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 26 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.

(d) Giải thích thuật toán chuyển đổi biểu diễn cây:

- **Mục tiêu:** Chương trình này chuyển đổi giữa 3 dạng biểu diễn chính của cây (tree) đơn (undirected simple tree): Mảng Cha (Array of Parents - AoP), Con Đầu Tiên Anh Em Kế Tiếp (First-Child Next-Sibling - FCNS), và biểu diễn dựa trên đồ thị (sử dụng Danh sách kề mở rộng - Extended Adjacency List - EAL).
- **Cây (Tree):** Là một đồ thị liên thông và không có chu trình. Mỗi cây (không rỗng) có đúng một gốc (root) và không có cạnh song song hay khuyên (self-loop).
- **3 dạng biểu diễn chính:**
  - **Mảng Cha (Array of Parents - AoP):** Một mảng P có n phần tử, trong đó $P[v]$ là đỉnh cha của v. Nếu v là gốc, $P[v] = -1$.
  - **Con Đầu Tiên Anh Em Kế Tiếp (FCNS):** Một cặp hai mảng F và N có n phần tử.
    - $F[v]$: Con đầu tiên của v. Nếu v là lá (leaf node), $F[v] = -1$.
    - $N[v]$: Anh em kế tiếp của v. Nếu v là con cuối cùng của cha nó, $N[v] = -1$.
  - **Biểu diễn dựa trên đồ thị (sử dụng Extended Adjacency List - EAL):** Biểu diễn cây như một đồ thị không định hướng sử dụng cấu trúc Extended Adjacency List (EAL).
    - `edges`: Danh sách các cặp (u, v) đại diện cho các cạnh trong cây.
    - `incoming[v]`: Danh sách các chỉ số của các cạnh trong `edges` mà kết nối với v.
    - `outgoing[u]`: Danh sách các chỉ số của các cạnh trong `edges` mà kết nối với u.
    - m: Tổng số cạnh trong cây.
- **Biểu diễn trung gian (TreeChildrenList):** Để đơn giản hóa quá trình chuyển đổi, chương trình sử dụng một biểu diễn nội bộ `TreeChildrenList`. Đây là một mảng n danh sách, trong đó `children[u]` là danh sách các con trực tiếp của u. Các danh sách con này luôn được sắp xếp để đảm bảo tính nhất quán (đặc biệt cho FCNS).

Công thức chuyển đổi:

- **TreeChildrenList ↔ Mảng Cha:**
  - CL → PA (`children_list_to_parent_array`): Duyệt qua danh sách con của mỗi đỉnh. Nếu v là con của u, thì u là cha của v, nên $P[v] = u$.
  - PA → CL (`parent_array_to_children_list`): Duyệt qua mảng cha. Nếu $P[v] \neq -1$, thì $P[v]$ là cha của v, nên thêm v vào danh sách con của $P[v]$. Sau đó, sắp xếp các danh sách con.
- **TreeChildrenList ↔ FCNS:**
  - CL → FCNS (`children_list_to_fcns`): Duyệt qua danh sách con đã sắp xếp của mỗi đỉnh u. Con đầu tiên trong danh sách là $F[u]$. Với các con còn lại, con thứ i sẽ có anh em kế tiếp là con thứ $i+1$, nên $N[con\_th\grave{u}\_i] = con\_th\grave{u}\_i + 1$.
  - FCNS → CL (`fcns_to_children_list`): Sử dụng BFS bắt đầu từ gốc. Với mỗi đỉnh u được thăm, tìm con đầu tiên $F[u]$. Sau đó, dùng $N[F[u]]$ để tìm các anh em kế tiếp cho đến khi gặp -1. Thêm các con này vào danh sách con của u.
- **TreeChildrenList ↔ TreeExtendedAdjacencyList:**
  - CL → TreeEAL (`children_list_to_tree_eal`): Duyệt qua danh sách con. Với mỗi cặp cha-con (u, v), thêm cạnh (u, v) vào `teal.edges`. Cập nhật `incoming` và `outgoing` cho cả u và v (vì EAL được xem là biểu diễn không định hướng cho các cạnh của cây).
  - TreeEAL → CL (`tree_eal_to_children_list`): Sử dụng BFS bắt đầu từ gốc. Tạo một danh sách kề tạm thời từ `teal.edges` (xem như đồ thị không định hướng). Khi BFS, nếu v là láng giềng của u và v chưa được thăm, thì u là cha của v. Thêm v vào danh sách con của u.

Giải thích code:

- **Các lớp biểu diễn:**
  - `ParentArray`: Chứa `n` (số đỉnh), `parents` (mảng cha), và `root_node` (đỉnh gốc).
  - `FCNS`: Chứa `n`, `first_child` (mảng con đầu tiên), và `next_sibling` (mảng anh em kế tiếp).
  - `TreeExtendedAdjacencyList`: Chứa `n`, `m` (số cạnh), `incoming` (danh sách chỉ số cạnh vào), `outgoing` (danh sách chỉ số cạnh ra), và `edges` (danh sách các cạnh).
  - `TreeChildrenList`: Lớp nội bộ chứa `n`, `children` (danh sách con), và `root_node`.
- **Biến toàn cục:** `current_parent_array`, `current_fcns`, `current_tree_eal`, `current_children_list_internal`, `current_tree_rep` lưu trữ trạng thái hiện tại của cây và kiểu biểu diễn.
- **Đọc đầu vào (`main` function):**
  - Người dùng nhập số đỉnh n.
  - Với mỗi đỉnh i, người dùng nhập số lượng con và danh sách các con.
  - Chương trình tính `in_degree` (bậc vào) của mỗi đỉnh để xác định gốc (đỉnh có bậc vào bằng 0).
  - Có kiểm tra tính hợp lệ của đầu vào (đỉnh con nằm trong khoảng, không có khuyên).
  - Các danh sách con (`current_children_list_internal.children`) được sắp xếp ngay sau khi nhập để đảm bảo tính nhất quán cho các chuyển đổi sau này.
- **Tìm gốc:** Duyệt qua mảng `in_degree`. Đỉnh nào có `in_degree` là 0 sẽ là gốc. Chương trình kiểm tra đảm bảo chỉ có đúng một gốc (nếu $n > 0$).
- **Menu tương tác:** Cung cấp các lựa chọn cho người dùng để hiển thị cây hiện tại hoặc thực hiện các chuyển đổi giữa các dạng biểu diễn.
- **Kiểm tra chuyển đổi hợp lệ (`is_valid_tree_conversion`):** Đảm bảo người dùng chỉ có thể thực hiện chuyển đổi từ dạng biểu diễn hiện tại.
- **Hàm hiển thị (`display_...`):** In ra thông tin chi tiết của từng dạng biểu diễn.

