Combinatorics & Graph Theory Final Project

Class: Combinatorics & Graph Theory

Lecturer: M.Sc. Nguyễn Quản Bá Hồng

Semester: Summer 2025

Student Name: Nguyễn Ngọc Thạch

Student ID: 2201700077

University of Management and Technology Ho Chi Minh City

Ngày 24 tháng 7 năm 2025

1 Project 3: Integer Partition – Đồ Án 3: Phân Hoạch Số Nguyên

Bài toán 1 (Ferrers & Ferrers transpose diagrams – Biểu đồ Ferrers & biểu đồ Ferrers chuyển vị)**.** *Nhập* $n, k \in \mathbb{N}$. *Viết chương trình* C/C++, Python *để in ra* $p_k(n)$ *biểu đồ Ferrers* F *&* *biểu đồ Ferrers chuyển vị* $F^\top$ *cho mỗi phân hoạch* $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_k) \in (\mathbb{N}^\star)^k$ *có định dạng các dấu chấm được biểu diễn bởi dấu* $*$.

Giải thích và công thức:

- **Phân hoạch số nguyên:** Một phân hoạch của n thành k phần là cách viết $n = \lambda_1 + \lambda_2 + \cdots + \lambda_k$ với $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k > 0$ và $\lambda_i \in \mathbb{N}^\star$.
- **Công thức đệ quy:** Gọi $P_k(n)$ là số phân hoạch của n thành k phần, ta có:

$$P_k(n) = \sum_{i=1}^{n-k+1} P_{k-1}(n-i), \quad P_0(0) = 1,\ P_0(n > 0) = 0$$

  - Để phân hoạch n thành k phần, ta chọn phần đầu tiên là i ($i \geq 1$), còn lại là phân hoạch $n-i$ thành $k-1$ phần, mỗi phần không nhỏ hơn i (để đảm bảo phân hoạch không giảm).
  - Duyệt i từ 1 đến $n-k+1$ (vì mỗi phần ít nhất là 1).
  - Trường hợp cơ sở: $P_0(0) = 1$ (chỉ có 1 cách phân hoạch 0 thành 0 phần), $P_0(n > 0) = 0$ (không thể phân hoạch số dương thành 0 phần).
- **Ý tưởng sinh phân hoạch:**
  - Ý tưởng là xây dựng dần phân hoạch từ trái sang phải (hoặc từ trên xuống dưới), mỗi lần chọn một số i (phần tử tiếp theo), đảm bảo $i \leq$ phần tử trước đó (hoặc $i \leq max$ ban đầu là n).
  - Sau khi chọn i, tiếp tục phân hoạch phần còn lại $n-i$ thành $k-1$ phần, mỗi phần không lớn hơn i.
  - Quá trình này được thực hiện đệ quy cho đến khi đủ k phần và tổng đúng bằng n.
  - Cách này đảm bảo không sinh trùng lặp, vì luôn chọn phần tiếp theo không lớn hơn phần trước.
- **Biểu đồ Ferrers:** Với phân hoạch $\lambda = (\lambda_1, \ldots, \lambda_k)$, biểu đồ Ferrers là bảng gồm k dòng, dòng i có λ_i dấu $*$.
- **Biểu đồ Ferrers chuyển vị:** Lấy bảng Ferrers, hoán vị dòng và cột (lấy cột thành dòng), ta được biểu đồ chuyển vị.

Giải thích code:

- **C++/Python:** Hai chương trình đều dùng hàm đệ quy để sinh tất cả phân hoạch của n thành k phần, mỗi phần không nhỏ hơn phần trước (đảm bảo không trùng lặp).
- **Các biến quan trọng:**
  - n, k: số cần phân hoạch và số phần.
  - `current`: vector/list lưu phân hoạch hiện tại đang xây dựng.
  - `result/partitions`: vector/list chứa tất cả các phân hoạch hợp lệ.
  - `max_val`: giá trị lớn nhất có thể chọn cho phần tiếp theo (đảm bảo không tăng).
- **In biểu đồ Ferrers:** Với mỗi phân hoạch, in ra từng dòng số lượng $*$ tương ứng.
- **In Ferrers chuyển vị:** Duyệt từng dòng (theo số cột lớn nhất), với mỗi phần kiểm tra nếu còn $*$ thì in, ngược lại in khoảng trắng.

Bài toán 2. *Nhập* $n, k \in \mathbb{N}$. *Đếm số phân hoạch của* $n \in \mathbb{N}$. *Viết chương trình* C/C++, Python *để đếm số phân hoạch* $p_{\max}(n, k)$ *của* n *sao cho phần tử lớn nhất là* k. *So sánh* $p_k(n)$ *&* $p_{\max}(n, k)$.

Giải thích và công thức:

- **Phân hoạch n thành k phần ($p_k(n)$):** Là số cách viết $n = \lambda_1 + \cdots + \lambda_k$ với $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k > 0$.
- **Phân hoạch n có phần tử lớn nhất là k ($p_{\max}(n, k)$):** Là số phân hoạch của n mà phần tử lớn nhất đúng bằng k.
- **Công thức đệ quy:**

- $p_k(n) = \sum_{i=1}^{n-k+1} p_{k-1}(n-i)$ với điều kiện phần tử tiếp theo $\leq$ phần trước.
- $p_{\max}(n,k)$ = số phân hoạch của n mà phần tử lớn nhất là k (có thể sinh bằng đệ quy, mỗi nhánh không vượt quá k và phải có ít nhất một phần tử bằng k).

- **Ý tưởng sinh phân hoạch:**
  - Với $p_k(n)$: Dùng đệ quy, mỗi lần chọn một số i ($1 \leq i \leq$ phần trước), tiếp tục phân hoạch $n-i$ thành $k-1$ phần, mỗi phần $\leq i$.
  - Với $p_{\max}(n,k)$: Dùng đệ quy với các bước sau:
    1. **Điều kiện biên:** Nếu $n = 0$ và danh sách hiện tại không rỗng, kiểm tra xem k có xuất hiện trong phân hoạch không. Nếu có, thêm vào kết quả.
    2. **Giới hạn giá trị:** Tại mỗi bước, chọn số i sao cho:
       * $1 \leq i \leq \min(\text{phần trước}, k)$ (đảm bảo không tăng và không vượt quá k)
       * $i \leq n$ (đảm bảo không vượt quá số còn lại)
    3. **Đệ quy:** Thêm i vào phân hoạch hiện tại, gọi đệ quy với $n-i$, sau đó backtrack.
    4. **Điều kiện phần tử lớn nhất:** Chỉ chấp nhận phân hoạch khi max(phân hoạch) $= k$, tức là:
       * k phải xuất hiện ít nhất một lần trong phân hoạch
       * Không có phần tử nào lớn hơn k

Giải thích code:

- **C++/Python:** Hai chương trình đều dùng đệ quy để sinh các phân hoạch theo hai tiêu chí trên.
- **Các biến quan trọng:**
  - n, k: số cần phân hoạch và số phần/phần tử lớn nhất.
  - `current`: vector/list lưu phân hoạch hiện tại.
  - `result`: vector/list chứa các phân hoạch hợp lệ.
  - `max_val`: giá trị lớn nhất có thể chọn cho phần tiếp theo (đảm bảo không tăng).
- **So sánh:** $p_k(n)$ và $p_{\max}(n,k)$

$$p_k(n) = p_{\max}(n,k)$$

Chứng minh. Ta chứng minh bằng phép biến đổi transpose trên sơ đồ Ferrers.

Ý tưởng: Mỗi phân hoạch được biểu diễn bởi sơ đồ Ferrers (dùng dấu *). Phép transpose là "lật"sơ đồ qua đường chéo chính.

Bước 1: Cho phân hoạch n thành k phần: $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$.

Bước 2: Biểu diễn bằng sơ đồ Ferrers và thực hiện phép transpose.

Ví dụ: $10 = 5 + 3 + 2$ (phân hoạch thành 3 phần)

```
Sơ đồ gốc:          Sơ đồ transpose:
* * * * *          * * *
* * *              * * *
* *                * *
                   *
                   *
```

Đọc theo hàng: $10 = 3 + 3 + 2 + 1 + 1$ (phần tử lớn nhất là 3)

Bước 3: Tổng quát hóa:

- Nếu λ có k hàng, thì λ' có cột đầu tiên cao k đơn vị
- Do đó λ' có phần tử lớn nhất là k
- Phép transpose là song ánh: $(\lambda')' = \lambda$

□

Ví dụ: $n = 5, k = 2$

- $p_2(5)$: các phân hoạch là $(4, 1), (3, 2)$.
- $p_{\max}(5, 2)$: các phân hoạch là $(2, 2, 1), (2, 1, 1, 1)$.

Bài toán 3 (Số phân hoạch tự liên hợp)**.** *Nhập $n, k \in \mathbb{N}$. (a) Đếm số phân hoạch tự liên hợp của n có k phần, ký hiệu $p_k^{\text{selfcjg}}(n)$, rồi in ra các phân hoạch đó. (b) Đếm số phân hoạch của n có lẻ phần, rồi so sánh với $p_k^{\text{selfcjg}}(n)$. (c) Thiết lập công thức truy hồi cho $p_k^{\text{selfcjg}}(n)$, rồi implementation bằng: (i) đệ quy. (ii) quy hoạch động.*

Giải thích và công thức:

- **Phân hoạch tự liên hợp (self-conjugate partition):**

  Một phân hoạch của số tự nhiên n là một cách viết n dưới dạng tổng của các số nguyên dương. Ví dụ, $(3, 1, 1)$ là một phân hoạch của 5.

  Để trực quan hóa một phân hoạch $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_k)$, ta dùng **sơ đồ Ferrers**. Sơ đồ này gồm k hàng, hàng thứ i có λ_i dấu sao.

  Ví dụ: Phân hoạch $(4, 2, 1)$ của $n = 7$ có sơ đồ Ferrers:

  ```
  ****
  **
  *
  ```

  Phân hoạch liên hợp (conjugate partition), ký hiệu $\boldsymbol{\lambda}'$, được tạo ra bằng cách "lật"sơ đồ Ferrers qua đường chéo chính (chuyển hàng thành cột và ngược lại).

  Với $\boldsymbol{\lambda} = (4, 2, 1)$, phân hoạch liên hợp $\boldsymbol{\lambda}' = (3, 2, 1, 1)$ có sơ đồ:

  ```
  ***
  **
  *
  *
  ```

  Một phân hoạch được gọi là **tự liên hợp** nếu nó bằng với phân hoạch liên hợp của chính nó, tức là $\boldsymbol{\lambda} = \boldsymbol{\lambda}'$. Điều này có nghĩa là sơ đồ Ferrers của nó đối xứng qua đường chéo chính. **Đường chéo chính** là các ô (i, i) trong sơ đồ, tức là ô thứ i của hàng thứ i.

- **Đặc trưng:**

  Một đặc trưng của phân hoạch tự liên hợp là sự tương ứng một-một với **phân hoạch của n thành các phần lẻ và phân biệt**.

  Ta có thể "bóc"sơ đồ Ferrers của một phân hoạch tự liên hợp thành các "khối hình chữ L"lồng vào nhau, được gọi là các **gnomon** hoặc **hook**. Mỗi hook bao gồm ô trên đường chéo chính, các ô bên phải nó (cánh tay) và các ô bên dưới nó (cái chân). Do tính đối xứng, số ô trên cánh tay bằng số ô trên cái chân.

  Gọi d_i là độ dài cạnh của hook thứ i. Khi đó, hook thứ i sẽ có d_i ô hàng ngang và $d_i - 1$ ô dọc. Tổng số ô của hook là $2d_i - 1$, luôn là số lẻ.

