# Week 8: Introduction to Graph Theory

## Algorithms & Data Structures

Thorben Klabunde

www.th-kl.ch

November 10, 2025

# Agenda

1. Mini-Quiz
2. Assignment
3. What is a Graph?
4. Properties of Graphs
5. Eulerian Walks
6. Additional Practice
7. Peer Grading

# Mini-Quiz

# Assignment

**Exercise 7.1** *1-3 subset sums* **(1 point)**.

Let $A[1, \ldots, n]$ be an array containing $n$ positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1, 2, \ldots, n\}$, together with multipliers $c_i \in \{1, 3\}$, $i \in I$ such that:

$$b = \sum_{i \in I} c_i \cdot A[i].$$

If this is possible, we say $b$ is a 1-3 subset sum of $A$. For example, if $A = [16, 4, 2, 7, 11, 1]$ and $b = 61$, we could write $b = 3 \cdot 16 + 4 + 3 \cdot 2 + 3 \cdot 1$.

Describe a DP algorithm that, given an array $A[1, \ldots, n]$ of positive integers, and a positive integer $b \in \mathbb{N}$ returns True if and only if $b$ is a 1-3 subset sum of $A$. Your algorithm should have asymptotic runtime complexity at most $O(b \cdot n)$.

**Exercise 7.1** *1-3 subset sums* **(1 point)**.

Let $A[1, \ldots, n]$ be an array containing $n$ positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1, 2, \ldots, n\}$, together with multipliers $c_i \in \{1, 3\}$, $i \in I$ such that:

$$b = \sum_{i \in I} c_i \cdot A[i].$$

If this is possible, we say $b$ is a 1-3 subset sum of $A$. For example, if $A = [16, 4, 2, 7, 11, 1]$ and $b = 61$, we could write $b = 3 \cdot 16 + 4 + 3 \cdot 2 + 3 \cdot 1$.

Describe a DP algorithm that, given an array $A[1, \ldots, n]$ of positive integers, and a positive integer $b \in \mathbb{N}$ returns True if and only if $b$ is a 1-3 subset sum of $A$. Your algorithm should have asymptotic runtime complexity at most $O(b \cdot n)$.

3. *Recursion*: $DP$ can be computed recursively as follows:

$$DP[0][s] = \texttt{False} \qquad 1 \le s \le b \qquad (1)$$

$$DP[a][0] = \texttt{True} \qquad 0 \le a \le b \qquad (2)$$

$$DP[a][s] = DP[a-1][s] \texttt{ or } DP[a-1][s - A[a]] \texttt{ or } DP[a-1][s - 3 \cdot A[a]] \qquad 1 \le a \le n, \quad 1 \le s \le b. \qquad (3)$$

Note that in equation (3), the entries '$DP[a-1][s - A[a]]$' and '$DP[a-1][s - 3 \cdot A[a]]$' might fall outside the range of the table, in which case we treat them as `False`.

**Exercise 7.3** *Road trip.*

You are planning a road trip for your summer holidays. You want to start from city $C_0$, and follow the only road that goes to city $C_n$ from there. On this road from $C_0$ to $C_n$, there are $n-1$ other cities $C_1, \ldots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \ldots, C_{n-1}$ are on the road from $C_0$ to $C_n$). For each $0 \leq i \leq n$, the city $C_i$ is at kilometer $k_i$ of the road for some given $0 = k_0 < k_1 < \ldots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \ldots, C_{n-1}$ you will make an additional stop (you will stop in $C_0$ and $C_n$ anyway). However, you do not want to drive more than $d$ kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city $C_i$ you can only go forward to cities $C_j$ with $j > i$).

1. *Dimensions of the DP table*: The DP table is linear, and its size is $n + 1$.
2. *Subproblems*: $DP[i]$ is the number of possible routes from $C_0$ to $C_i$ (which stop at $C_i$).
3. *Recursion*: Initialize $DP[0] = 1$.