Ý nghĩa các biến chính:

- `ParentArray.parents[v]`: Chỉ số của đỉnh cha của v.
- `ParentArray.root_node`: Chỉ số của đỉnh gốc.
- `FCNS.first_child[v]`: Chỉ số của con đầu tiên của v.
- `FCNS.next_sibling[v]`: Chỉ số của anh em kế tiếp của v.
- `TreeExtendedAdjacencyList.edges`: Danh sách các tuple (u, v) đại diện cho các cạnh.
- `TreeExtendedAdjacencyList.outgoing[u]`: Danh sách các chỉ số cạnh trong `edges` bắt đầu từ u.
- `TreeExtendedAdjacencyList.incoming[v]`: Danh sách các chỉ số cạnh trong `edges` kết thúc tại v.
- `TreeChildrenList.children[u]`: Danh sách các con trực tiếp của u.
- `in_degree[v]`: Số lượng cạnh đi vào đỉnh v.

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 $n = 2$.**

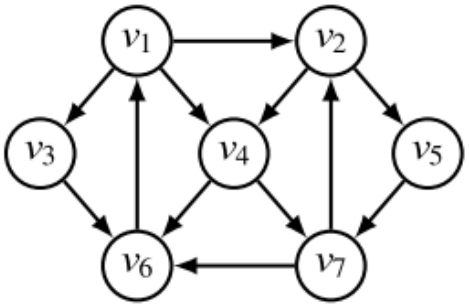
Problem 1.3

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.

Exercise 1.3
Implement algorithms to generate the path graph P_n, the circle graph C_n, and the wheel graph W_n on n vertices, using the collection of 32 abstract operations from Sect. 1.3.
Definitions

- A **path graph** P_n on n vertices is a graph with n vertices and $n-1$ edges, forming a single path.
- A **circle graph** C_n on n vertices is a graph with n vertices and n edges, forming a single cycle. C_n is defined for $n \geq 3$.
- A **wheel graph** W_n on n vertices is formed by connecting a single "hub"vertex to all vertices of a cycle graph C_{n-1}. Thus, W_n has n vertices and $2(n-1)$ edges. W_n is defined for $n \geq 4$.

Solution
We will provide algorithms for generating each of the specified graph types. These algorithms assume the availability of a 'Graph' object with the abstract operations listed in the problem description.

- **Algorithm to Generate Path Graph P_n**

  A path graph P_n is constructed by sequentially adding n vertices and then connecting each vertex to its successor.

- **Procedure** GeneratePathGraph(n)
- $G \leftarrow$ new Graph()
- **If** $n \leq 1$ **Then**
- **Return** G
- **End If**
- $V \leftarrow []$ (List to store created vertices)
- **For** $i \leftarrow 0$ **to** $n - 1$ **Do**
- $v \leftarrow G$.new_vertex() (Create a new vertex)
- V.append(v) (Add vertex to our list)
- **End For**
- **For** $i \leftarrow 0$ **to** $n - 2$ **Do**
- G.new_edge($V[i], V[i + 1]$) (Add an edge between consecutive vertices)
- **End For**
- **Return** G
- **End Procedure**

- **Algorithm to Generate Circle Graph** C_n

  A circle graph C_n builds upon the path graph structure by adding one additional edge to connect the last vertex back to the first, thereby completing the cycle. This is valid for $n \geq 3$.

- **Procedure** GenerateCircleGraph(n)
- $G \leftarrow$ new Graph()
- **If** $n < 3$ **Then**
- **Return** G
- **End If**
- $V \leftarrow []$ (List to store created vertices)
- **For** $i \leftarrow 0$ **to** $n - 1$ **Do**
- $v \leftarrow G$.new_vertex()
- V.append(v)
- **End For**
- **For** $i \leftarrow 0$ **to** $n - 2$ **Do**
- G.new_edge($V[i], V[i + 1]$) (Add edges to form a path)
- **End For**
- G.new_edge($V[n - 1], V[0]$) (Add the closing edge to complete the cycle)
- **Return** G
- **End Procedure**

- **Algorithm to Generate Wheel Graph** W_n

  A wheel graph W_n is formed by creating a central "hub"vertex and a cycle graph C_{n-1} with the remaining $n - 1$ vertices. The hub vertex is then connected to every vertex in the cycle. This is valid for $n \geq 4$.

- **Procedure** GenerateWheelGraph(n)
- $G \leftarrow$ new Graph()
- **If** $n < 4$ **Then**
- **Return** G
- **End If**
- $v_h \leftarrow G$.new_vertex() (Create the central hub vertex)
- $V_{cycle} \leftarrow []$ (List to store vertices for the cycle)
- **For** $i \leftarrow 0$ **to** $n - 2$ **Do**
- $v \leftarrow G$.new_vertex() (Create a vertex for the cycle)

- V_{cycle}.append(v)
- **End For**
- (Form the cycle C_{n-1})
- **For** $i \leftarrow 0$ **to** $n-3$ **Do**
- G.new_edge($V_{cycle}[i], V_{cycle}[i+1]$) (Add edges for the path part of the cycle)
- **End For**
- G.new_edge($V_{cycle}[n-2], V_{cycle}[0]$) (Add the closing edge to complete the cycle)
- (Add edges from the hub to all cycle vertices)
- **For** $i \leftarrow 0$ **to** $n-2$ **Do**
- G.new_edge($v_h, V_{cycle}[i]$) (Connect hub to each cycle vertex)
- **End For**
- **Return** G
- **End Procedure**

Exercise 1.4
Implement an algorithm to generate the complete graph K_n on n vertices and the complete bipartite graph $K_{p,q}$ with $p+q$ vertices, using the collection of 32 abstract operations from Sect. 1.3.
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
We will provide algorithms for generating each of the specified graph types. These algorithms assume the availability of a 'Graph' object with the abstract operations listed in the problem description.