  Ví dụ: Phân hoạch tự liên hợp $\boldsymbol{\lambda} = (5, 4, 3, 2, 1)$ của $n = 15$:

  ```
  *****
  ****
  ***
  **
  *
  ```

  - **Hook ngoài cùng:** gồm hàng 1 và cột 1. Có $d_1 = 5$, tổng số ô là $2d_1 - 1 = 9$.
  - Bóc hook này ra, còn lại phân hoạch tự liên hợp $(3, 2, 1)$ của $n = 15 - 9 = 6$:

```
***
**
*
```

- **Hook thứ hai:** $d_2 = 3 \Rightarrow$ số ô là $2 \cdot 3 - 1 = 5$
- Còn lại phân hoạch (1) của $n = 6 - 5 = 1$
- **Hook cuối cùng:** $d_3 = 1$, số ô là 1

Như vậy, phân hoạch tự liên hợp $(5, 4, 3, 2, 1)$ tương ứng với phân hoạch $15 = 9 + 5 + 1$ gồm các phần lẻ phân biệt.

Phép tương ứng này là một song ánh (bijection). Do đó, **số phân hoạch tự liên hợp của n bằng số phân hoạch của n thành các phần lẻ và phân biệt**.

- **Công thức truy hồi cho $p_k^{\text{selfcjg}}(n)$:**

  Như đã thiết lập, số phân hoạch tự liên hợp của n có k hook trên đường chéo chính, ký hiệu $p_k^{\text{selfcjg}}(n)$, bằng số phân hoạch của n thành k phần lẻ và phân biệt. Ta sẽ xây dựng công thức truy hồi dựa trên tính chất này.

  Gọi một phân hoạch của n thành k phần lẻ, phân biệt là $(h_1, h_2, \ldots, h_k)$ với $h_1 > h_2 > \cdots > h_k \geq 1$. Ta xét hai trường hợp cho phần nhỏ nhất h_k:

  1. **Trường hợp 1: Phần nhỏ nhất bằng 1 ($h_k = 1$).** Nếu ta bỏ phần này đi, ta còn lại $k - 1$ phần lẻ, phân biệt $(h_1, \ldots, h_{k-1})$ có tổng là $n - 1$. Các phần này đều lớn hơn 1. Đây chính là một phân hoạch của $n - 1$ thành $k - 1$ phần lẻ, phân biệt và lớn hơn 1.

  2. **Trường hợp 2: Phần nhỏ nhất lớn hơn 1 ($h_k > 1$).** Vì tất cả các phần h_i đều là số lẻ, nên $h_i \geq 3$. Ta có thể tạo ra một phân hoạch mới bằng cách trừ 2 từ mỗi phần: $(h_1 - 2, h_2 - 2, \ldots, h_k - 2)$. Các phần mới này vẫn là số lẻ, phân biệt, và có phần nhỏ nhất ≥ 1. Tổng của chúng là $(h_1 + \cdots + h_k) - 2k = n - 2k$. Đây là một phân hoạch của $n - 2k$ thành k phần lẻ, phân biệt.

  Hai trường hợp này rời nhau và bao quát tất cả các khả năng. Do đó, ta có công thức truy hồi:

$$p_k^{\text{selfcjg}}(n) = p_k^{\text{selfcjg}}(n - 2k) + p_{k-1}^{\text{selfcjg}}(n - 1).$$

  Điều kiện cơ sở:

  - $p_0^{\text{selfcjg}}(0) = 1$ (phân hoạch rỗng).
  - $p_k^{\text{selfcjg}}(n) = 0$ nếu $n < k^2$ (tổng của k số lẻ phân biệt nhỏ nhất là $1 + 3 + \cdots + (2k - 1) = k^2$).
  - $p_k^{\text{selfcjg}}(n) = 0$ nếu $n < 0$ hoặc $k < 0$.

- **Ý tưởng sinh phân hoạch tự liên hợp:**

  Để sinh một phân hoạch tự liên hợp của n có k hook (ứng với k ô trên đường chéo chính), ta làm như sau:

  1. Tìm tất cả các phân hoạch của n thành đúng k số lẻ và phân biệt. Gọi mỗi phân hoạch là $(h_1, h_2, \ldots, h_k)$ với $h_1 > h_2 > \cdots > h_k > 0$ và $\sum h_i = n$.
  2. Với mỗi h_i, tính $d_i = (h_i + 1)/2$. Khi đó $d_1 > d_2 > \cdots > d_k > 0$.
  3. Từ dãy $(d_1, \ldots, d_k)$, dựng lại phân hoạch tự liên hợp bằng cách lồng các hook vào nhau.

  Ví dụ: Tìm các phân hoạch tự liên hợp của $n = 9$.

  - Phân hoạch $9 = (9) \Rightarrow k = 1$ hook, $d_1 = (9 + 1)/2 = 5$
  - Dựng lại phân hoạch: $(5, 1, 1, 1, 1)$

```
*****
*
*
*
*
```

  - Phân hoạch $9 = (5, 3, 1) \Rightarrow k = 3$ hook, $d = (3, 2, 1)$
  - Dựng lại phân hoạch: $(3, 3, 3)$

```
***
***
***
```

Vậy $n = 9$ có 2 phân hoạch tự liên hợp là: $(5, 1, 1, 1, 1)$ và $(3, 3, 3)$.

(i) Phương pháp đệ quy (Backtracking):

- Duyệt tất cả các phân hoạch của n thành k số lẻ phân biệt giảm dần.
- Ở mỗi bước, thử thêm một số lẻ h vào phân hoạch hiện tại sao cho:
  - $h \leq$ số trước đó (để giảm dần),
  - h là số lẻ chưa dùng,
  - tổng không vượt quá n,
  - số phần không vượt quá k.
- Nếu tổng đạt n và đủ k phần: chuyển thành phân hoạch tự liên hợp và lưu.
- Cắt nhánh sớm nếu: tổng vượt n, số phần vượt k, hoặc tổng không thể đạt được do còn lại $< k^2$.

(ii) Phương pháp quy hoạch động (Dynamic Programming):

- Dùng bảng $dp[i][j]$ là danh sách các phân hoạch của i thành j số lẻ phân biệt.
- Khởi tạo: $dp[0][0] = \{[]\}$.
- Với mỗi i từ 1 đến n, mỗi j từ 1 đến k:
  - Nguồn 1: từ $dp[i-2j][j]$, cộng 2 vào mỗi phần tử trong phân hoạch.
    Giả sử ta đã có một phân hoạch $P = (a_1, a_2, \ldots, a_j)$ của $i - 2j$ thành j số lẻ phân biệt. Nếu ta cộng thêm 2 vào mỗi phần tử, ta được phân hoạch mới $P' = (a_1 + 2, a_2 + 2, \ldots, a_j + 2)$ có tổng là i.
    Vì các a_i là số lẻ phân biệt, nên $a_i + 2$ vẫn là số lẻ và phân biệt. Do đó, P' là một phân hoạch của i thành j số lẻ phân biệt. Ta thu được $dp[i][j]$ từ đây.
  - Nguồn 2: từ $dp[i-1][j-1]$, thêm phần tử 1 nếu các phần tử trong phân hoạch đều > 1.
    Giả sử ta có một phân hoạch $Q = (a_1, a_2, \ldots, a_{j-1})$ của $i - 1$ thành $j - 1$ số lẻ phân biệt. Nếu tất cả $a_i > 1$, ta có thể thêm phần tử 1 vào để tạo thành phân hoạch $Q' = (a_1, \ldots, a_{j-1}, 1)$.
    Khi đó, Q' là phân hoạch của i thành j phần lẻ phân biệt (do 1 nhỏ hơn tất cả các phần còn lại, và vẫn giữ phân biệt). Sắp xếp lại theo thứ tự giảm dần nếu cần.

  Hai nguồn được chia ra là vì chúng phản ánh hai cách xây dựng hoàn toàn khác nhau để tạo ra phân hoạch của i thành j số lẻ phân biệt:

  - **Nguồn 1 ($dp[i-2j][j]$):** Dựa trên việc tăng *đồng đều* mỗi phần tử của một phân hoạch có j phần tử lên thêm 2. Đây là cách **bảo toàn số lượng phần tử**, và chỉ làm tổng tăng thêm đúng $2j$. Nhờ đó, ta đảm bảo các phần tử vẫn lẻ, vẫn phân biệt, và tổng thành i.
  - **Nguồn 2 ($dp[i-1][j-1]$):** Dựa trên việc **tăng số lượng phần tử thêm 1** bằng cách thêm phần tử 1 vào một phân hoạch có $j - 1$ phần tử. Tuy nhiên, để giữ phân biệt, ta chỉ được phép thêm 1 khi phân hoạch cũ không chứa 1, tức là tất cả các phần tử cũ phải > 1.

  Việc chia 2 nguồn như vậy giúp bao phủ **tất cả các trường hợp có thể** để tạo phân hoạch của i thành j số lẻ phân biệt mà không bị trùng và không bỏ sót.
- Với mỗi phân hoạch H trong $dp[n][k]$, chuyển sang phân hoạch tự liên hợp như trên.

- **Ý tưởng đếm số phân hoạch của n có số phần tử là lẻ:**

  Đây là bài toán đếm các phân hoạch $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_k)$ của n sao cho độ dài (số phần tử) k là một số lẻ.

  Phương pháp tiếp cận trực tiếp (Dùng Quy hoạch động):

  Cách tiếp cận tự nhiên và hiệu quả nhất để giải quyết bài toán này là trước hết tính số phân hoạch của n thành đúng k phần, ký hiệu là $p(n, k)$, cho mọi k có thể. Sau đó, ta chỉ cần lấy tổng của các giá trị $p(n, k)$ với k là số lẻ.

  1. **Tính $p(n, k)$:** Ta sử dụng công thức truy hồi kinh điển sau:

  $$p(n, k) = p(n - 1, k - 1) + p(n - k, k)$$

  Giải thích công thức: Một phân hoạch của n thành k phần có thể thuộc một trong hai loại sau:

- **Loại 1: Có chứa ít nhất một phần tử bằng 1.** Nếu ta bỏ đi một phần tử '1', ta sẽ còn lại một phân hoạch của số $n-1$ thành $k-1$ phần. Số cách làm như vậy là $p(n-1,k-1)$.
- **Loại 2: Tất cả các phần tử đều lớn hơn hoặc bằng 2.** Nếu ta trừ 1 từ mỗi phần tử trong k phần này, ta sẽ nhận được một phân hoạch mới của số $n-k$ thành k phần. Số cách làm như vậy là $p(n-k,k)$.

2. **Xây dựng bảng giá trị:** Ta có thể dùng quy hoạch động để xây dựng một bảng 2 chiều (ví dụ, 'dp[i][j]') lưu giá trị $p(i,j)$ cho tất cả $1 \le i \le n$ và $1 \le j \le i$.

Điều kiện cơ sở:

- $p(k,k) = 1$ (phân hoạch k thành k phần chỉ có một cách: $1+1+\cdots+1$).
- $p(n,1) = 1$ (phân hoạch n thành 1 phần chỉ có một cách: chính là n).
- $p(n,k) = 0$ nếu $k > n$.
- $p(n,k) = 0$ nếu $n \le 0$ hoặc $k \le 0$, trừ $p(0,0) = 1$.

3. **Tính tổng cuối cùng:** Sau khi bảng 'dp[n][k]' đã được điền đầy đủ, số phân hoạch của n có số phần tử là lẻ được tính bằng tổng:

$$S = \sum_{j=0}^{\lfloor (n-1)/2 \rfloor} p(n, 2j+1) = p(n,1) + p(n,3) + p(n,5) + \dots$$

- **So sánh với phân hoạch có lẻ phần:**

  Cần phân biệt rõ số phân hoạch của n có lẻ phần, và số phân hoạch của n có lẻ phần phân biệt.

  Ví dụ: Xét $n = 5$:

  - Phân hoạch tự liên hợp: chỉ có $(3,1,1)$, tương ứng với hook có $h = (5) \Rightarrow$ có **1** phân hoạch.
  - Phân hoạch của 5 thành các phần lẻ:
    1. (5)
    2. $(3,1,1)$
    3. $(1,1,1,1,1)$

    Tổng cộng **3** phân hoạch.

