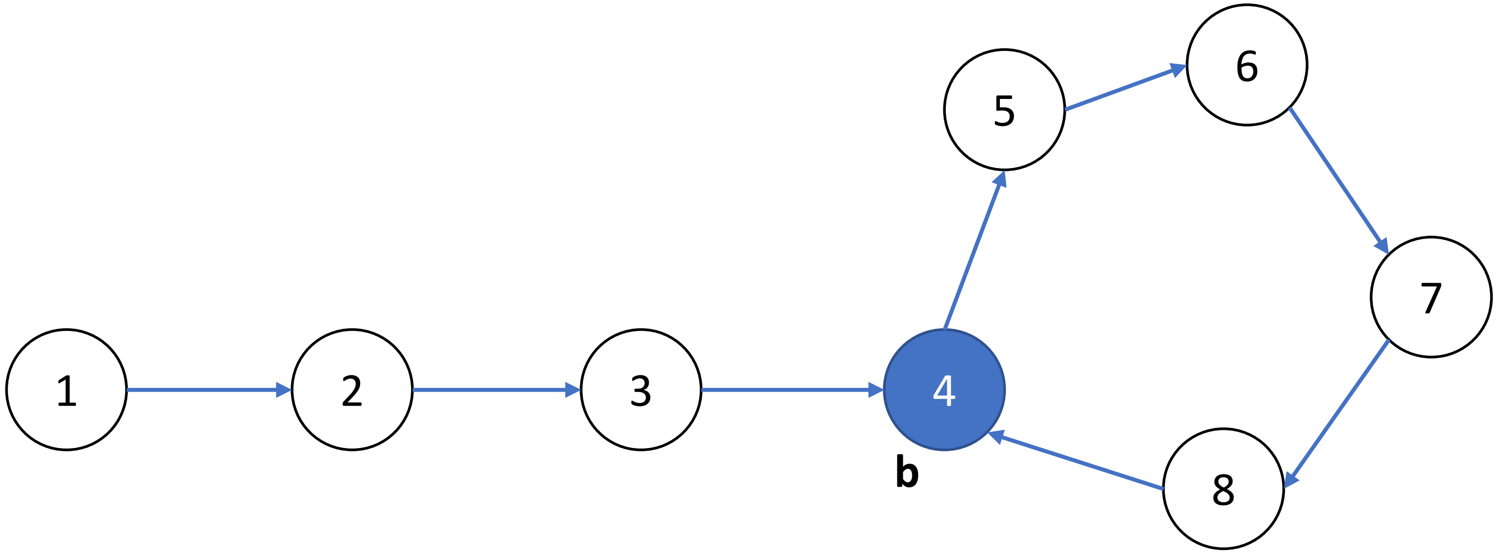
Tortoise and the Hare algorithm – stage 3



We send b off to find the length of the cycle on its own

Functional Graphs

Tortoise and the Hare algorithm – stage 3



Once b returns, we know the length of the cycle (by how many steps it took)

Functional Graphs

Cycle finding in Functional Graphs – Floyd's Algorithm (aka Tortoise and the Hare algorithm)

Pseudocode:

```
// Stage 1
a = succ(x)
b = succ(succ(x))

while a != b:
       a = succ(a)
       b = succ(succ(b)

// Find first node in cycle
a = x
while a != b:
       a = succ(a)
       b = succ(b)
first = a
```

Functional Graphs

Cycle finding in Functional Graphs – Floyd's Algorithm (aka Tortoise and the Hare algorithm)

Pseudocode:

```
// Find length of cycle
b = succ(first)
length = 1
while a != b:
        b = succ(b)
        length++
```

Practice problems

Core:

1) More Highways: https://orac2.info/problem/graphfloyd/
2) Highway Travelling (with Bellman-Ford, not Dijkstras): https://orac2.info/problem/graphhighways/
3) Konigsberg: https://orac2.info/problem/graphkonig/

Additional:

1) (Negative) Cycle Finding: https://cses.fi/problemset/task/1197