- **Algorithm to Generate Complete Graph K_n**

  A complete graph K_n is formed by creating n vertices and then adding an edge between every unique pair of these vertices.

  - **Procedure** GenerateCompleteGraph(n)
  - $G \leftarrow$ new Graph()
  - **If** $n \leq 0$ **Then**
  - **Return** G
  - **End If**
  - $V \leftarrow []$ (List to store created vertices)
  - **For** $i \leftarrow 0$ **to** $n-1$ **Do**
  - $v \leftarrow G$.new_vertex() (Create a new vertex)
  - V.append(v) (Add vertex to our list)
  - **End For**
  - (Add an edge between every distinct pair of vertices)
  - **For** $i \leftarrow 0$ **to** $n-1$ **Do**
  - **For** $j \leftarrow i+1$ **to** $n-1$ **Do**
  - G.new_edge($V[i], V[j]$)
  - **End For**
  - **End For**
  - **Return** G
  - **End Procedure**

- **Algorithm to Generate Complete Bipartite Graph $K_{p,q}$**

  A complete bipartite graph $K_{p,q}$ is constructed by creating two disjoint sets of vertices, one with p vertices and another with q vertices. Then, an edge is added from every vertex in the first set to every vertex in the second set.

  - **Procedure** GenerateCompleteBipartiteGraph(p, q)
  - $G \leftarrow$ new Graph()
  - **If** $p < 0$ **or** $q < 0$ **Then**
  - **Return** G (Invalid input for p or q)
  - **End If**
  - $V_P \leftarrow []$ (List to store vertices of the first set (size p))
  - $V_Q \leftarrow []$ (List to store vertices of the second set (size q))
  - (Create p vertices for the first set)
  - **For** $i \leftarrow 0$ **to** $p - 1$ **Do**
  - $v \leftarrow G$.new_vertex()
  - V_P.append(v)
  - **End For**
  - (Create q vertices for the second set)
  - **For** $i \leftarrow 0$ **to** $q - 1$ **Do**
  - $v \leftarrow G$.new_vertex()
  - V_Q.append(v)
  - **End For**
  - **For each** u **in** V_P **Do**
  - **For each** v **in** V_Q **Do**
  - G.new_edge(u, v)
  - **End For**
  - **End For**
  - **Return** G
  - **End Procedure**

Exercise 1.5

Implement the extended adjacency matrix graph representation given in Problem 1.4, wrapped in a Python class, using Python lists together with the internal numbering of the vertices.

Solution

We will implement a Python class 'GraphAdjacencyMatrix' that represents a graph using an adjacency matrix. The vertices will be internally numbered from 0 up to $n-1$. The abstract operations will be implemented, with those from Problem 1.4 adapted to use source and target vertices directly instead of edge objects.

The adjacency matrix will be a list of lists (a 2D list) where `self.adj_matrix[u][v] = 1` if an edge exists from vertex `u` to vertex `v`, and `0` otherwise.

- **Class 'GraphAdjacencyMatrix' Implementation**

  Core Data Structure:

  - `self.adj_matrix`: A list of lists representing the adjacency matrix.
  - `self.num_vertices`: An integer tracking the current number of vertices.
  - `self.num_edges`: An integer tracking the current number of edges.
  - `self.vertex_map`: A list to store the actual vertex objects (or simply their internal integer IDs if vertices are just numbers). For this implementation, we'll assume vertices are represented by their integer indices.

  Abstract Operations Implementation:

  - **Initialization:**

    - `__init__(self)`: Initializes an empty graph with an empty adjacency matrix, zero vertices, and zero edges.
- **Vertex and Edge Creation/Deletion:**
    - `new_vertex(self)`: Adds a new vertex to the graph. Expands the adjacency matrix.
    - `new_edge(self, u, v)`: Inserts a new edge from vertex `u` to vertex `v`. Updates `self.adj_matrix[u][v]` to `1`.
    - `del_vertex(self, v)`: Deletes vertex `v` and all incident edges. This involves removing the corresponding row and column from the adjacency matrix and re-indexing subsequent vertices. This is a complex operation for an adjacency matrix and typically inefficient. For simplicity in this context, we will mark vertices as "inactive"rather than physically resizing the matrix for deletion, or raise an error if re-indexing is not explicitly required. A true "delete and re-index"would require a new matrix. A simplified `del_vertex` might just set all incoming/outgoing edges to `0` and conceptually remove the vertex.
    - `del_edge(self, u, v)`: Deletes the edge from vertex `u` to vertex `v`. Sets `self.adj_matrix[u][v]` to `0`.
- **Graph Information:**
    - `number_of_vertices(self)`: Returns `self.num_vertices`.
    - `number_of_edges(self)`: Returns `self.num_edges`.
    - `vertices(self)`: Returns a list of all active vertex indices (0 to `num_vertices-1`).
    - `edges(self)`: Returns a list of all edges as `(source, target)` tuples where `self.adj_matrix[source][targe == 1`.
    - `adjacent(self, u, v)`: Returns `True` if `self.adj_matrix[u][v] == 1`, `False` otherwise.
- **Vertex-Specific Information (Extended for Problem 1.4):**
    - `incoming(self, v)`: Returns a list of `source` vertices such that `self.adj_matrix[source][v] == 1`.
    - `outgoing(self, v)`: Returns a list of `target` vertices such that `self.adj_matrix[v][target] == 1`.
    - `indeg(self, v)`: Returns the count of incoming edges to `v`.
    - `outdeg(self, v)`: Returns the count of outgoing edges from `v`.
- **Traversal Operations (Simulated):**
    - `first_vertex(self)`: Returns the smallest active vertex index.
    - `last_vertex(self)`: Returns the largest active vertex index.
    - `pred_vertex(self, v)`: Returns `v-1` if $v > 0$.
    - `succ_vertex(self, v)`: Returns `v+1` if $v <$ num_vertices $- 1$.
    - `first_edge(self)`: Returns the first `(u,v)` edge found by iterating through the matrix.
    - `last_edge(self)`: Returns the last `(u,v)` edge found.
    - `pred_edge(self, edge)`: Given `(u,v)`, returns the "previous"edge in row-major order. (Complex for direct adjacency matrix, will be simplified to finding the previous edge by iterating).
    - `succ_edge(self, edge)`: Given `(u,v)`, returns the "next"edge. (Complex for direct adjacency matrix, will be simplified).
    - `first_in_edge(self, v)`: Returns the first `(u,v)` edge incoming to `v`.
    - `last_in_edge(self, v)`: Returns the last `(u,v)` edge incoming to `v`.
    - `in_pred(self, edge)`: Returns the previous incoming edge for `edge[1]`.
    - `in_succ(self, edge)`: Returns the next incoming edge for `edge[1]`.
    - `first_adj_edge(self, v)`: Returns the first `(v,w)` edge outgoing from `v`.
    - `last_adj_edge(self, v)`: Returns the last `(v,w)` edge outgoing from `v`.
    - `adj_pred(self, edge)`: Returns the previous outgoing edge for `edge[0]`.
    - `adj_succ(self, edge)`: Returns the next outgoing edge for `edge[0]`.
- **Edge Source/Target (Problem 1.4 Specific):**
    - `source(self, edge)`: Returns `edge[0]`.
    - `target(self, edge)`: Returns `edge[1]`.
    - `opposite(self, v, edge)`: Returns `edge[1]` if `v == edge[0]`, else `edge[0]`.