  Rõ ràng: $1 \neq 3$. Vì vậy, **số phân hoạch tự liên hợp** của n không bằng số phân hoạch của n thành các phần lẻ nói chung, mà bằng số phân hoạch có các phần *vừa lẻ, vừa phân biệt.*

Giải thích code:

- **Các biến quan trọng:**
  - `n, k`: tổng cần phân hoạch và số hook (số phần trên đường chéo chính).
  - `current`, `curr`: vector tạm lưu dãy các số lẻ phân biệt khi sinh đệ quy.
  - `odd_parts`: vector chứa tất cả phân hoạch lẻ phân biệt tìm được.
  - `max_val`: giá trị lẻ lớn nhất còn có thể chọn cho phần tử kế tiếp.
  - `rec_cnt`: kết quả đếm tự liên hợp bằng *đệ quy.*
  - `dp`: bảng DP với `dp[sum][parts]` = số phân hoạch tự liên hợp của `sum` thành `parts` hook.
  - `dp_cnt`: kết quả đếm tự liên hợp bằng *quy hoạch động.*

2 Project 4: Graph & Tree Traversing Problems – Đồ Án 4: Các Bài Toán Duyệt Đồ Thị & Cây

Bài toán 4. *Viết chương trình* C/C++, Python *chuyển đổi giữa 4 dạng biểu diễn: adjacency matrix, adjacency list, extended adjacency list, adjacency map cho 3 đồ thị: đơn đồ thị, đa đồ thị, đồ thị tổng quát; & 3 dạng biểu diễn: array of parents, first-child next-sibling, graph-based representation of trees của cây.*

Sẽ có $3A_4^3 + A_3^2 = 36 + 6 = 42$ converter programs.

(a) Giải thích thuật toán chuyển đổi biểu diễn đồ thị đơn:

- **Mục tiêu:** Chuyển đổi giữa 4 dạng biểu diễn đồ thị đơn (undirected simple graph): adjacency matrix, adjacency list, extended adjacency list, adjacency map.
- **Đồ thị đơn:** Đồ thị không có cạnh song song và không có khuyên (self-loop), mỗi cặp đỉnh có tối đa 1 cạnh nối.
- **4 dạng biểu diễn:**
  - **Adjacency Matrix (AM):** Ma trận $n \times n$ với $A[i][j] = 1$ nếu có cạnh (i, j), $A[i][j] = 0$ nếu không có cạnh.
  - **Adjacency List (AL):** Mảng n danh sách, mỗi danh sách chứa các đỉnh kề với đỉnh tương ứng.
  - **Extended Adjacency List (EAL):** Gồm danh sách các cạnh, danh sách incoming edges và outgoing edges cho mỗi đỉnh.
  - **Adjacency Map (AMap):** Dictionary với key là đỉnh, value là dictionary chứa các cạnh incoming và outgoing.

Công thức chuyển đổi:

- **Adjacency List ↔ Adjacency Matrix:**
  - AL → AM: Với mỗi đỉnh i và mỗi đỉnh kề j trong danh sách kề của i: $A[i][j] = 1$
  - AM → AL: Với mỗi cặp (i, j) mà $A[i][j] = 1$: thêm j vào danh sách kề của i
- **Adjacency List ↔ Extended Adjacency List:**
  - AL → EAL: Thu thập tất cả các cạnh (u, v) với $u \leq v$, xây dựng danh sách edges, incoming và outgoing
  - EAL → AL: Với mỗi cạnh (u, v) trong danh sách edges: thêm v vào danh sách kề của u và ngược lại
- **Adjacency List ↔ Adjacency Map:**
  - AL → AMap: Với mỗi cạnh (u, v) với $u \leq v$: thêm mapping vào outgoing và incoming dictionaries
  - AMap → AL: Duyệt qua tất cả các cạnh trong outgoing dictionaries và thêm vào danh sách kề
- **Adjacency Matrix ↔ Extended Adjacency List:**
  - AM → EAL: Thu thập tất cả các cạnh (u, v) với $u \leq v$ từ ma trận, xây dựng danh sách edges, incoming và outgoing
  - EAL → AM: Với mỗi cạnh (u, v) trong danh sách edges: $A[u][v] = A[v][u] = 1$
- **Adjacency Matrix ↔ Adjacency Map:**
  - AM → AMap: Với mỗi cạnh (u, v) với $u \leq v$: thêm mapping vào outgoing và incoming dictionaries
  - AMap → AM: Duyệt qua tất cả các cạnh trong outgoing dictionaries và đặt $A[u][v] = A[v][u] = 1$
- **Extended Adjacency List ↔ Adjacency Map:**
  - EAL → AMap: Với mỗi cạnh (u, v) trong danh sách edges: thêm mapping vào outgoing và incoming dictionaries
  - AMap → EAL: Thu thập các cạnh duy nhất từ outgoing dictionaries, xây dựng danh sách edges, incoming và outgoing

Giải thích code:

- **Biến số quan trọng:**
  - n: số lượng đỉnh của đồ thị
  - m: số lượng cạnh của đồ thị
  - `adj_list`: danh sách kề, `adj_list[i]` chứa các đỉnh kề với đỉnh i
  - `matrix`: ma trận kề $n \times n$, `matrix[i][j]` = true nếu có cạnh (i, j)
  - `ext_list`: extended adjacency list với danh sách incoming/outgoing edges
  - `adj_map`: adjacency map với dictionary cho incoming/outgoing edges

- **Đọc input:** Dòng đầu: n m, sau đó m dòng mỗi dòng chứa 2 số u v biểu diễn cạnh (u, v)
- **Menu tương tác:** Cho phép người dùng chọn loại chuyển đổi và hiển thị kết quả

Ý nghĩa các biến chính:

- `AdjacencyMatrix.matrix[i][j]`: Biểu diễn có cạnh từ đỉnh i đến đỉnh j hay không
- `AdjacencyList.adj[i]`: Danh sách các đỉnh kề với đỉnh i
- `ExtendedAdjacencyList.edges`: Danh sách tất cả các cạnh của đồ thị
- `ExtendedAdjacencyList.outgoing[i]`: Chỉ số các cạnh đi ra từ đỉnh i
- `ExtendedAdjacencyList.incoming[i]`: Chỉ số các cạnh đi vào đỉnh i
- `AdjacencyMap.outgoing[i][j]`: Cạnh (i, j) đi ra từ đỉnh i
- `AdjacencyMap.incoming[i][j]`: Cạnh (j, i) đi vào đỉnh i

(b) Giải thích thuật toán chuyển đổi biểu diễn đa đồ thị (multigraph):

- **Mục tiêu:** Chuyển đổi giữa 4 dạng biểu diễn đồ thị đa bộ (multigraph) không có khuyên (no self-loops): ma trận kề (Adjacency Matrix - AM), danh sách kề (Adjacency List - AL), danh sách kề mở rộng (Extended Adjacency List - EAL), và bản đồ kề (Adjacency Map - AMap).
- **Đa đồ thị (Multigraph - không có khuyên):** Là đồ thị cho phép có nhiều hơn một cạnh nối giữa cùng một cặp đỉnh (cạnh song song), nhưng không cho phép cạnh nối một đỉnh với chính nó (khuyên - self-loop).
- **4 dạng biểu diễn:**
  - **Adjacency Matrix (AM):** Ma trận $n \times n$ với $A[i][j]$ biểu thị **số lượng cạnh** nối từ đỉnh i đến đỉnh j. Vì đồ thị vô hướng, $A[i][j] = A[j][i]$. $A[i][i]$ luôn là 0 vì không có khuyên.
  - **Adjacency List (AL):** Mảng n danh sách. Mỗi danh sách $AL[i]$ chứa tất cả các đỉnh kề với đỉnh i. Nếu có nhiều cạnh giữa i và j, j sẽ xuất hiện nhiều lần trong $AL[i]$ (và i sẽ xuất hiện nhiều lần trong $AL[j]$).
  - **Extended Adjacency List (EAL):**
    - `edges`: Một danh sách chứa tất cả các cạnh riêng lẻ dưới dạng cặp (u, v). Nếu có nhiều cạnh giữa u và v, cặp (u, v) sẽ xuất hiện nhiều lần trong danh sách này.
    - `incoming[v]`: Danh sách các chỉ số của các cạnh trong `edges` mà đi vào đỉnh v.
    - `outgoing[u]`: Danh sách các chỉ số của các cạnh trong `edges` mà đi ra từ đỉnh u.
    - m: Tổng số lượng cạnh (bao gồm các cạnh song song).
  - **Adjacency Map (AMap):**
    - `outgoing[u]`: Một danh sách các tuple. Mỗi tuple là $(v, (canonical\_u, canonical\_v))$, trong đó v là đỉnh kề và $(canonical\_u, canonical\_v)$ là dạng chính tắc của cạnh (min(u,v), max(u,v)). Nếu có nhiều cạnh giữa u và v, tuple $(v, (min(u, v), max(u, v)))$ sẽ xuất hiện nhiều lần trong danh sách này.
    - `incoming[v]`: Tương tự như `outgoing`, nhưng cho các cạnh đi vào.
    - m: Tổng số lượng cạnh (bao gồm các cạnh song song).

Công thức chuyển đổi cho đa đồ thị (không có khuyên):

- **Adjacency List ↔ Adjacency Matrix:**
  - AL → AM: Với mỗi đỉnh i và mỗi đỉnh kề j trong $AL[i]$: tăng $A[i][j]$ lên 1.
  - AM → AL: Với mỗi cặp (i, j) mà $A[i][j] > 0$: thêm j vào $AL[i]$ $A[i][j]$ lần.
- **Adjacency List ↔ Extended Adjacency List:**
  - AL → EAL: Duyệt qua AL. Với mỗi cạnh (u, v) (với $u < v$ để tránh trùng lặp và khuyên), thêm (u, v) vào `ext.edges` và cập nhật `incoming`/`outgoing` cho cả u và v. Số lần xuất hiện của (u, v) trong $AL[u]$ (hoặc $AL[v]$) sẽ xác định số lượng cạnh song song.
  - EAL → AL: Với mỗi cạnh (u, v) trong `ext.edges`: thêm v vào $AL[u]$ và thêm u vào $AL[v]$.

- **Adjacency List ↔ Adjacency Map:**
  - AL → AMap: Duyệt qua AL. Với mỗi cạnh (u, v) (với $u < v$), thêm $(v, (min(u, v), max(u, v)))$ vào `adj_map.outgoing[u]` và $(u, (min(u, v), max(u, v)))$ vào `adj_map.incoming[v]`.
  - AMap → AL: Duyệt qua `adj_map.outgoing`. Với mỗi cặp $(v, \text{canonical\_edge})$ trong `adj_map.outgoing[u]`: thêm v vào $AL[u]$.
- **Adjacency Matrix ↔ Extended Adjacency List:**
  - AM → EAL: Duyệt qua $A[i][j]$. Với mỗi $A[i][j] > 0$ và $i < j$: thêm (i, j) vào `ext.edges` $A[i][j]$ lần và cập nhật `incoming/outgoing` tương ứng.
  - EAL → AM: Với mỗi cạnh (u, v) trong `ext.edges`: tăng $A[u][v]$ và $A[v][u]$ lên 1.
- **Adjacency Matrix ↔ Adjacency Map:**
  - AM → AMap: Duyệt qua $A[i][j]$. Với mỗi $A[i][j] > 0$ và $i < j$: thêm $(j, (min(i, j), max(i, j)))$ vào `adj_map.outgoing[i]` $A[i][j]$ lần và $(i, (min(i, j), max(i, j)))$ vào `adj_map.incoming[j]` $A[i][j]$ lần.
  - AMap → AM: Duyệt qua `adj_map.outgoing`. Với mỗi cặp $(v, \text{canonical\_edge})$ trong `adj_map.outgoing[u]`: tăng $A[u][v]$ lên 1.
- **Extended Adjacency List ↔ Adjacency Map:**
  - EAL → AMap: Với mỗi cạnh (u, v) trong `ext.edges`: thêm $(v, (min(u, v), max(u, v)))$ vào `adj_map.outgoing[u]` và $(u, (min(u, v), max(u, v)))$ vào `adj_map.incoming[v]`.
  - AMap → EAL: Duyệt qua `adj_map.outgoing` để đếm số lần xuất hiện của mỗi cạnh chính tắc. Sau đó, thêm các cạnh vào `ext.edges` và cập nhật `incoming/outgoing` tương ứng.