   For every $i > 0$, we can compute $DP[i]$ using the formula

   $$DP[i] = \sum_{\substack{0 \le j < i \\ k_i \le k_j + d}} DP[j]. \tag{4}$$

   This recursion is correct since city $C_i$ can be reached from $C_j$ (for $0 \le j < i$) if and only if $k_i \le k_j + d$. Summing up the number of routes from $C_0$ to these $C_j$, we get the number of routes from $C_0$ to $C_i$.
4. *Calculation order*: We can calculate the entries of $DP$ as follows:

   **for** $i = 0 \ldots n$ **do**
   Compute $DP[i]$.
5. *Extracting the solution*: All we have to do is read the value at $DP[n]$.
6. *Running time*: For $i = 0$, $DP[0]$ is computed in $O(1)$ time. For $i \ge 1$, the entry $DP[i]$ is computed in $O(i)$ time (as we potentially need to take the sum of $i$ entries). Therefore, the total runtime is $O(1) + \sum_{i=1}^{n} O(i) = O(n^2)$.

**Exercise 7.4** *String counting* **(1 point)**.

Given a binary string $S \in \{0,1\}^n$ of length $n$, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{“}\underline{11}10\underline{1}\text{1”}) = 3$ because the string contains three 1 (underlined) that are followed by another 1. Given $n$ and $k$, the goal is to count the number of binary strings $S$ of length $n$ with $f(S) = k$.

Describe a DP algorithm that, given positive integers $n$ and $k$ with $k < n$, reports the required number. Your solution should have complexity at most $O(nk)$.

1. *Dimensions of the DP table*: $DP[1 \dots n][0 \dots k][0 \dots 1]$.

2. *Subproblems*: The entry $DP[i][j][l]$ describes the number of strings of length $i$ with $j$ occurrences of "11" that end in $l$.

3. *Recursion*: The base cases for $i = 1$ are given by $DP[1][0][0] = 1$ (the string "0"), $DP[1][0][1] = 1$ (the string "1"), $DP[1][j][0] = 0$ and $DP[1][j][1] = 0$ for $1 \leq j \leq k$. The update rule is as follows: For $1 < i \leq n$ and $0 \leq j \leq k$, to get a string of length $i$ with $j$ occurrences of "11" ending in 0, we can append "0" to a string of length $i - 1$ with $j$ occurrences of "11" ending in 0 or 1, which gives

$$DP[i][j][0] = DP[i-1][j][0] + DP[i-1][j][1].$$

To get a string of length $i$ with $j$ occurrences of "11" ending in 1, we can append "1" to a string of length $i-1$ with $j$ occurrences of "11" ending in 0 or to a string of length $i-1$ with $j-1$ occurrences of "11" ending in 1 (if $j > 0$). Thus,

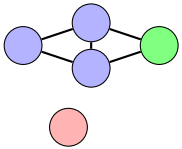
$$DP[i][j][1] = \begin{cases} DP[i-1][j][0], & \text{if } j = 0 \\ DP[i-1][j][0] + DP[i-1][j-1][1], & \text{if } j > 0. \end{cases}$$

# What is a Graph?

# Motivation: Graphs Are Everywhere

**Three examples of real-world problems:**

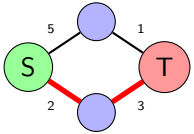
**1. Reachability**



**Social Networks**

"Can Alice reach Bob through friends?"

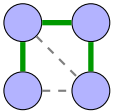
**2. Shortest Paths**




**Navigation**

"Fastest route from home to university?"

**3. Spanning Trees**



**Network Design**

"Connect all cities with minimum railway length"

**Key Insight**

Graphs provide a **unifying language** to model relationships and solve a large number of problems!

# Definition: What is a Graph?

**Definition 1.1 (Graph)**

A **graph** $G$ is a pair $G = (V, E)$, where:

- $V$ is a finite, non-empty set of **vertices** (or *nodes*)
- $E \subseteq \binom{V}{2} := \{\{x, y\} \mid x, y \in V, x \neq y\}$ is a set of **edges**

- An edge $\{u, v\} \in E$ connects vertices $u$ and $v$
- We write $\{u, v\}$ for an edge

**Note:** By this definition, we exclude loops (edges from a vertex to itself) and multiple edges between the same pair of vertices.