Exercise 1.6
Enumerate all perfect matchings in the complete bipartite graph $K_{p,q}$ on $p+q$ vertices.
Definitions

- A **bipartite graph** is a graph whose vertices can be divided into two disjoint and independent sets, L and R, such that every edge connects a vertex in L to one in R.
- A **complete bipartite graph** $K_{p,q}$ is a bipartite graph where the two sets of vertices have sizes p and q respectively, and every vertex in the first set is connected to every vertex in the second set.
- A **matching** in a graph is a set of edges such that no two edges share a common vertex.
- A **perfect matching** is a matching that covers all vertices in the graph. In a bipartite graph with bipartition (L, R), a perfect matching exists only if $|L| = |R|$. If it exists, it will contain $|L|$ (or $|R|$) edges.

Solution
For a perfect matching to exist in a complete bipartite graph $K_{p,q}$, the number of vertices in both partitions must be equal, i.e., $p = q$. If $p \neq q$, there are no perfect matchings.

When $p = q = n$, the graph is $K_{n,n}$. A perfect matching in $K_{n,n}$ consists of n edges, where each vertex from the first partition is paired with a unique vertex from the second partition. This problem is equivalent to finding all possible bijections (or permutations) between the vertices of the two partitions. The number of such matchings is $n!$.

We can enumerate these matchings using a recursive backtracking algorithm. Let the vertices in the first partition be $L = \{0, 1, \ldots, n-1\}$ and the vertices in the second partition be $R = \{n, n+1, \ldots, 2n-1\}$.

- **Algorithm to Enumerate Perfect Matchings in $K_{p,q}$**
  - **Procedure** EnumeratePerfectMatchingsKpq(p, q)
  - **If** $p \neq q$ **Then**
  - **Return** empty list (No perfect matching exists)
  - **End If**
  - $n \leftarrow p$
  - *all_matchings* $\leftarrow$ [] (List to store all found perfect matchings)
  - *current_matching* $\leftarrow$ [] (Stores edges for the current matching being built)
  - *used_right_vertices* $\leftarrow$ list of n False values (Boolean array to track used vertices in R)
  - **Function** FindMatchingsRecursive(*l_idx*)
  - **If** *l_idx* $= n$ **Then**
  - Add a copy of *current_matching* to *all_matchings*
  - **Return**
  - **End If**
  - (Iterate through all vertices in R to find a match for $L[$*l_idx*$]$)
  - **For** *r_relative_idx* $\leftarrow 0$ **to** $n-1$ **Do**
  - **If not** *used_right_vertices*[*r_relative_idx*] **Then**
  - *used_right_vertices*[*r_relative_idx*] $\leftarrow$ True
  - *edge* $\leftarrow$ (*l_idx*, $n +$ *r_relative_idx*) (Form edge using absolute vertex IDs)
  - Add *edge* to *current_matching*
  - Call FindMatchingsRecursive(*l_idx* $+ 1$)
  - (Backtrack: undo the choice for the current $L[$*l_idx*$]$)
  - Remove last edge from *current_matching*
  - *used_right_vertices*[*r_relative_idx*] $\leftarrow$ False
  - **End If**
  - **End For**
  - **End Function**
  - Call FindMatchingsRecursive(0) (Start matching from the first vertex in L)

– **Return** *all_matchings*

Exercise 1.7
Implement an algorithm to generate the complete binary tree with n nodes, using the collection of 13 abstract operations from Sect. 1.3.
Definitions

- A **binary tree** is a tree in which each node has at most two children, typically referred to as the left child and the right child.
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Assumptions for Abstract Operations (Sect. 1.3 adapted for Tree Generation):
To generate a tree using these abstract operations, we assume the existence of a `Tree` object with methods to create and link nodes. Since the provided operations are primarily for querying tree structure, we must infer or define basic node creation and parent-child linking for the purpose of *building* the tree. We will assume the following simplified core operations are available for generation:

- `T.new_node()`: Creates a new, unattached node and returns its identifier.
- `T.set_root(node)`: Sets the given node as the root of the tree.
- `T.add_child(parent_node, child_node, position)`: Adds `child_node` as a child of `parent_node`. `position` could be 'left' or 'right' for binary trees, or an index for ordered children. For complete binary trees, we primarily care about adding the *next available* child.
- `T.number_of_nodes()`: As given.
- `T.root()`: As given.
- `T.number_of_children(v)`: As given.

The operations like `T.parent(v)` and `T.children(v)` implicitly define the tree structure after `add_child` is used.
Algorithm to Generate a Complete Binary Tree with n Nodes
A complete binary tree can be constructed level by level using a breadth-first approach. We will use a queue to keep track of parent nodes that still have available "slots"for children (i.e., less than two children).