Giải thích code:

- **Các lớp biểu diễn:** Các lớp `AdjacencyMatrix`, `AdjacencyList`, `ExtendedAdjacencyList`, `AdjacencyMap` được định nghĩa để lưu trữ cấu trúc dữ liệu của đồ thị.
- **Xử lý đa cạnh:**
  - `AdjacencyMatrix.matrix[i][j]`: Lưu trữ số nguyên (integer) để đếm số lượng cạnh giữa i và j.
  - `AdjacencyList.adj[i]`: Một đỉnh j có thể xuất hiện nhiều lần trong danh sách `adj[i]` nếu có nhiều cạnh giữa i và j.
  - `ExtendedAdjacencyList.edges`: Lưu trữ từng thể hiện riêng lẻ của một cạnh. Ví dụ, nếu có hai cạnh giữa 0 và 1, `edges` sẽ chứa `(0,1)` hai lần.
  - `AdjacencyMap.outgoing[u]` và `incoming[v]`: Mỗi danh sách này chứa các tuple `(neighbor, canonical_edge)`. Nếu có nhiều cạnh giữa u và v, tuple `(v, (min(u,v), max(u,v)))` sẽ xuất hiện nhiều lần.
- **Xử lý không có khuyên:**
  - Trong hàm `main`, có kiểm tra `if u == v:` để bỏ qua các cạnh là khuyên.
  - Trong các hàm chuyển đổi như `matrix_to_extended`, `matrix_to_map`, `list_to_extended`, `list_to_map`, và `extended_to_map`, có các điều kiện `if i == j: continue` hoặc `if u == v: continue` để đảm bảo khuyên không được thêm vào hoặc xử lý.
  - Khi tính `actual_edge_count` trong `list_to_extended` và `map_to_extended`, không có trường hợp đặc biệt cho khuyên vì chúng đã được lọc ra.
- **Tính tổng số cạnh (m):** Biến m trong `ExtendedAdjacencyList` và `AdjacencyMap` được cập nhật để phản ánh tổng số lượng thể hiện cạnh.

Ý nghĩa các biến chính:

- `AdjacencyMatrix.matrix[i][j]`: Số lượng cạnh nối đỉnh i và đỉnh j.
- `AdjacencyList.adj[i]`: Danh sách các đỉnh kề với đỉnh i, bao gồm cả sự lặp lại của các đỉnh nếu có đa cạnh.
- `ExtendedAdjacencyList.edges`: Danh sách các cặp (u, v) biểu diễn từng cạnh riêng lẻ của đồ thị.
- `ExtendedAdjacencyList.outgoing[i]`: Danh sách các chỉ số (index) của các cạnh trong `ext.edges` mà bắt đầu từ đỉnh i.

- `ExtendedAdjacencyList.incoming[i]`: Danh sách các chỉ số (index) của các cạnh trong `ext.edges` mà kết thúc tại đỉnh i.
- `AdjacencyMap.outgoing[i]`: Danh sách các tuple `(neighbor, canonical_edge)` cho các cạnh đi ra từ đỉnh i.
- `AdjacencyMap.incoming[i]`: Danh sách các tuple `(neighbor, canonical_edge)` cho các cạnh đi vào đỉnh i.
- `canonical_edge`: Một tuple `(min(u,v), max(u,v))` dùng để định danh duy nhất một cạnh không định hướng, bất kể thứ tự nhập vào.

(c) Giải thích thuật toán chuyển đổi biểu diễn đồ thị tổng quát (General Graph):

- **Mục tiêu:** Chuyển đổi giữa 4 dạng biểu diễn đồ thị tổng quát (general graph) cho phép cả đa cạnh và khuyên: ma trận kề (AM), danh sách kề (AL), danh sách kề mở rộng (EAL), và bản đồ kề (AMap).
- **Đồ thị tổng quát (General Graph):** Là đồ thị cho phép có nhiều hơn một cạnh nối giữa cùng một cặp đỉnh (cạnh song song) và cho phép cạnh nối một đỉnh với chính nó (khuyên - self-loop).
- **4 dạng biểu diễn:** Cấu trúc cơ bản tương tự như đa đồ thị, nhưng có sự khác biệt trong cách xử lý khuyên.
  - **Adjacency Matrix (AM):** $A[i][j]$ vẫn biểu thị số lượng cạnh nối từ i đến j. Điểm khác biệt là $A[i][i]$ có thể lớn hơn 0 nếu có khuyên tại đỉnh i.
  - **Adjacency List (AL):** Tương tự đa đồ thị, nhưng $AL[i]$ có thể chứa i nhiều lần nếu có khuyên tại đỉnh i.
  - **Extended Adjacency List (EAL):** `edges` sẽ chứa các cặp (u, u) cho khuyên. `incoming`/`outgoing` sẽ trỏ đến các chỉ số của khuyên đó.
  - **Adjacency Map (AMap):** `outgoing[u]` và `incoming[v]` sẽ chứa các tuple `(u, (u,u))` cho khuyên tại đỉnh u.

Công thức chuyển đổi cho đồ thị tổng quát:

- **Adjacency List ↔ Adjacency Matrix:**
  - AL → AM: Giống như đa đồ thị. Khuyên (i, i) sẽ làm tăng $A[i][i]$ lên 1.
  - AM → AL: Giống như đa đồ thị. Nếu $A[i][i] > 0$, i sẽ được thêm vào $AL[i]$ $A[i][i]$ lần.
- **Adjacency List ↔ Extended Adjacency List:**
  - AL → EAL: Duyệt qua AL. Với mỗi cạnh (u, v):
    - Nếu $u = v$ (khuyên): thêm (u, u) vào `ext.edges` và cập nhật `incoming`/`outgoing` cho u.
    - Nếu $u \neq v$: thêm $(min(u, v), max(u, v))$ vào `ext.edges` và cập nhật `incoming`/`outgoing` cho cả u và v.
  - EAL → AL: Với mỗi cạnh (u, v) trong `ext.edges`: thêm v vào $AL[u]$. Nếu $u \neq v$, thêm u vào $AL[v]$.
- **Adjacency List ↔ Adjacency Map:**
  - AL → AMap: Duyệt qua AL. Với mỗi cạnh (u, v):
    - Thêm $(v, (min(u, v), max(u, v)))$ vào `adj_map.outgoing[u]`.
    - Thêm $(u, (min(u, v), max(u, v)))$ vào `adj_map.incoming[v]`.
  - AMap → AL: Giống như đa đồ thị. Duyệt qua `adj_map.outgoing`. Với mỗi cặp $(v, \text{canonical\_edge})$ trong `adj_map.outgoing[u]`: thêm v vào $AL[u]$.
- **Adjacency Matrix ↔ Extended Adjacency List:**
  - AM → EAL: Duyệt qua $A[i][j]$. Với mỗi $A[i][j] > 0$:
    - Nếu $i = j$ (khuyên): thêm (i, i) vào `ext.edges` $A[i][i]$ lần và cập nhật `incoming`/`outgoing` cho i.
    - Nếu $i < j$ (cạnh không phải khuyên): thêm (i, j) vào `ext.edges` $A[i][j]$ lần và cập nhật `incoming`/`outgoing` cho cả i và j.

  - EAL → AM: Giống như đa đồ thị. Với mỗi cạnh (u, v) trong `ext.edges`: tăng $A[u][v]$ lên 1. Nếu $u \neq v$, tăng $A[v][u]$ lên 1.

- **Adjacency Matrix ↔ Adjacency Map:**
  - AM → AMap: Duyệt qua $A[i][j]$. Với mỗi $A[i][j] > 0$:
    * Thêm $(j, (min(i, j), max(i, j)))$ vào `adj_map.outgoing[i]` $A[i][j]$ lần.
    * Thêm $(i, (min(i, j), max(i, j)))$ vào `adj_map.incoming[j]` $A[i][j]$ lần.
  - AMap → AM: Giống như đa đồ thị. Duyệt qua `adj_map.outgoing`. Với mỗi cặp $(v, \text{canonical\_edge})$ trong `adj_map.outgoing[u]`: tăng $A[u][v]$ lên 1.

- **Extended Adjacency List ↔ Adjacency Map:**
  - EAL → AMap: Giống như đa đồ thị. Với mỗi cạnh (u, v) trong `ext.edges`: thêm $(v, (min(u, v), max(u, v)))$ vào `adj_map.outgoing[u]` và $(u, (min(u, v), max(u, v)))$ vào `adj_map.incoming[v]`.
  - AMap → EAL: Duyệt qua `adj_map.outgoing` để đếm số lần xuất hiện của mỗi cạnh chính tắc. Sau đó, thêm các cạnh vào `ext.edges` và cập nhật `incoming`/`outgoing` tương ứng.

Giải thích code:

- **Các lớp biểu diễn:** Cấu trúc các lớp Python `AdjacencyMatrix`, `AdjacencyList`, `ExtendedAdjacencyList`, `AdjacencyMap` về cơ bản giống như phiên bản đa đồ thị, nhưng cách chúng được điền dữ liệu và xử lý khuyên sẽ khác.

- **Xử lý đa cạnh và khuyên:**
  - `AdjacencyMatrix.matrix[i][j]`: Vẫn lưu trữ số lượng cạnh. $A[i][i]$ có thể lớn hơn 0.
  - `AdjacencyList.adj[i]`: Đỉnh i có thể xuất hiện nhiều lần trong `adj[i]` nếu có khuyên.
  - `ExtendedAdjacencyList.edges`: Sẽ chứa các cặp (u, u) cho khuyên.
  - `AdjacencyMap.outgoing[u]` và `incoming[v]`: Sẽ chứa các tuple `(u, (u,u))` cho khuyên.

- **Điểm khác biệt chính so với đa đồ thị (không có khuyên):**
  - **Loại bỏ kiểm tra khuyên:** Trong hàm `main` và các hàm chuyển đổi (ví dụ: `matrix_to_extended`, `matrix_to_map`, `list_to_extended`, `list_to_map`, `extended_to_map`), các điều kiện `if u == v: continue` hoặc `if i == j: continue` sẽ **bị loại bỏ** hoặc **thay đổi** để cho phép và xử lý khuyên.
  - **Tính toán `actual_edge_count`:** Trong các hàm như `list_to_extended` và `map_to_extended`, logic để tính `actual_edge_count` sẽ phân biệt rõ ràng giữa khuyên và cạnh thông thường.
    * Đối với khuyên (u, u): `actual_edge_count` sẽ bằng số lần nó xuất hiện trong tổng danh sách kề (hoặc tổng số lần xuất hiện trong map).
    * Đối với cạnh (u, v) không phải khuyên: `actual_edge_count` sẽ bằng một nửa số lần nó xuất hiện trong tổng danh sách kề (hoặc tổng số lần xuất hiện trong map), vì nó được đếm hai lần (một lần từ u, một lần từ v).

- **Tính tổng số cạnh (m):** Biến m vẫn phản ánh tổng số lượng thể hiện cạnh, bao gồm cả khuyên.

Ý nghĩa các biến chính:

- `AdjacencyMatrix.matrix[i][j]`: Số lượng cạnh nối đỉnh i và đỉnh j. Bao gồm cả khuyên ($i = j$).