# Example: A Simple Graph

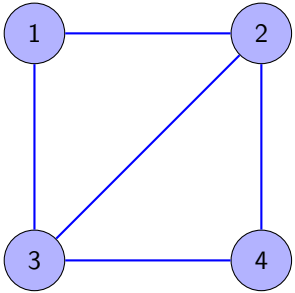
**Graph** $G = (V, E)$

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

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

**Properties:**

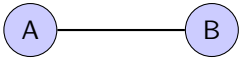
- 4 vertices: $|V| = 4$
- 5 edges: $|E| = 5$



# Directed vs. Undirected Graphs

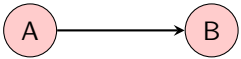
**Undirected Graph**

- Edges have no direction
- $\{u, v\} = \{v, u\}$
- Examples: friendships, physical connections




**Directed Graph (Digraph)**

- Edges have direction (arrows)
- $(u, v) \neq (v, u)$
- Examples: one-way streets



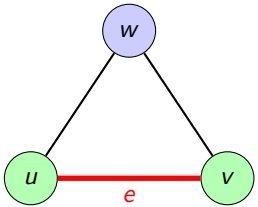

**Today's focus:** We'll primarily work with **undirected graphs**, but the concepts extend naturally to directed graphs.

# Key Definitions

## Adjacency and Incidence

Let $G = (V, E)$ be a graph, $u, v \in V$, and $e \in E$.

- Vertices $u$ and $v$ are **adjacent** if $\{u, v\} \in E$
- $u$ and $v$ are called the **endpoints** of edge $e = \{u, v\}$
- A vertex $u$ and an edge $e$ are **incident** if $u$ is an endpoint of $e$



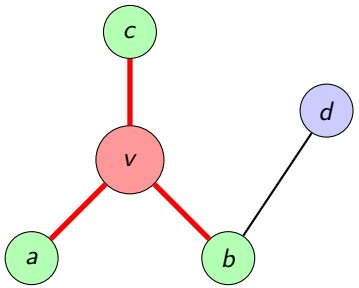

- $u$ and $v$ are **adjacent**
- $e$ is **incident** to both $u$ and $v$
- $u$ and $w$ are also adjacent

# Neighborhood and Degree

**Definition 1.2 (Neighborhood and Degree)**

For a vertex $v \in V$ in graph $G = (V, E)$:

- The **neighborhood** of $v$ is: $N_G(v) := \{u \in V \mid \{v, u\} \in E\}$
- The **degree** of $v$ is: $\deg_G(v) := |N_G(v)|$




**For vertex $v$:**

- $N(v) = \{a, b, c\}$
- $\deg(v) = 3$

**Other degrees:**

- $\deg(a) = 1$
- $\deg(b) = 2$
- $\deg(c) = 1$
- $\deg(d) = 1$

# Connectedness and Connected Components

**Definition: Reachability**

A vertex $u$ **reaches** $v$ (written $u \rightsquigarrow v$) if there exists a path from $u$ to $v$.

**Definition: Connected**

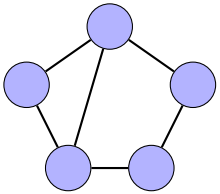
A graph $G$ is **connected** if every vertex reaches every other vertex.

**Definition: Connected Component**

A **connected component** is a maximal connected subgraph and the equivalence class of the "reaches" relation.

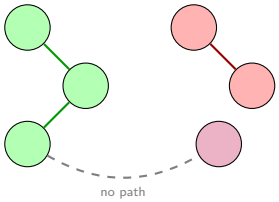
**Note:** Connected components partition the vertices. Each vertex belongs to exactly one component.

**Connected Graph**



1 connected component

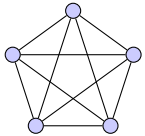
**Disconnected Graph**




3 connected components

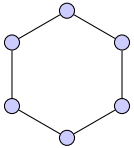
# Special Types of Graphs

**Complete Graph** $K_n$



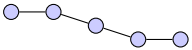
All vertices pairwise connected
$K_5$ (5 vertices, 10 edges)

**Cycle** $C_n$



Circular connection
$C_6$ (6 vertices, 6 edges)

**Path** $P_n$



Linear connection
$P_5$ (5 vertices, 4 edges)

These are **canonical examples** that appear frequently in graph theory and algorithms.

# Your Turn: Maximum Number of Edges

**Determine** the maximum number of edges a Graph $G = (V, E)$ can have, i.e., the number of edges in the complete graph $K_n$ in terms of $n$.