- **Procedure** GenerateCompleteBinaryTree(n)
- Initialize an empty Tree object, T.
- **If** $n \leq 0$ **Then**
- Return T
- **End If**
- $root\_node \leftarrow T$.new_node()
- T.set_root($root\_node$)
- $nodes\_created \leftarrow 1$
- $queue \leftarrow$ empty Queue
- $queue$.enqueue($root\_node$)
- **While** $nodes\_created < n$ **Do**
- $current\_parent \leftarrow queue$.dequeue()
- Add left child if needed
- **If** $nodes\_created < n$ **Then**
- $left\_child \leftarrow T$.new_node()
- T.add_child($current\_parent, left\_child$, 'left')

- $nodes\_created \leftarrow nodes\_created + 1$
- $queue$.enqueue($left\_child$)
- **End If**
- Add right child if needed
- **If** $nodes\_created < n$ **Then**
- $right\_child \leftarrow T$.new_node()
- T.add_child($current\_parent, right\_child$, 'right')
- $nodes\_created \leftarrow nodes\_created + 1$
- $queue$.enqueue($right\_child$)
- **End If**
- **End While**
- **Return** T
- **End Procedure**

Exercise 1.8

Implement an algorithm to generate random trees with n nodes, using the collection of 13 abstract operations from Sect. 1.3. Give the time and space complexity of the algorithm.

Definitions

- A **tree** is an undirected graph in which any two vertices are connected by exactly one path, or equivalently, a connected acyclic graph. In the context of the abstract operations, it implies a rooted tree with parent-child relationships.
- A **random tree** refers to a tree generated through a stochastic process. For this exercise, we will implement a common method: generating a random tree by iteratively adding nodes and attaching each new node to a randomly selected existing node in the growing tree. This process ensures the resulting structure is always a tree.

Assumptions for Abstract Operations (Sect. 1.3 adapted for Tree Generation):

The 13 abstract operations provided in Section 1.3 are primarily for querying properties of an existing tree structure. To generate a tree, we must assume the availability of fundamental construction operations. We will assume the following simplified core operations for building the tree:

- `T.new_node()`: Creates a new, unattached node and returns its unique identifier. This operation is assumed to handle internal node instantiation.
- `T.set_root(node)`: Sets the given `node` as the root of the tree `T`. This operation should only be called once for an empty tree.
- `T.add_child(parent_node, child_node)`: Establishes a parent-child relationship where `child_node` becomes a child of `parent_node`. This operation implicitly updates the tree's internal representation such that `T.parent(child_node)` returns `parent_node` and `T.children(parent_node)` includes `child_node`.

The other 13 operations from Section 1.3 are available for querying the tree structure but are not directly used in the construction logic, other than the implicit effect of `add_child` on them.

Algorithm to Generate a Random Tree with n Nodes

The algorithm will start with a single root node and then iteratively add $n - 1$ new nodes. Each new node will be connected as a child to a randomly chosen existing node in the tree. This ensures that the graph remains connected and acyclic throughout the process, thus forming a tree.

- **Procedure** GenerateRandomTree(n)
- Initialize an empty Tree object, T.
- Initialize an empty list, $existing\_nodes\_in\_T$.
- **If** $n \leq 0$ **Then**

- Return T
- **End If**
- $root\_node \leftarrow T$.new_node()
- T.set_root($root\_node$)
- Add $root\_node$ to $existing\_nodes\_in\_T$.
- **For** $i \leftarrow 2$ **to** n **Do**
- $new\_node \leftarrow T$.new_node()
- Randomly select $parent\_candidate$ from $existing\_nodes\_in\_T$.
- T.add_child($parent\_candidate, new\_node$)
- Add new_node to $existing\_nodes\_in\_T$.
- **End For**
- **Return** T
- **End Procedure**

Time and Space Complexity Analysis
Let n be the number of nodes in the tree to be generated.

- **Time Complexity:**
  - **Initialization:** Creating an empty tree and an empty list takes constant time, $O(1)$.
  - **Root Creation:** `T.new_node()` and `T.set_root()` are assumed to be $O(1)$ operations. Adding the root to $existing\_nodes\_in\_T$ (a list) is $O(1)$.
  - **Loop for $n-1$ nodes:** The loop runs $n-1$ times.
    - `T.new_node()`: Each call is $O(1)$. Total: $O(n)$.
    - **Randomly select parent:** If $existing\_nodes\_in\_T$ is stored in a dynamic array (like a Python list or C++ `std::vector`), selecting a random element by index is $O(1)$. Total: $O(n)$.
    - `T.add_child(parent_candidate, new_node)`: This operation's complexity depends on the underlying representation of children.
      - If children are stored in an unordered list/array within each parent, adding a child is typically $O(1)$ amortized (for dynamic arrays) or $O(1)$ (for linked lists).
      - If children are ordered and insertion requires shifting, it could be $O(k)$ where k is the number of children, worst case $O(n)$. However, for general random tree generation, we usually assume a simple append.

      Assuming an efficient $O(1)$ amortized `add_child` operation: Total: $O(n)$.
    - **Add `new_node` to `existing_nodes_in_T`:** Appending to a dynamic array is $O(1)$ amortized. Total: $O(n)$.
  - **Overall Time Complexity:** Summing these up, the dominant factor is the loop that runs $n-1$ times, with each iteration performing constant time operations. Therefore, the total time complexity is $O(n)$.
- **Space Complexity:**
  - **Tree Representation:** The tree itself will store n nodes and $n-1$ edges. The internal representation (e.g., adjacency lists or parent pointers) will require space proportional to the number of nodes and edges. For an adjacency list representation where each node stores a list of its children, this would be $O(n)$ for nodes and $O(n)$ for edges, totaling $O(n)$.
  - **Auxiliary Storage (`existing_nodes_in_T`):** This list stores references to all nodes created so far. Its size grows linearly with n, reaching n elements. This requires $O(n)$ space.
  - **Overall Space Complexity:** The total space complexity is dominated by the tree structure and the auxiliary list, both of which are linear in n. Therefore, the total space complexity is $O(n)$.

Exercise 1.9

Give an implementation of operation `T.previous_sibling(v)` using the array-of-parents tree representation.

Array-of-Parents Tree Representation

In an array-of-parents tree representation, the tree structure is stored using an array (or list) where each element at index i stores the parent of the node identified by index i.

- Let the nodes be identified by integer indices from 0 to $N-1$, where N is the total number of nodes.
- We define an array, say `parent_array`, of size N.
- For each node v (at index v), `parent_array[v]` stores the index of its parent.
- The root node typically has a special value in its entry, such as -1 or `nil`, to indicate it has no parent.