- `AdjacencyList.adj[i]`: Danh sách các đỉnh kề với đỉnh i, bao gồm cả sự lặp lại của các đỉnh và chính đỉnh i nếu có khuyên.

- `ExtendedAdjacencyList.edges`: Danh sách các cặp (u, v) biểu diễn từng cạnh riêng lẻ của đồ thị, bao gồm cả các khuyên (u, u).

- `ExtendedAdjacencyList.outgoing[i]`: Danh sách các chỉ số (index) của các cạnh trong `ext.edges` mà bắt đầu từ đỉnh i.

- `ExtendedAdjacencyList.incoming[i]`: Danh sách các chỉ số (index) của các cạnh trong `ext.edges` mà kết thúc tại đỉnh i.

- `AdjacencyMap.outgoing[i]`: Danh sách các tuple `(neighbor, canonical_edge)` cho các cạnh đi ra từ đỉnh i. Nếu có khuyên, nó sẽ chứa `(i, (i,i))`.

- `AdjacencyMap.incoming[i]`: Danh sách các tuple `(neighbor, canonical_edge)` cho các cạnh đi vào đỉnh i. Nếu có khuyên, nó sẽ chứa `(i, (i,i))`.
- `canonical_edge`: Một tuple `(min(u,v), max(u,v))` dùng để định danh duy nhất một cạnh không định hướng, bao gồm cả khuyên.

Bài toán 5. *Làm Problems 1.1–1.6 & Exercises 1.1–1.10*

Problem 1.1
Determine the size of the complete graph K_n on n vertices and the complete bipartite graph $K_{p,q}$ on $p+q$ vertices.
Definitions

- A **complete graph** K_n is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
- A **complete bipartite graph** $K_{p,q}$ is a bipartite graph whose vertex set is partitioned into two disjoint sets with p and q vertices respectively, and where every vertex in the first set is connected to every vertex in the second set.

Solution

- The size of a graph refers to the number of edges it contains.
- For the complete graph K_n, each pair of n vertices is connected by an edge. The number of such pairs is given by the binomial coefficient:

$$\text{Size of } K_n = \binom{n}{2} = \frac{n(n-1)}{2}$$

- For the complete bipartite graph $K_{p,q}$, each of the p vertices in the first set is connected to each of the q vertices in the second set. Thus, the number of edges is:

$$\text{Size of } K_{p,q} = p \cdot q$$

Problem 1.2
Determine the values of n for which the circle graph C_n on n vertices is bipartite, and also the values of n for which the complete graph K_n is bipartite.
Definitions

- A graph is called **bipartite** if its vertex set can be divided into two disjoint sets such that no two vertices within the same set are adjacent.
- A **cycle graph** C_n is a graph that consists of a single cycle through n vertices.
- A **complete graph** K_n is a graph in which every pair of distinct vertices is connected by a unique edge.

Solution

- For the cycle graph C_n:
  - A cycle is bipartite if and only if it has an even number of vertices.
  - This is because an odd cycle forces a vertex to connect to another in the same partition, violating the bipartite condition.
  - **Therefore, C_n is bipartite if and only if n is even.**
- For the complete graph K_n:
  - In a bipartite graph, there are no edges between vertices of the same partition.
  - In K_n, every vertex is connected to every other vertex, so the only way for K_n to be bipartite is if no two vertices within the same set are connected.
  - This is only possible when $n = 2$.
  - **Therefore, K_n is bipartite if and only if $n = 2$.**

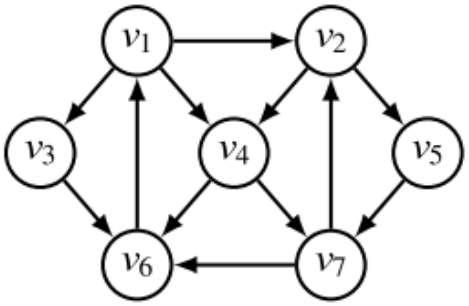
Problem 1.3

Give all the spanning trees of the graph in Fig. 1.30, and also the number of spanning trees of the underlying undirected graph.

Definitions

- A **spanning tree** of a graph is a subgraph that:
  - includes all the vertices of the original graph,
  - is a tree (i.e., connected and acyclic),
  - and has exactly $n-1$ edges if the graph has n vertices.
- The **underlying undirected graph** of a directed graph is obtained by replacing each directed edge with an undirected edge, ignoring the direction.

Given Graph (Fig. 1.30)



Vertices: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$

Edges in the undirected version:

$$v_1 - v_2, \quad v_1 - v_3, \quad v_1 - v_4, \quad v_2 - v_4, \quad v_2 - v_5, \quad v_1 - v_6,$$
$$v_3 - v_6, \quad v_4 - v_6, \quad v_4 - v_7, \quad v_5 - v_7, \quad v_6 - v_7, \quad v_2 - v_7$$

There are 7 vertices and 12 edges in the undirected version. A spanning tree of this graph will have exactly 6 edges and no cycles.

Solution

Spanning tree of the graph in Fig. 1.30:

$$T_1 = \{\{v_1 \to v_2\}, \{v_2 \to v_5\}, \{v_5 \to v_7\}, \{v_1 \to v_3\}, \{v_3 \to v_6\}, \{v_1 \to v_4\}\}$$
$$T_2 = \{\{v_1 \to v_2\}, \{v_2 \to v_5\}, \{v_5 \to v_7\}, \{v_1 \to v_3\}, \{v_3 \to v_6\}, \{v_2 \to v_4\}\}$$
$$T_3 = \{\{v_1 \to v_4\}, \{v_4 \to v_7\}, \{v_7 \to v_2\}, \{v_1 \to v_3\}, \{v_3 \to v_6\}, \{v_2 \to v_5\}\}$$
$$T_4 = \{\{v_1 \to v_4\}, \{v_4 \to v_7\}, \{v_1 \to v_2\}, \{v_1 \to v_3\}, \{v_3 \to v_6\}, \{v_2 \to v_5\}\}$$
$$T_5 = \{\{v_1 \to v_3\}, \{v_6 \to v_1\}, \{v_1 \to v_4\}, \{v_4 \to v_7\}, \{v_7 \to v_2\}, \{v_2 \to v_5\}\}$$
$$T_6 = \{\{v_1 \to v_3\}, \{v_6 \to v_1\}, \{v_1 \to v_4\}, \{v_2 \to v_4\}, \{v_7 \to v_2\}, \{v_5 \to v_7\}\}$$
$$T_7 = \{\{v_1 \to v_3\}, \{v_6 \to v_1\}, \{v_7 \to v_6\}, \{v_7 \to v_2\}, \{v_2 \to v_5\}, \{v_2 \to v_4\}\}$$
$$T_8 = \{\{v_1 \to v_3\}, \{v_6 \to v_1\}, \{v_7 \to v_6\}, \{v_7 \to v_2\}, \{v_2 \to v_5\}, \{v_1 \to v_4\}\}$$

```
import networkx as nx

G = nx.Graph()
edges = [
    (1, 2), (1, 3), (1, 4),
    (2, 4), (2, 5), (3, 6),
    (4, 6), (4, 7), (5, 7), (6, 7)
]
G.add_edges_from(edges)

n_spanning_trees = nx.number_of_spanning_trees(G)
print(n_spanning_trees)
```

Using the code above, we can get the number of spanning trees of the underlying undirected graph in Fig. 1.30 is **288**.

Explanation of `number_of_spanning_trees()`

The function `networkx.number_of_spanning_trees(G)` returns the total number of distinct spanning trees of a connected undirected graph G.

Internally, it applies the **Matrix-Tree Theorem**, which states:

> The number of spanning trees of a graph is equal to any cofactor (i.e., determinant of a minor) of its Laplacian matrix.

Given a graph G with n vertices, the Laplacian matrix L is defined as:

$$L = D - A$$

where:

- D is the degree matrix (a diagonal matrix where D_{ii} is the degree of vertex v_i),
- A is the adjacency matrix of the graph.

To compute the number of spanning trees, one row and the corresponding column are removed from L, and the determinant of the resulting $(n-1) \times (n-1)$ matrix is taken.

This value is guaranteed to be an integer for any connected undirected graph.

Problem 1.4

Extend the adjacency matrix graph representation by replacing those operations having an edge as argument or giving an edge or a list of edges as result, by corresponding operations having as argument or giving as result the source and target vertices of the edge or edges.

Explanation:

In many graph libraries, edge-based operations are defined using edge objects or identifiers. However, when using an adjacency matrix representation, edges are naturally represented by ordered pairs of vertices (v, w) such that the matrix entry $A[v][w] = 1$. Therefore, we can extend and redefine these operations directly in terms of vertex pairs:

- `G.del_edge(e)` ⇒ `G.del_edge(v, w)`
  Deletes the edge from vertex v to vertex w by setting $A[v][w] := 0$.
- `G.edges()` ⇒ return all pairs (v, w) such that $A[v][w] = 1$
  Returns the set of all edges as vertex pairs.
- `G.incoming(v)` ⇒ return all u such that $A[u][v] = 1$
  Returns all vertices that have an edge going into vertex v.
- `G.outgoing(v)` ⇒ return all w such that $A[v][w] = 1$
  Returns all vertices that are targets of edges going out from vertex v.
- `G.source(e)` ⇒ extract v from edge (v, w)
  The source of the edge is simply the first vertex of the pair.
- `G.target(e)` ⇒ extract w from edge (v, w)
  The target of the edge is the second vertex of the pair.

Problem 1.5

Extend the first-child, next-sibling tree representation, in order to support the operations `T.root()`, `T.number_of_children(v)`, `T.children(v)` in $\mathcal{O}(1)$ time.

Solution:

We augment the traditional first-child, next-sibling (FCNS) tree structure by storing additional information in each node and in the tree structure itself.

- `T.root()`: Maintain a direct reference to the root node in the tree object `T`. Accessing the root is then a simple pointer dereference and takes constant time.
- `T.number_of_children(v)`: Add a field `v.num_children` in each node `v`, which is incremented or decremented whenever a child is added or removed. This allows the number of children to be returned in constant time.

- `T.children(v)`: In addition to `v.first_child` and `v.next_sibling`, maintain a separate list or array `v.children[]` storing direct references to all of `v`'s children. This allows for immediate access to all children in $\mathcal{O}(1)$ time if the list is directly returned (not iterated over).

Trade-off: These enhancements increase the space complexity and require careful maintenance of the extra fields during updates (insertions, deletions). However, they provide significant performance improvements for child-related queries.

Problem 1.6

Show how to double check that the graph-based representation of a tree is indeed a tree, in time linear in the size of the tree.

To verify that a graph-based representation indeed corresponds to a valid tree, we perform the following checks in $O(n)$ time, where n is the number of nodes in the graph:

1. **Check for exactly one root node.**

   A root node is defined as a node with no incoming edges. We iterate through all nodes and count how many satisfy this condition:

   $$\text{root\_count} \leftarrow 0$$

   $$\text{for each } v \in T.vertices() : \quad \text{if } T.incoming(v).empty() : \quad \text{root\_count} \leftarrow \text{root\_count} + 1$$

   If root_count $\neq 1$, the structure is not a valid tree.

2. **Check that every non-root node has exactly one parent.**

   For each node v with incoming edges, we ensure:

   $$\text{if } \neg T.incoming(v).empty(), \text{ then } T.incoming(v).size() = 1$$

   This guarantees that every node except the root has exactly one parent.

3. **Check for connectivity and absence of cycles using DFS.**

   We perform a depth-first search (DFS) from the root node and:

   - Mark each visited node to avoid revisiting.
   - If a node is visited more than once, a cycle exists.
   - If some nodes are not visited after DFS, the graph is disconnected.

   Pseudocode:

```
visited = empty set

function DFS(v):
    if v in visited:
        return False  # cycle detected
    visited.add(v)
    for u in T.children(v):
        if not DFS(u):
            return False
    return True
```

   After DFS:

   $$\text{if } |\text{visited}| \neq T.number\_of\_nodes(), \text{ the structure is not connected.}$$

Since each operation (checking incoming/outgoing edges, DFS traversal) takes constant or linear time over all nodes and edges, the total verification process runs in $O(n)$ time.