*Discuss with your neighbor (2-3 minutes)*

# Special Types of Graphs - Trees

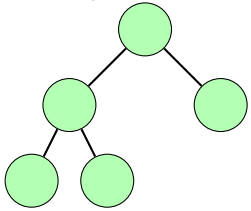
**Definition 1.4 (Tree)**

An undirected graph $G = (V, E)$ is a **tree** if it is:

- **Connected**, and
- **Acyclic** (contains no cycles).

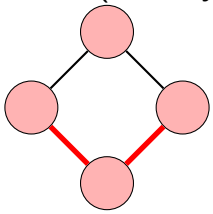
A **forest** is an acyclic graph (i.e., a collection of disjoint trees).

**Example: A Tree**



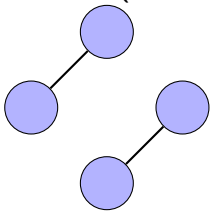
Connected and Acyclic

**Not a Tree (Has a cycle)**



Connected, but not acyclic

**Not a Tree (A forest)**

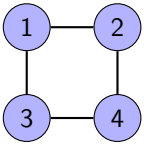


Acyclic, but not connected

# How to Represent a Graph?

**Two main data structures for storing graphs:**

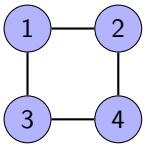
**Adjacency Matrix**




$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$A[i][j] = 1$ if edge $\{i, j\}$ exists

**Adjacency List**




1: $\{2, 3\}$
2: $\{1, 4\}$
3: $\{1, 4\}$
4: $\{2, 3\}$

Each vertex stores its neighbors

# Adjacency Matrix vs. Adjacency List: The Tradeoff

| Operation | Adjacency Matrix | Adjacency List |
|---|---|---|
| Space | $O(\|V\|^2)$ | $O(\|V\| + \|E\|)$ |
| Check if edge $\{u, v\}$ exists | $O(1)$ | $O(\min\{\deg(u), \deg(v)\})$ |
| Find all neighbors of $v$ | $O(\|V\|)$ | $O(\deg(v))$ |
| Add edge | $O(1)$ | $O(1)$ |
| Remove edge $\{u, v\}$ | $O(1)$ | $O(\deg(u) + \deg(v))$ |

**Use Adjacency Matrix when:**

- Dense graphs ($|E| \approx |V|^2$)
- Need fast edge lookups
- Graph is small

**Use Adjacency List when:**

- Sparse graphs ($|E| \ll |V|^2$)
- Need to iterate over neighbors
- Graph is large

**Most real-world graphs are sparse**, so adjacency lists are typically preferred!

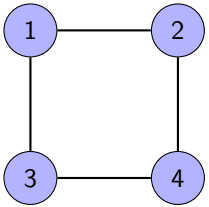
# Properties of Graphs

# The Handshake Lemma

**Theorem 1.2 (Handshake Lemma)**

For every graph $G = (V, E)$:

$$\sum_{v \in V} \deg(v) = 2|E|$$




**Degrees:**

- $\deg(1) = 2$
- $\deg(2) = 2$
- $\deg(3) = 2$
- $\deg(4) = 2$

$\sum \deg(v) = 2 + 2 + 2 + 2 = 8$
$|E| = 4$, so $2|E| = 8$ ✓

# Proof of the Handshake Lemma

**Proof.**

**Intuition:** Each edge contributes 2 to the sum of degrees (once for each endpoint).

- Consider the sum $\sum_{v \in V} \deg(v)$
- This counts how many times vertices are incident to edges
- Each edge $e = \{u, v\}$ has exactly 2 endpoints: $u$ and $v$
- So edge $e$ is counted exactly twice in the sum:
  - Once when counting $\deg(u)$
  - Once when counting $\deg(v)$
- Since there are $|E|$ edges, and each is counted twice:

$$\sum_{v \in V} \deg(v) = 2|E|$$

# A Surprising Consequence

**Corollary 1.3**

In every graph $G = (V, E)$, the **number of vertices with odd degree is even**.