This representation directly supports the `T.parent(v)` operation by a simple array lookup.

Abstract Operations from Sect. 1.3 relevant for `T.previous_sibling(v)`

To implement `T.previous_sibling(v)` using this representation and the abstract operations, we will primarily rely on:

- `T.parent(v)`: Returns the parent of node `v`. This is directly available from our `parent_array`.
- `T.children(v)`: Returns a list of the children of node `v`. This operation is crucial because siblings are children of the same parent. To implement this using `parent_array`, one would iterate through `parent_array` and collect all nodes whose parent is `v`.
- The understanding that the children returned by `T.children(v)` are in a specific order fixed by the representation (e.g., by their node ID, or insertion order). For `previous_sibling` to be well-defined, this order is essential.

Implementation of `T.previous_sibling(v)`

The operation `T.previous_sibling(v)` returns the sibling node that precedes `v` in the ordered list of children of its parent. If `v` is the root, or if `v` is the first child of its parent, it has no previous sibling, and `nil` (or an equivalent indicator like a special node ID) should be returned.

- **Procedure** T.previous_sibling(v)
- **If** T.is_root(v) **Then**
- Return `nil` (Root has no siblings)
- **End If**
- $p \leftarrow T$.parent(v)
- $children\_of\_p \leftarrow T$.children($p$) (This list must preserve the order of children)
- $v\_index\_in\_children \leftarrow$ index of v in $children\_of\_p$
- **If** $v\_index\_in\_children > 0$ **Then**
- Return $children\_of\_p[v\_index\_in\_children - 1]$
- **Else** (v is the first child)
- Return `nil`
- **End If**
- **End Procedure**

Note on T.children(p) implementation in Array-of-Parents: For the `T.children(p)` operation itself to be efficient with an array-of-parents, it would typically involve iterating through the entire `parent_array` ($O(N)$ time, where N is the total number of nodes) to find all nodes whose parent is p. Alternatively, a more augmented array-of-parents representation might include a list of children for each node, which would effectively make it an adjacency list representation. Assuming `T.children(p)` returns an ordered list, the complexity is dominated by that operation.

Exercise 1.10

Implement the extended first-child, next-sibling tree representation of Problem 1.5, wrapped in a Python class, using Python lists together with the internal numbering of the nodes.

The traditional first-child, next-sibling (FCNS) representation stores for each node:

- A reference to its first child.
- A reference to its next sibling.

To support operations like `T.root()`, `T.number_of_children(v)`, and `T.children(v)` in $\mathcal{O}(1)$ time, Problem 1.5 proposed augmenting this representation. For an implementation using internal node numbering (i.e., integer IDs for nodes), this translates to maintaining several parallel lists (arrays).

Data Structures for the Extended FCNS Representation

We will represent nodes by their integer indices, starting from 0. The tree state will be managed by a collection of Python lists, where the index in each list corresponds to the node ID.

- `_root`: An integer storing the ID of the tree's root node. Initialized to a sentinel value (e.g., -1) for an empty tree.
- `_next_node_id`: An integer that keeps track of the next available unique ID for a new node.
- `_parent`: A list where `_parent[i]` stores the integer ID of the parent of node i. The root's parent is -1.
- `_first_child`: A list where `_first_child[i]` stores the integer ID of the first child of node i. If node i has no children, this is -1.
- `_next_sibling`: A list where `_next_sibling[i]` stores the integer ID of the next sibling of node i. If node i is the last child, this is -1.
- `_previous_sibling`: A list where `_previous_sibling[i]` stores the integer ID of the previous sibling of node i. This is included for $\mathcal{O}(1)$ access for `previous_sibling(v)`. If node i is the first child or has no parent, this is -1.
- `_num_children`: A list where `_num_children[i]` stores the integer count of children of node i. This is updated when children are added or removed.
- `_children_lists`: A list of lists, where `_children_lists[i]` is a Python list containing the integer IDs of all children of node i, in order. This allows for $\mathcal{O}(1)$ access to the full list of children (by returning a reference to it).

All these lists are dynamically sized using Python's list capabilities, expanding as new nodes are added. Unused slots or non-existent relationships are marked with a sentinel value (e.g., -1).

Implementation Details and Abstract Operations

The Python class `ExtendedFCNS_Tree` will encapsulate these data structures and provide methods corresponding to the abstract operations from Section 1.3, along with necessary construction operations (`new_node`, `add_child`, `set_root`).

- **Constructor (`__init__`)**
  - Initializes all internal lists (`_parent`, `_first_child`, `_next_sibling`, `_previous_sibling`, `_num_children`, `_children_lists`) as empty lists.
  - Sets `_root = -1` and `_next_node_id = 0`.
- **Node Creation (`new_node()`)**
  - Assigns the current `_next_node_id` as the new node's ID.
  - Extends all internal lists by one element, initializing the new slots with sentinel values (-1 for IDs, 0 for child count, empty list for `_children_lists`).
  - Increments `_next_node_id`.
  - Returns the new node's ID.
- **Setting Root (`set_root(node_id)`)**
  - Sets `_root = node_id`.
  - The parent of the root node is set to -1.
- **Adding Child (`add_child(parent_id, child_id)`)**
  - Sets the parent pointer for the child: `_parent[child_id] = parent_id`.
  - Increments the parent's child count: `_num_children[parent_id]`.

  - Appends `child_id` to the parent's `_children_lists[parent_id]` (this maintains the order of children).
  - If this is the parent's first child, sets `_first_child[parent_id] = child_id`.
  - Otherwise (if the parent already had children), finds the previously last child in `_children_lists[parent_id]` and updates its `_next_sibling` pointer to the new `child_id`. Also, sets the new `child_id`'s `_previous_sibling` to the previously last child.
  - This ensures all relevant pointers and counts in the extended FCNS are consistent.