Exercise 1.1: The standard representation of an undirected graph in the format adopted for the DIMACS Implementation Challenges consists of a problem definition line of the form `p edge n m`, where n and m are, respectively, the number of vertices and the number of edges, followed by m edge descriptor lines of the form `e i j`, each of them giving an edge as a pair of vertex numbers in the range 1 to n. Comment lines of the form `c ...` are also allowed. Implement procedures to read a DIMACS graph and to write a graph in DIMACS format.

The DIMACS format represents an undirected graph in a standardized way:

- Comment lines start with `c` and can be ignored.
- A line starting with `p edge n m` defines the number of vertices (n) and edges (m).
- Each subsequent line starting with `e i j` defines an undirected edge between vertex i and vertex j.

To handle this format, we implement two procedures:
Reading a DIMACS graph We read each line of the input file:

- Skip lines starting with `c`.
- Parse the line `p edge n m` to get the number of vertices and edges.
- Parse each edge line `e u v` and store the edges as pairs (u, v).

Writing a DIMACS graph To output a graph in DIMACS format:

- Print the line `p edge n m`, where m is the number of edges.
- For each edge (u, v), write the line `e u v`.

Exercise 1.2: The external representation of a graph in the Stanford GraphBase (SGB) format consists essentially of a first line of the form `* GraphBase graph (utiltypes ...,nV,mA)`, where n and m are, respectively, the number of vertices and the number of edges; a second line containing an identification string; a `* Vertices` line; n vertex descriptor lines of the form `label,Ai,0,0`, where i is the number of the first edge in the range 0 to $m-1$ going out of the vertex and `label` is a string label; an `* Arcs` line; m edge descriptor lines of the form `V j,Ai,label,0`, where j is the number of the target vertex in the range 0 to $n-1$, i is the number of the next edge in the range 0 to $m-1$ going out of the same source vertex, and `label` is an integer label; and a last `* Checksum ...` line. Further, in the description of a vertex with no outgoing edge, or an edge with no successor going out of the same source vertex, `Ai` becomes `0`. Implement procedures to read a SGB graph and to write a graph in SGB format.

The **Stanford GraphBase (SGB)** format represents a directed graph in a structured way:

- The first line begins with `* GraphBase graph (utiltypes ...,nV,mA)` where n is the number of vertices and m is the number of edges.
- The second line is an identification string (can be stored or ignored).
- The `* Vertices` section follows, with n lines describing vertices in the format: `label,Ai,0,0`
  - `label` is a string identifier.
  - `Ai` indicates the index (from 0 to $m-1$) of the first outgoing edge from that vertex.
  - If there is no outgoing edge, `Ai` is `0`.
- Then comes the `* Arcs` section, with m lines describing directed edges (arcs) in the format: `Vj,Ai,label,0`
  - `Vj` is the target vertex (in the range 0 to $n-1$).
  - `Ai` is the index of the next edge from the same source vertex.
  - If there is no next edge, `Ai` is `0`.
  - `label` is an integer label for the edge.
- A final line `* Checksum ...` concludes the graph.

To process this format, we implement two procedures:
Reading a SGB graph

- Parse the first line to extract n and m.
- Skip or store the second line (identifier).
- In the `* Vertices` section:
  - Read each vertex's label and first outgoing edge index.
  - Store vertex labels and associate them with outgoing edge indices.
- In the `* Arcs` section:

- Read each edge's target vertex, next edge index, and label.
- Construct an adjacency list or edge list accordingly.

Writing a SGB graph

- Output the graph header and identification string.
- Write the `* Vertices` section with vertex descriptors.
- Write the `* Arcs` section with edge descriptors.
- Append a `* Checksum ...` line if needed.

Bài toán 6 (Tree edit distance)**.** *Viết chương trình* C/C++, Python *để giải bài toán tree edit distance problem bằng cách sử dụng: (a) Backtracking. (b) Branch-&-bound. (c) Divide-&-conquer – chia để trị. (d) Dynamic programming – Quy hoạch động.*

(a) Phương pháp Backtracking
Giải thích và công thức:

- **Bài toán:** Cho hai cây có gốc $T_1 = (V_1, E_1)$ và $T_2 = (V_2, E_2)$ (có thứ tự), tìm tất cả các ánh xạ hợp lệ từ các nút của T_1 sang các nút của T_2 (hoặc nút giả λ), tương ứng với các phép biến đổi cây (tree edit mapping).
- **Ý tưởng:** Duyệt tất cả các ánh xạ từng phần $M \subseteq V_1 \times (V_2 \cup \{\lambda\})$ bằng đệ quy quay lui (backtracking), đảm bảo các ràng buộc:
  - Mỗi nút $v \in V_1$ được gán cho đúng một nút $w \in V_2$ cùng độ sâu (hoặc λ nếu xóa v).
  - Không có hai nút $v_1, v_2 \in V_1$ cùng gán vào một nút $w \in V_2$ (trừ λ) – ánh xạ là song ánh trên V_2.
  - Nếu v là cha của x trong T_1 và w là ảnh của v trong T_2, thì ảnh của x (nếu không phải λ) phải là con của w trong T_2.
  - Ràng buộc thứ tự anh em: nếu x là anh phải của v trong T_1, thì ảnh của x (nếu không phải λ) không được là anh trái của w trong T_2.
- **Cách xây dựng ánh xạ:**
  - Duyệt các nút v của T_1 theo thứ tự preorder.
  - Với mỗi v, thử gán vào từng w thuộc tập ứng viên $C[v]$ (các nút $w \in V_2$ cùng độ sâu với v hoặc λ).
  - Sau mỗi lần gán $v \to w$, cập nhật lại tập ứng viên cho các nút còn lại (loại bỏ w khỏi các ứng viên khác nếu $w \neq \lambda$; loại bỏ các y không phải con của w khỏi ứng viên của các con x của v; loại bỏ các y vi phạm thứ tự anh em).
  - Nếu đã gán hết các nút của T_1, lưu ánh xạ vào danh sách kết quả.
- **Công thức đệ quy:**
  - Gọi M là ánh xạ hiện tại, C là tập ứng viên hiện tại, v là nút đang xét.
  - Với mỗi $w \in C[v]$:
    - Gán $M[v] = w$.
    - Nếu v là nút cuối cùng: lưu M vào kết quả.
    - Ngược lại: cập nhật C thành N bằng hàm refine (loại các ứng viên không hợp lệ), gọi đệ quy với nút tiếp theo.
- **Tập ứng viên $C[v]$:** Gồm các nút $w \in V_2$ cùng độ sâu với v và nút giả λ (tương ứng với phép xóa).

Giải thích code:

- **Các lớp chính:**
  - `TreeNode`: đại diện cho một nút của cây, gồm nhãn, danh sách con, cha, độ sâu, thứ tự preorder, thứ tự anh em.
  - `Tree`: đại diện cho một cây, gồm danh sách nút, gốc, nút giả λ, độ sâu lớn nhất.
- **Các hàm chính:**

  - `assign_preorder_and_depth`: Gán số preorder, độ sâu, thứ tự anh em cho từng nút bằng duyệt DFS.
  - `add_dummy`: Thêm nút giả λ vào cây (dùng cho phép xóa).
  - `set_up_candidate_nodes`: Khởi tạo tập ứng viên $C[v]$ cho mỗi v của T_1.
  - `refine_candidate_nodes`: Sau khi gán $v \to w$, cập nhật lại C cho các nút còn lại:
    * Loại w khỏi các ứng viên khác nếu $w \neq \lambda$ (ràng buộc song ánh).
    * Với mỗi con x của v, loại các y không phải con của w khỏi $C[x]$.
    * Ràng buộc thứ tự anh em: nếu x là anh phải của v, loại các y là anh trái của w khỏi $C[x]$.
  - `extend_tree_edit`: Hàm đệ quy chính, thử tất cả các ánh xạ $v \to w$ và gọi đệ quy cho nút tiếp theo.
  - `backtracking_tree_edit`: Hàm tổng, khởi tạo các biến, gọi hàm đệ quy và trả về danh sách ánh xạ hợp lệ.

- **Ý nghĩa các biến số chính:**
  - `T1, T2`: hai cây đầu vào.
  - `M`: ánh xạ hiện tại từ nút T_1 sang T_2 (hoặc λ).
  - `L`: danh sách các ánh xạ hợp lệ tìm được.
  - `C`: tập ứng viên cho từng nút T_1.
  - `v, w`: nút đang xét của T_1 và ứng viên ánh xạ của T_2.
  - `preorder_list_T1`: danh sách các nút T_1 theo thứ tự preorder.
- **Độ phức tạp:** Phương pháp này liệt kê tất cả các ánh xạ hợp lệ, nên độ phức tạp là hàm mũ theo số nút.
- **Ưu điểm:** Đơn giản, dễ cài đặt, cho phép sinh tất cả ánh xạ hợp lệ (dùng cho kiểm thử, sinh ví dụ nhỏ).
- **Nhược điểm:** Không hiệu quả với cây lớn do số ánh xạ hợp lệ tăng rất nhanh.

Bài toán 7 (Tree traversal – Duyệt cây)**.** *Viết chương trình* C/C++, Python *để duyệt cây: (a) preorder traversal. (b) postorder traversal. (c) top-down traveral. (d) bottom-up traversal.*

(a) Preorder traversal
Giải thích và công thức:

- **Duyệt cây theo thứ tự trước (preorder traversal):** *Preorder* là phương pháp duyệt cây mà tại mỗi nút, ta thăm nút đó trước, sau đó lần lượt duyệt các cây con từ trái sang phải.
- **Đệ quy:**
  - Gọi T là một cây gốc, u là gốc của T, $C(u)$ là tập các con trực tiếp của u.
  - Thứ tự duyệt preorder của T là: đầu tiên thăm u, sau đó lần lượt duyệt preorder từng cây con T_v với $v \in C(u)$ (theo thứ tự từ trái sang phải).
  - Nếu u là lá (không có con), chỉ thăm u. Nếu u có các con $v_1, v_2, ..., v_k$, thì thứ tự duyệt là: $u, \text{preorder}(v_1), \text{preorder}(v_2), ..., \text{preorder}(v_k)$.
- **Ý tưởng chi tiết:**
  - **Bước 1:** Bắt đầu từ gốc u, in ra nhãn của u (thăm u).
  - **Bước 2:** Duyệt qua từng con v của u (theo thứ tự đã cho), với mỗi v ta gọi đệ quy duyệt preorder cây con gốc v.

Giải thích code:

- **Biến số quan trọng:**
  - n: số lượng đỉnh của cây.
  - `tree`: danh sách kề, `tree[u]` chứa các con trực tiếp của đỉnh u.
  - `is_child`: mảng đánh dấu đỉnh nào là con (để tìm gốc).
  - `root`: đỉnh gốc của cây (không là con của đỉnh nào).

- **Đọc input:** Mỗi dòng gồm: u k v_1 v_2 ... v_k (đỉnh u có k con là $v_1, ..., v_k$).
- **Hàm `preorder(u, tree)`:** In ra u, sau đó đệ quy duyệt từng con v của u.
- **Kết quả:** In ra thứ tự các đỉnh theo duyệt preorder.