**Let's prove this together!**

- Let $V_{\text{even}}$ = vertices with even degree
- Let $V_{\text{odd}}$ = vertices with odd degree
- We know: $V = V_{\text{even}} \cup V_{\text{odd}}$ (disjoint union)

**Your task:** Use the Handshake Lemma to prove $|V_{\text{odd}}|$ is even.

# Proof: Number of Odd-Degree Vertices is Even

**Proof:**

*Work through the proof with your neighbor (2-3 minutes)*

*Hint: What can you say about the parity of* $\sum_{v \in V} \deg(v)$*?*

# Eulerian Walks

# Important Concepts: Walks, Paths, and Cycles

**Definitions**

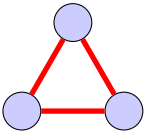
Let $G = (V, E)$ be a graph. A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is:

- A **walk** if $\{v_i, v_{i+1}\} \in E$ for all $0 \leq i \leq k-1$
  - The **length** is $k$ (number of edges)
  - Vertices can be repeated
- A **closed walk** if it is a walk, $k \geq 2$, and $v_0 = v_k$
  - Starts and ends at the same vertex
- A **path** if it is a walk and all vertices are distinct ($v_i \neq v_j$ for $i \neq j$)
  - No vertex is visited twice
- A **cycle** if it is a closed walk, $k \geq 3$, and all vertices except $v_0 = v_k$ are distinct
  - A closed path (returns to start)

# Eulerian Walks and Hamiltonian Paths

**Eulerian Concepts (about edges):**

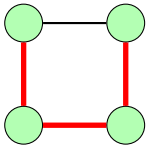
- An **Eulerian walk** is a walk that contains every edge exactly once
- A **closed Eulerian walk** is a closed walk that contains every edge exactly once



Traverses all edges

**Hamiltonian Concepts (about vertices):**

- A **Hamiltonian path** is a path that contains every vertex
- A **Hamiltonian cycle** is a cycle that contains every vertex



Visits all vertices

**Key Distinction:** Eulerian = *edges*, Hamiltonian = *vertices*

# Euler's Answer: When Does an Eulerian Tour Exist?

**Theorem 1.31 (Euler's Theorem)**

A **connected** graph $G$ has an Eulerian tour if and only if **every vertex has even degree**.

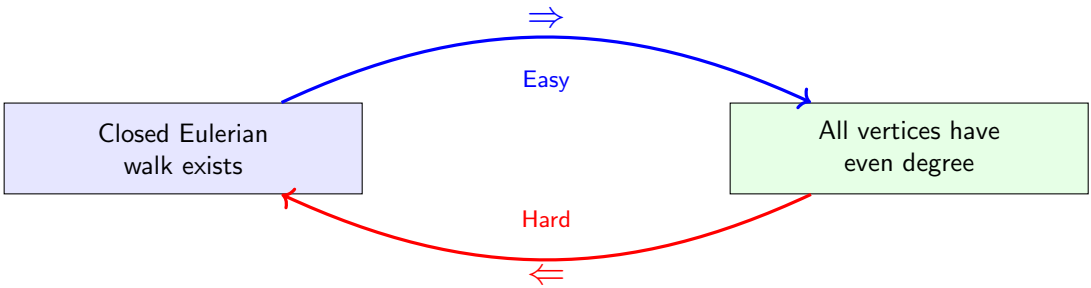
**Why does this make sense?**

- Every time we *enter* a vertex, we must also *leave* it
- This uses up 2 edges at that vertex
- If we traverse all edges exactly once, we must enter/leave each vertex the same number of times

$\Longrightarrow$ Each vertex must have even degree!

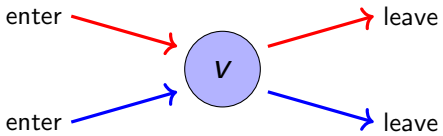
# Proof Strategy

We prove both directions:



# Proof: Direction $\Rightarrow$ (Easy)

**Assume:** There exists a closed Eulerian walk $W$
**Show:** All vertices have even degree




Each visit uses 2 edges

Pick any vertex $v$. In walk $W$, every time we pass through $v$, we use 2 edges (one in, one out). This holds even for the start/end vertex.

Therefore: $\deg(v) = 2 \times (\#$ of visits to $v)$ is even. $\square$

# Proof: Direction $\Leftarrow$ (Hard) - Part 1

**Assume:** All vertices have even degree and $G$ is connected
**Show:** A closed Eulerian walk exists

**Construction:** Start at any vertex $v_0$. Walk randomly, never repeating an edge.

**Key Observation**

**Claim:** This walk must eventually return to $v_0$.

**Proof:** Suppose the walk gets stuck at vertex $u \neq v_0$. During the walk, we visited $u$ some number of times (say $k$ times) before getting stuck. Each visit used 2 edges. Plus the final arrival used 1 edge.

Total edges used at $u$: $2k + 1$ (odd number). But $\deg(u)$ is even! So there must be unused edges at $u$.

Contradiction! We're not stuck. Therefore $u = v_0$.

So we have built a closed walk $W$ starting and ending at $v_0$.

# Proof: Direction $\Leftarrow$ (Hard) - Part 2

We have a closed walk $W$. Does it use all edges?

**Case 1:** Yes, $W$ uses all edges. Then we're done! $W$ is our closed Eulerian walk. ✓

**Case 2:** No, some edges remain unused.

**Notice**

After deleting edges of $W$, the remaining graph may be disconnected. But each connected component still has all vertices with even degree (we removed 2 edges from each vertex on $W$). So we can apply the same procedure to each component independently!

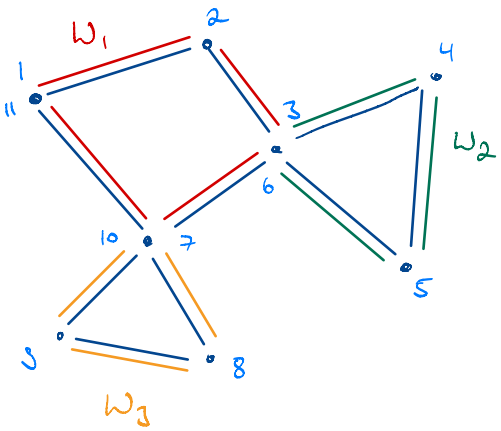
**Extension Step:** Since $G$ is connected, there exists a vertex $u$ on walk $W$ that has unused edges. Starting from $u$, build a new closed walk $W'$ (using the same method as before).

**Merge the walks:** Insert $W'$ into $W$ at vertex $u$:

Follow $W$ until $u$ $\xrightarrow{\text{detour}}$ Follow $W'$ $\xrightarrow{\text{resume}}$ Continue $W$

Repeat this process. Since $G$ has finitely many edges, we eventually use them all. □

# Visual Example: Building a Closed Eulerian Walk



# What About Eulerian Walks (Not Tours)?

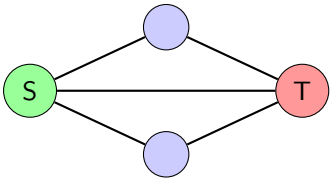
**Relaxation:** What if we don't need to return to the start?

**Theorem (Eulerian Walk)**

A connected graph has an Eulerian **walk** (but not a tour) if and only if it has **exactly 2 vertices of odd degree**.

**Intuition:**

- The walk must *start* at one odd-degree vertex
- It must *end* at the other odd-degree vertex
- All other vertices must have even degree (enter = leave)




Degrees: S=3, A=2, B=2, T=3

Walk: S → A → T → B → S → T

(Starts at S, ends at T)

Questions?

# Additional Practice

# Sufficient Condition for Cycles

**Lemma:** If a graph G has $\deg(v) \geq 2$ for every vertex v, then G must contain a cycle.

# CodeExpert - Determine if a Graph is Eulerian

Like the last two weeks, you can find a CodeExpert template on my website. **Copy** the contents of the **main file** into the main file of the **"Welcome"** exercise. Also copy the **custom.in** and **custom.out** test-cases into the corresponding files in "Welcome".

You are given a graph in the form of an **adjacency list**, where **each vertex** is represented by a **natural number** $0 \leq v \leq n-1$, $n = |V|$, and the **neighbors** of $v$ can be **accessed through** $E$**.get(**$v$**)** (returns a list of the neighboring vertices).

**Implement** the eulerian() method such that it determines if the given graph is Eulerian or not.

**Note,** one condition requires you to check the **connectedness**, which you have not looked at in the lecture yet. However, the algorithm to do so is quite intuitive. Give it a shot!

# Peer Grading

# Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 7.1**

Please follow the usual process:

- Grade the assigned group's submission.
- Give constructive feedback and upload your feedback to Moodle by 23.59 tonight, at the latest.
- Contact me if you don't receive a submission to grade.