(b) Postorder traversal
Giải thích và công thức:

- **Duyệt cây theo thứ tự sau (postorder traversal):** *Postorder* là phương pháp duyệt cây mà tại mỗi nút, ta duyệt tất cả các cây con từ trái sang phải trước, sau đó mới thăm nút đó.
- **Đệ quy:**
  - Gọi T là một cây gốc, u là gốc của T, $C(u)$ là tập các con trực tiếp của u.
  - Thứ tự duyệt postorder của T là: đầu tiên lần lượt duyệt postorder từng cây con T_v với $v \in C(u)$ (theo thứ tự từ trái sang phải), sau đó mới thăm u.
  - Nếu u là lá (không có con), chỉ thăm u.
  - Nếu u có các con $v_1, v_2, ..., v_k$, thì thứ tự duyệt là: $\text{postorder}(v_1), \text{postorder}(v_2), ..., \text{postorder}(v_k), u$.
- **Ý tưởng chi tiết:**
  - **Bước 1:** Bắt đầu từ gốc u, lần lượt duyệt qua từng con v của u (theo thứ tự đã cho), với mỗi v ta gọi đệ quy duyệt postorder cây con gốc v.
  - **Bước 2:** Sau khi duyệt xong tất cả các con, in ra nhãn của u (thăm u).
  - **Bản chất:** Quá trình này là "đi hết các nhánh con trước, cha sau", đảm bảo thứ tự duyệt là: con trái $\rightarrow$ con phải $\rightarrow$ cha.

Giải thích code:

- **Biến số quan trọng:**
  - n: số lượng đỉnh của cây.
  - `tree`: danh sách kề, `tree[u]` chứa các con trực tiếp của đỉnh u.
  - `is_child`: mảng đánh dấu đỉnh nào là con (để tìm gốc).
  - `root`: đỉnh gốc của cây (không là con của đỉnh nào).
- **Đọc input:** Mỗi dòng gồm: u k v_1 v_2 ... v_k (đỉnh u có k con là $v_1, ..., v_k$).
- **Hàm `postorder(u, tree)`:** Đầu tiên đệ quy duyệt từng con v của u, sau đó in ra u.
- **Kết quả:** In ra thứ tự các đỉnh theo duyệt postorder.

(c) Top-down traversal
Giải thích và công thức:

- **Duyệt cây top-down (từ gốc xuống lá):** Top-down là phương pháp duyệt cây mà tại mỗi nút, ta xử lý nút đó trước, sau đó truyền thông tin (nếu có) từ cha xuống các con, rồi tiếp tục duyệt các con.
- **Thứ tự thăm:** Các đỉnh được thăm theo thứ tự không giảm của độ sâu (depth), và các đỉnh cùng độ sâu được thăm từ trái sang phải (theo thứ tự con trong danh sách kề).
- **Đệ quy:**
  - Gọi T là một cây gốc, u là gốc của T, $C(u)$ là tập các con trực tiếp của u.
  - Khi duyệt top-down, ta có thể truyền một đại lượng (ví dụ: độ sâu, tổng giá trị từ gốc đến u, ...) từ cha xuống con.
  - Với mỗi lời gọi `top_down(u, depth)`, ta xử lý u ở độ sâu *depth*, sau đó lần lượt duyệt các con v của u với $depth + 1$ (theo thứ tự trái sang phải).
  - Ví dụ: Nếu truyền độ sâu, thì info là depth, update(depth) = depth + 1.
- **Ý tưởng chi tiết:**
  - **Bước 1:** Bắt đầu từ gốc u với thông tin ban đầu (ví dụ: depth = 0), xử lý u (in ra nhãn, độ sâu, ...).

- **Bước 2:** Với mỗi con v của u (theo thứ tự trái sang phải), truyền thông tin mới (ví dụ: depth + 1) và gọi đệ quy duyệt top-down cây con gốc v.
- **Bản chất:** Tất cả các đỉnh ở độ sâu d sẽ được thăm trước khi đến các đỉnh ở độ sâu $d+1$, và các đỉnh cùng độ sâu được thăm từ trái sang phải.

Giải thích code:

- **Biến số quan trọng:**
  - n: số lượng đỉnh của cây.
  - `tree`: danh sách kề, `tree[u]` chứa các con trực tiếp của đỉnh u.
  - `is_child`: mảng đánh dấu đỉnh nào là con (để tìm gốc).
  - `root`: đỉnh gốc của cây (không là con của đỉnh nào).
  - `depth`: độ sâu hiện tại của đỉnh u (truyền từ cha xuống con).
- **Đọc input:** Mỗi dòng gồm: u k v_1 v_2 ... v_k (đỉnh u có k con là $v_1, ..., v_k$).
- **Hàm `top_down(u, tree, depth)`:** Xử lý u (in ra nhãn, độ sâu), sau đó đệ quy duyệt từng con v của u với `depth` + 1 (theo thứ tự trái sang phải).
- **Kết quả:** In ra từng đỉnh và độ sâu tương ứng theo thứ tự top-down: các đỉnh ở độ sâu nhỏ hơn được in trước, các đỉnh cùng độ sâu in từ trái sang phải.

(d) Bottom-up traversal
Giải thích và công thức:

- **Duyệt cây bottom-up (từ lá lên gốc):** Bottom-up là phương pháp duyệt cây mà các đỉnh được thăm theo thứ tự không giảm của chiều cao (height), các đỉnh cùng chiều cao được thăm theo thứ tự không giảm của độ sâu (depth), các đỉnh cùng chiều cao và độ sâu được thăm từ trái sang phải.
- **Thứ tự thăm:**
  - Đầu tiên thăm tất cả các đỉnh có chiều cao nhỏ nhất (tức là các lá), sau đó đến các đỉnh có chiều cao lớn hơn, ... cuối cùng là gốc (chiều cao lớn nhất).
  - Các đỉnh cùng chiều cao được thăm theo thứ tự không giảm của độ sâu (tức là các đỉnh ở gần gốc hơn được in sau).
  - Nếu cùng chiều cao và cùng độ sâu, các đỉnh được thăm từ trái sang phải (theo thứ tự con trong danh sách kề).
- **Công thức:**
  - Gọi $h(u)$ là chiều cao của đỉnh u, $d(u)$ là độ sâu của u.
  - Duyệt qua tất cả các đỉnh, sắp xếp theo bộ $(h(u), d(u),$ thứ tự trái sang phải) tăng dần, rồi in ra.
  - Chiều cao $h(u)$ được tính đệ quy: $h(u) = 1 + \max\{h(v) : v \text{ là con của } u\}$, lá có $h(u) = 0$.
- **Ý tưởng chi tiết:**
  - **Bước 1:** Duyệt cây để tính chiều cao và độ sâu cho từng đỉnh.
  - **Bước 2:** Gom tất cả các đỉnh lại, sắp xếp theo chiều cao tăng dần, cùng chiều cao thì theo độ sâu tăng dần, cùng chiều cao và độ sâu thì theo thứ tự trái sang phải.
  - **Bước 3:** In ra các đỉnh theo từng mức chiều cao.

Giải thích code:

- **Biến số quan trọng:**
  - n: số lượng đỉnh của cây.
  - `tree`: danh sách kề, `tree[u]` chứa các con trực tiếp của đỉnh u.
  - `is_child`: mảng đánh dấu đỉnh nào là con (để tìm gốc).
  - `root`: đỉnh gốc của cây (không là con của đỉnh nào).
  - `depths[u]`: độ sâu của đỉnh u.

  - `heights[u]`: chiều cao của đỉnh u.

- **Đọc input:** Mỗi dòng gồm: u k v_1 v_2 ... v_k (đỉnh u có k con là $v_1, ..., v_k$).
- **Hàm `dfs_height(u, tree, depth, depths, heights)`:** Tính đệ quy chiều cao và độ sâu cho từng đỉnh.
- **Hàm `bottom_up(tree, root, n)`:** Gom thông tin các đỉnh, sắp xếp và in ra theo thứ tự bottom-up.
- **Kết quả:** In ra các đỉnh theo từng mức chiều cao, mỗi mức là các đỉnh cùng chiều cao, theo thứ tự độ sâu tăng dần, trái sang phải.

2.1 Breadth-first search algorithm – Thuật toán tìm kiếm theo chiều rộng

Bài toán 8. *Let $G = (V, E)$ be a finite simple graph. Implement the breadth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều rộng (BFS) trên đồ thị đơn hữu hạn, bắt đầu từ một đỉnh cho trước.
- **Ý tưởng chính:** BFS duyệt đồ thị theo từng lớp, sử dụng hàng đợi (queue) để lần lượt thăm các đỉnh kề gần nhất trước.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề (adjacency list), `adj[u]` chứa các đỉnh kề với đỉnh u.
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `queue`: Hàng đợi để lưu trữ các đỉnh sẽ được thăm tiếp theo.
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh.
  2. Nhập đỉnh bắt đầu *start*.
  3. Khởi tạo mảng `visited` với tất cả giá trị `false`, đánh dấu `visited[start] = true`.
  4. Đưa *start* vào hàng đợi.
  5. Lặp cho đến khi hàng đợi rỗng:
     - Lấy ra đỉnh đầu hàng đợi, in ra đỉnh đó.
     - Với mỗi đỉnh kề chưa được thăm, đánh dấu đã thăm và đưa vào hàng đợi.
- **Lưu ý:** Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều.

Bài toán 9. *Let $G = (V, E)$ be a finite multigraph. Implement the breadth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều rộng (BFS) trên đa đồ thị hữu hạn, cho phép nhiều cạnh giữa hai đỉnh nhưng không có khuyên (loop).
- **Ý tưởng chính:** BFS vẫn áp dụng được cho multigraph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp) giữa hai đỉnh. Đảm bảo không duyệt lại đỉnh đã thăm.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các đỉnh kề với u (có thể lặp lại nếu có nhiều cạnh).
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `queue`: Hàng đợi để lưu trữ các đỉnh sẽ được thăm tiếp theo.
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại, không cho phép cạnh khuyên $u = v$).
  2. Nhập đỉnh bắt đầu *start*.
  3. Khởi tạo mảng `visited` với tất cả giá trị `false`, đánh dấu `visited[start] = true`.

4. Đưa *start* vào hàng đợi.
5. Lặp cho đến khi hàng đợi rỗng:
   - Lấy ra đỉnh đầu hàng đợi, in ra đỉnh đó.
   - Với mỗi đỉnh kề chưa được thăm, đánh dấu đã thăm và đưa vào hàng đợi.

- **Lưu ý:** Đa đồ thị cho phép nhiều cạnh giữa hai đỉnh, nhưng không cho phép cạnh khuyên ($u = v$). Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều.

Bài toán 10. *Let $G = (V, E)$ be a general graph. Implement the breadth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều rộng (BFS) trên đồ thị tổng quát (general graph) hữu hạn, cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên (loop).
- **Ý tưởng chính:** BFS vẫn áp dụng được cho general graph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp và cạnh khuyên) giữa hai đỉnh. Đảm bảo không duyệt lại đỉnh đã thăm.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các đỉnh kề với u (có thể lặp lại nếu có nhiều cạnh, và có thể có $u = v$ nếu là cạnh khuyên).
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `queue`: Hàng đợi để lưu trữ các đỉnh sẽ được thăm tiếp theo.
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại và có thể là cạnh khuyên).
  2. Nhập đỉnh bắt đầu *start*.
  3. Khởi tạo mảng `visited` với tất cả giá trị `false`, đánh dấu `visited[start] = true`.
  4. Đưa *start* vào hàng đợi.
  5. Lặp cho đến khi hàng đợi rỗng:
     - Lấy ra đỉnh đầu hàng đợi, in ra đỉnh đó.
     - Với mỗi đỉnh kề chưa được thăm, đánh dấu đã thăm và đưa vào hàng đợi.
- **Lưu ý:** Đồ thị tổng quát cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên ($u = v$). Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều (kể cả cạnh khuyên).

2.2 Depth-first search algorithm – Thuật toán tìm kiếm theo chiều rộng

Bài toán 11. *Let $G = (V, E)$ be a finite simple graph. Implement the depth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều sâu (DFS) trên đồ thị đơn hữu hạn, bắt đầu từ một đỉnh cho trước.
- **Ý tưởng chính:** DFS duyệt đồ thị bằng cách đi sâu vào một nhánh trước khi quay lại và thử nhánh khác. Sử dụng đệ quy hoặc stack để thực hiện.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề (adjacency list), `adj[u]` chứa các đỉnh kề với đỉnh u.
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `stack`: Cấu trúc dữ liệu stack để lưu trữ các đỉnh cần thăm (trong phiên bản iterative).
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh.
  2. Nhập đỉnh bắt đầu *start*.
  3. Khởi tạo mảng `visited` với tất cả giá trị `false`.
  4. Thực hiện DFS bằng hai cách:

  - **Đệ quy:** Gọi hàm `dfs_recursive` với đỉnh bắt đầu.
    * **Cách thức:**
    (a) Đánh dấu đỉnh hiện tại là đã thăm.
    (b) In ra đỉnh hiện tại.
    (c) Với mỗi đỉnh kề chưa thăm, gọi đệ quy `dfs_recursive(adj, visited, neighbor)`.
    (d) Khi không còn đỉnh kề nào chưa thăm, quay lại (backtrack).
  - **Iterative:** Sử dụng stack trong hàm `dfs_iterative`.
5. Trong mỗi bước DFS:
  - Đánh dấu đỉnh hiện tại là đã thăm.
  - In ra đỉnh hiện tại.
  - Thăm tất cả các đỉnh kề chưa được thăm.

- **Lưu ý:**
  - Trong phiên bản iterative, các đỉnh kề được đẩy vào stack theo thứ tự ngược để duy trì thứ tự DFS.
  - Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều.

Bài toán 12. *Let $G = (V, E)$ be a finite multigraph. Implement the depth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều sâu (DFS) trên đa đồ thị hữu hạn, cho phép nhiều cạnh giữa hai đỉnh nhưng không có khuyên (loop).
- **Ý tưởng chính:** DFS vẫn áp dụng được cho multigraph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp) giữa hai đỉnh. Đảm bảo không duyệt lại đỉnh đã thăm.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các đỉnh kề với u (có thể lặp lại nếu có nhiều cạnh).
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `stack`: Cấu trúc dữ liệu stack để lưu trữ các đỉnh cần thăm (trong phiên bản iterative).
- **Các bước chính:**
1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại, không cho phép cạnh khuyên $u = v$).
2. Nhập đỉnh bắt đầu *start*.
3. Khởi tạo mảng `visited` với tất cả giá trị `false`.
4. Thực hiện DFS bằng hai cách:
  - **Đệ quy:** Gọi hàm `dfs_recursive` với đỉnh bắt đầu.
    * **Cách thức:**
    (a) Đánh dấu đỉnh hiện tại là đã thăm.
    (b) In ra đỉnh hiện tại.
    (c) Với mỗi đỉnh kề chưa thăm, gọi đệ quy `dfs_recursive(adj, visited, neighbor)`.
    (d) Khi không còn đỉnh kề nào chưa thăm, quay lại (backtrack).
  - **Iterative:** Sử dụng stack trong hàm `dfs_iterative`.
5. Trong mỗi bước DFS:
  - Đánh dấu đỉnh hiện tại là đã thăm.
  - In ra đỉnh hiện tại.
  - Thăm tất cả các đỉnh kề chưa được thăm.

- **Lưu ý:**
  - Đa đồ thị cho phép nhiều cạnh giữa hai đỉnh, nhưng không cho phép cạnh khuyên ($u = v$).
  - Trong phiên bản iterative, các đỉnh kề được đẩy vào stack theo thứ tự ngược để duy trì thứ tự DFS.
  - Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều.

Bài toán 13. *Let $G = (V, E)$ be a general graph. Implement the depth-first search on G.*

Code Explanation:

- **Mục tiêu:** Thực hiện thuật toán duyệt theo chiều sâu (DFS) trên đồ thị tổng quát (general graph) hữu hạn, cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên (loop).
- **Ý tưởng chính:** DFS vẫn áp dụng được cho general graph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp và cạnh khuyên) giữa hai đỉnh. Đảm bảo không duyệt lại đỉnh đã thăm.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các đỉnh kề với u (có thể lặp lại nếu có nhiều cạnh, và có thể có $u = v$ nếu là cạnh khuyên).
  - `visited`: Mảng boolean để đánh dấu các đỉnh đã được thăm.
  - `stack`: Cấu trúc dữ liệu stack để lưu trữ các đỉnh cần thăm (trong phiên bản iterative).
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại và có thể là cạnh khuyên) cùng trọng số.
  2. Nhập đỉnh bắt đầu *start*.
  3. Khởi tạo mảng `visited` với tất cả giá trị `false`.
  4. Thực hiện DFS bằng hai cách:
     - **Đệ quy:** Gọi hàm `dfs_recursive` với đỉnh bắt đầu.
       - **Cách thức:**
       (a) Đánh dấu đỉnh hiện tại là đã thăm.
       (b) In ra đỉnh hiện tại.
       (c) Với mỗi đỉnh kề chưa thăm, gọi đệ quy `dfs_recursive(adj, visited, neighbor)`.
       (d) Khi không còn đỉnh kề nào chưa thăm, quay lại (backtrack).
     - **Iterative:** Sử dụng stack trong hàm `dfs_iterative`.
  5. Trong mỗi bước DFS:
     - Đánh dấu đỉnh hiện tại là đã thăm.
     - In ra đỉnh hiện tại.
     - Thăm tất cả các đỉnh kề chưa được thăm.
- **Lưu ý:**
  - Đồ thị tổng quát cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên ($u = v$).
  - Trong phiên bản iterative, các đỉnh kề được đẩy vào stack theo thứ tự ngược để duy trì thứ tự DFS.
  - Đồ thị được coi là vô hướng (undirected), nên mỗi cạnh được thêm vào cả hai chiều (kể cả cạnh khuyên).

3 Project 5: Shortest Path Problems on Graphs – Đồ Án 5: Các Bài Toán Tìm Đường Đi Ngắn Nhất Trên Đồ Thị

3.1 Dijkstra's algorithm – Thuật toán Dijkstra

Bài toán 14. *Let $G = (V, E)$ be a finite simple graph. Implement the Dijkstra's algorithm to find the shortest path problem on G.*

Code Explanation:

- **Mục tiêu:** Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trên đồ thị đơn hữu hạn với trọng số không âm.
- **Ý tưởng chính:** Sử dụng hàng đợi ưu tiên (priority queue) để luôn chọn đỉnh có khoảng cách tạm thời nhỏ nhất, cập nhật dần dần các khoảng cách ngắn nhất từ nguồn đến các đỉnh.
- **Các biến và cấu trúc chính:**

  - `adj`: Danh sách kề (adjacency list), `adj[u]` chứa các cặp (v, w) với v là đỉnh kề u, w là trọng số cạnh $u - v$.
  - `dist`: Mảng lưu khoảng cách ngắn nhất từ nguồn đến mỗi đỉnh, khởi tạo là `INF` (vô cùng lớn).
  - `priority_queue`: Hàng đợi ưu tiên kiểu min-heap, luôn lấy ra đỉnh có khoảng cách tạm thời nhỏ nhất.

- **Các bước chính:**

  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh cùng trọng số.
  2. Nhập đỉnh nguồn src.
  3. Khởi tạo `dist[src] = 0`, các đỉnh còn lại là `INF`.
  4. Đưa $(0, src)$ vào hàng đợi ưu tiên.
  5. Lặp cho đến khi hàng đợi rỗng:
     - Lấy ra đỉnh u có khoảng cách tạm thời nhỏ nhất.
     - Nếu đã có đường đi ngắn hơn đến u, bỏ qua.
     - Với mỗi đỉnh kề v của u, nếu tìm được đường đi ngắn hơn qua u, cập nhật `dist[v]` và đưa vào hàng đợi.
  6. In ra khoảng cách ngắn nhất từ nguồn đến các đỉnh còn lại (nếu không tới được thì in `INF`).

Bài toán 15. *Let $G = (V, E)$ be a finite multigraph. Implement the Dijkstra's algorithm to find the shortest path problem on G.*

Code Explanation:

- **Mục tiêu:** Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trên đồ thị đa đồ thị (multigraph) hữu hạn với trọng số không âm, cho phép nhiều cạnh giữa hai đỉnh nhưng không có khuyên (loop).
- **Ý tưởng chính:** Thuật toán Dijkstra vẫn áp dụng được cho multigraph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp) giữa hai đỉnh.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các cặp (v, w) với v là đỉnh kề u, w là trọng số cạnh $u - v$ (có thể có nhiều cặp giống nhau nếu có nhiều cạnh).
  - `dist`: Mảng lưu khoảng cách ngắn nhất từ nguồn đến mỗi đỉnh, khởi tạo là `INF`.
  - `priority_queue`: Hàng đợi ưu tiên kiểu min-heap.
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại) cùng trọng số. Nếu nhập cạnh khuyên ($u = v$), bỏ qua.
  2. Nhập đỉnh nguồn src.
  3. Khởi tạo `dist[src] = 0`, các đỉnh còn lại là `INF`.
  4. Đưa $(0, src)$ vào hàng đợi ưu tiên.
  5. Lặp cho đến khi hàng đợi rỗng:
     - Lấy ra đỉnh u có khoảng cách tạm thời nhỏ nhất.
     - Nếu đã có đường đi ngắn hơn đến u, bỏ qua.
     - Với mỗi cạnh kề $u - v$ (kể cả trùng lặp), nếu tìm được đường đi ngắn hơn qua u, cập nhật `dist[v]` và đưa vào hàng đợi.
  6. In ra khoảng cách ngắn nhất từ nguồn đến các đỉnh còn lại (nếu không tới được thì in `INF`).
- **Lưu ý:** Đa đồ thị cho phép nhiều cạnh giữa hai đỉnh, nhưng không cho phép cạnh khuyên ($u = v$).

Bài toán 16. *Let $G = (V, E)$ be a general graph. Implement the Dijkstra's algorithm to find the shortest path problem on G.*

Code Explanation:

- **Mục tiêu:** Tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trên đồ thị tổng quát (general graph) hữu hạn với trọng số không âm, cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên (loop).
- **Ý tưởng chính:** Thuật toán Dijkstra vẫn áp dụng được cho general graph. Khi duyệt các cạnh kề, xét tất cả các cạnh (kể cả trùng lặp và cạnh khuyên) giữa hai đỉnh.
- **Các biến và cấu trúc chính:**
  - `adj`: Danh sách kề, `adj[u]` chứa tất cả các cặp (v, w) với v là đỉnh kề u, w là trọng số cạnh $u - v$ (có thể có nhiều cặp giống nhau nếu có nhiều cạnh, và có thể có $u = v$ nếu là cạnh khuyên).
  - `dist`: Mảng lưu khoảng cách ngắn nhất từ nguồn đến mỗi đỉnh, khởi tạo là `INF`.
  - `priority_queue`: Hàng đợi ưu tiên kiểu min-heap.
- **Các bước chính:**
  1. Nhập số đỉnh n, số cạnh m và danh sách các cạnh (có thể lặp lại và có thể là cạnh khuyên) cùng trọng số.
  2. Nhập đỉnh nguồn src.
  3. Khởi tạo `dist[src] = 0`, các đỉnh còn lại là `INF`.
  4. Đưa $(0, src)$ vào hàng đợi ưu tiên.
  5. Lặp cho đến khi hàng đợi rỗng:
     - Lấy ra đỉnh u có khoảng cách tạm thời nhỏ nhất.
     - Nếu đã có đường đi ngắn hơn đến u, bỏ qua.
     - Với mỗi cạnh kề $u - v$ (kể cả trùng lặp và cạnh khuyên), nếu tìm được đường đi ngắn hơn qua u, cập nhật `dist[v]` và đưa vào hàng đợi.
  6. In ra khoảng cách ngắn nhất từ nguồn đến các đỉnh còn lại (nếu không tới được thì in `INF`).
- **Lưu ý:** Đồ thị tổng quát cho phép nhiều cạnh giữa hai đỉnh và cho phép cạnh khuyên ($u = v$).