- **Abstract Operations Implementation:**
  - `T.number_of_nodes()`: Returns `_next_node_id`. ($\mathcal{O}(1)$)
  - `T.root()`: Returns `_root`. ($\mathcal{O}(1)$)
  - `T.is_root(v)`: Returns `v == _root`. ($\mathcal{O}(1)$)
  - `T.number_of_children(v)`: Returns `_num_children[v]`. ($\mathcal{O}(1)$)
  - `T.parent(v)`: Returns `_parent[v]`. ($\mathcal{O}(1)$)
  - `T.children(v)`: Returns `_children_lists[v]`. ($\mathcal{O}(1)$)
  - `T.is_leaf(v)`: Returns `_num_children[v] == 0`. ($\mathcal{O}(1)$)
  - `T.first_child(v)`: Returns `_first_child[v]`. ($\mathcal{O}(1)$)
  - `T.last_child(v)`: Returns the last element of `_children_lists[v]` (if it exists), or -1 if no children. ($\mathcal{O}(1)$)
  - `T.previous_sibling(v)`: Returns `_previous_sibling[v]`. ($\mathcal{O}(1)$)
  - `T.next_sibling(v)`: Returns `_next_sibling[v]`. ($\mathcal{O}(1)$)
  - `T.is_first_child(v)`: Returns `_parent[v] != -1` and `_first_child[_parent[v]] == v`. ($\mathcal{O}(1)$)
  - `T.is_last_child(v)`: Returns `_next_sibling[v] == -1` and `_parent[v] != -1`. ($\mathcal{O}(1)$)

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 Khoảng cách chỉnh sửa cây (Tree Edit Distance):**
  - Cho hai cây có thứ tự T_1 và T_2.
  - Mục tiêu là tìm một chuỗi các phép biến đổi có chi phí tối thiểu để biến T_1 thành T_2.
  - Các phép biến đổi cơ bản:
    * **Deletion (Xóa):** Xóa một nút khỏi T_1. Chi phí: $DEL\_COST = 1$.
    * **Insertion (Chèn):** Chèn một nút vào T_2 (nút này không có nút tương ứng trong T_1). Chi phí: $INS\_COST = 1$.
    * **Relabeling (Đổi nhãn):** Đổi nhãn của một nút trong T_1 để khớp với nhãn của nút tương ứng trong T_2. Chi phí: $REP\_COST = 1$.

- **Backtracking (Quay lui):**
  - Backtracking là một kỹ thuật thuật toán tổng quát để tìm tất cả các giải pháp cho một vấn đề tính toán bằng cách xây dựng dần các giải pháp và loại bỏ các giải pháp không hợp lệ.
  - Nếu một giải pháp cục bộ không thể được mở rộng thành một giải pháp hợp lệ hoàn chỉnh, thuật toán sẽ "quay lui"(backtrack) và thử một lựa chọn khác.

- **Áp dụng Backtracking cho Tree Edit Distance:**
  - **Ánh xạ (Mapping):** Một giải pháp cho bài toán được định nghĩa bởi một ánh xạ $M \subseteq V_1 \times V_2$ giữa các nút của T_1 và T_2. Mỗi nút $v \in V_1$ được ánh xạ tới một nút $w \in V_2$ hoặc tới một nút "ảo"λ (ký hiệu cho việc xóa nút v).
  - **Các ràng buộc (Constraints):** Ánh xạ M phải thỏa mãn ba ràng buộc chính để đại diện cho một phép biến đổi cây hợp lệ:

1. **Song ánh (Bijection):** Mỗi nút trong T_1 được ánh xạ tới nhiều nhất một nút trong T_2, và mỗi nút trong T_2 được ánh xạ từ nhiều nhất một nút trong T_1.
2. **Bảo toàn quan hệ cha-con (Parent-Child Preservation):** Nếu nút $v \in T_1$ ánh xạ tới $w \in T_2$, và v_c là con của v, thì v_c phải ánh xạ tới w_c là con của w (hoặc v_c bị xóa).
3. **Bảo toàn thứ tự anh em (Sibling Order Preservation):** Nếu nút $v \in T_1$ ánh xạ tới $w \in T_2$, và v_s là anh/chị em đứng bên phải của v, thì v_s phải ánh xạ tới w_s là anh/chị em đứng bên phải của w (hoặc v_s bị xóa).

– **Ý tưởng đệ quy:**
  * Duyệt qua các nút của T_1 theo thứ tự preorder.
  * Tại mỗi nút v của T_1, thử ánh xạ nó tới tất cả các nút $w \in T_2$ hợp lệ (và λ).
  * Sau mỗi lần thử ánh xạ ($v \to w$), cập nhật tập hợp các ứng viên hợp lệ cho các nút còn lại của T_1 dựa trên các ràng buộc trên.
  * Đệ quy gọi hàm cho nút tiếp theo trong preorder.
  * Nếu tất cả các nút của T_1 đã được ánh xạ, một giải pháp hoàn chỉnh được tìm thấy, tính toán chi phí và lưu trữ.
  * Sau khi nhánh đệ quy kết thúc, quay lui để thử các ánh xạ khác.

Giải thích code:

- **Các hằng số và chi phí:**
  – `LAMBDA_NODE = -1`: Biến số nguyên đặc biệt đại diện cho nút "ảo"λ (nút bị xóa). Giá trị này được chọn để không trùng với bất kỳ ID nút hợp lệ nào (ID nút bắt đầu từ 0).
  – `DEL_COST`, `INS_COST`, `REP_COST`: Chi phí cho các phép biến đổi tương ứng.
- **Các lớp và cấu trúc dữ liệu:**
  – `struct TreeNode`: Biểu diễn một nút của cây với các thuộc tính như ID, nhãn, ID cha, danh sách con, độ sâu, và chỉ số thứ tự duyệt preorder.
  – `class Tree`: Biểu diễn một cây có thứ tự.
    * `nodes`: Một `std::map<int, TreeNode>` lưu trữ tất cả các nút của cây, ánh xạ ID nút tới đối tượng `TreeNode`.
    * `root_id`: ID của nút gốc.
    * `add_node(...)`: Thêm một nút mới vào cây.
    * `get_node(...)`: Trả về con trỏ tới nút theo ID (hoặc `nullptr` nếu không tìm thấy).
    * `compute_preorder_and_depth()`: Thực hiện duyệt DFS để tính toán độ sâu và chỉ số preorder cho tất cả các nút, lưu vào `_preorder_traversal_list`.
  – `struct Solution`: Lưu trữ một giải pháp tìm được, bao gồm ánh xạ (`mapping`), tổng chi phí (`cost`), và chi tiết về số lượng xóa, chèn, đổi nhãn (`details`).
- **Các hàm chính:**
  – `calculate_edit_distance(T1, T2, M)`:
    * Tham số: Hai cây T_1, T_2 và một ánh xạ M hoàn chỉnh.
    * Chức năng: Tính toán tổng chi phí (xóa, chèn, đổi nhãn) cho ánh xạ M đã cho.
    * Cách tính:
      · **Xóa:** Duyệt qua M, đếm số nút $v \in T_1$ được ánh xạ tới `LAMBDA_NODE`.
      · **Chèn:** Đếm số nút $w \in T_2$ không được ánh xạ từ bất kỳ nút nào trong T_1.
      · **Đổi nhãn:** Duyệt qua M, đếm số cặp (v, w) nơi $v \in T_1$ ánh xạ tới $w \in T_2$ (không phải λ) nhưng nhãn của v khác nhãn của w.
  – `set_up_candidate_nodes(T1, T2)`:
    * Chức năng: Khởi tạo tập hợp ứng viên C. Ban đầu, với mỗi nút $v \in T_1$, $C[v]$ chứa `LAMBDA_NODE` và tất cả các nút $w \in T_2$ có cùng độ sâu với v.
  – `refine_candidate_nodes(T1, T2, C_copy, v_id, w_id)`:
    * Chức năng: Lọc bỏ các ứng viên không hợp lệ trong *C_copy* dựa trên ánh xạ hiện tại ($v\_id \to w\_id$) và các ràng buộc (song ánh, cha-con, thứ tự anh em). Hàm này thay đổi trực tiếp *C_copy*.
  – `extend_tree_edit(T1, T2, M_current, L_solutions, C_current, current_v_idx)`:

    * Đây là hàm đệ quy cốt lõi của thuật toán backtracking.
    * `M_current`: Ánh xạ cục bộ đang được xây dựng. `L_solutions`: Danh sách lưu trữ tất cả các giải pháp hoàn chỉnh tìm được.
    * `C_current`: Tập hợp các ứng viên cho các nút chưa được ánh xạ (được truyền theo giá trị để tạo bản sao sâu cho mỗi nhánh đệ quy).
    * `current_v_idx`: Chỉ số của nút T_1 hiện tại cần ánh xạ (theo thứ tự preorder).
    * **Trường hợp cơ sở:** Nếu tất cả các nút của T_1 đã được ánh xạ (`current_v_idx` bằng kích thước của danh sách preorder), một `Solution` mới được tạo, chi phí được tính toán và thêm vào `L_solutions`.
    * **Bước đệ quy:** Chọn nút v ở `current_v_idx`. Với mỗi ứng viên w trong $C\_current[v]$:
        · Ánh xạ $v \to w$ vào `M_current`.
        · Tạo bản sao `C_next` từ `C_current` và tinh chỉnh nó bằng `refine_candidate_nodes`.
        · Gọi đệ quy cho nút tiếp theo (`current_v_idx + 1`) với `C_next` đã tinh chỉnh.
        · Khi hàm đệ quy trả về, ánh xạ $v \to w$ được "hủy"(do các lựa chọn tiếp theo sẽ ghi đè lên `M_current[v_id]`).
- `backtracking_tree_edit(T1, T2)`:
    * Hàm chính, khởi tạo các cấu trúc dữ liệu cần thiết (`M`, `L`, `C`) và gọi `extend_tree_edit` để bắt đầu quá trình backtracking.
    * Trả về danh sách tất cả các giải pháp tìm được.

(b) Thuật toán Branch-and-Bound
Giải thích và công thức:

- **Branch-and-Bound (Nhánh-cận):**
    - Branch-and-Bound là một kỹ thuật tối ưu hóa được sử dụng để tìm lời giải tối ưu cho các bài toán tối ưu hóa tổ hợp (combinatorial optimization problems).
    - Nó là một mở rộng của thuật toán Backtracking, nơi mà việc tìm kiếm được tổ chức như một quá trình duyệt cây trạng thái.
    - **Ý tưởng chính:**
        * **Branching (Phân nhánh):** Chia bài toán thành các bài toán con (tạo ra các nhánh trong cây tìm kiếm).
        * **Bounding (Cận):** Tại mỗi nút của cây tìm kiếm, tính toán một "cận dưới"(lower bound) cho chi phí của bất kỳ lời giải nào có thể được xây dựng từ bài toán con tại nút đó.
        * **Pruning (Cắt tỉa):** Nếu cận dưới của một nhánh (bài toán con) lớn hơn hoặc bằng chi phí của lời giải tốt nhất đã tìm thấy (cận trên toàn cục), thì nhánh đó và tất cả các bài toán con của nó sẽ bị loại bỏ khỏi quá trình tìm kiếm. Điều này giúp giảm đáng kể không gian tìm kiếm.
- **Áp dụng Branch-and-Bound cho Tree Edit Distance:**
    - Thuật toán Branch-and-Bound được xây dựng dựa trên khung sườn của thuật toán Backtracking đã giải thích ở phần (a).
    - Mục tiêu là tìm lời giải có chi phí tối thiểu, thay vì tìm tất cả các lời giải hoặc chỉ một lời giải bất kỳ.
    - **Cận dưới (`lower_bound_cost`):** Trong quá trình xây dựng ánh xạ từng phần (`M_current`), ta tính toán chi phí hiện tại của các phép biến đổi đã được "cam kết"(xóa và đổi nhãn) trong ánh xạ đó. Chi phí này chắc chắn là một cận dưới cho chi phí cuối cùng của bất kỳ lời giải hoàn chỉnh nào được mở rộng từ ánh xạ từng phần này, vì chi phí các phép biến đổi không bao giờ âm.
    - **Cận trên toàn cục (`best_solution.cost`):** Biến này lưu giữ chi phí tối thiểu của lời giải tốt nhất đã tìm thấy cho đến thời điểm hiện tại trong quá trình duyệt. Nó được khởi tạo bằng vô cùng lớn (`std::numeric_limits<int>::max()`) và được cập nhật mỗi khi một lời giải hoàn chỉnh mới tốt hơn được tìm thấy.
    - **Quy tắc cắt tỉa (Pruning Rule):** Tại mỗi bước đệ quy, trước khi đi sâu vào nhánh con:
        * Tính toán `lower_bound_cost` cho ánh xạ từng phần `M_current`.
        * Nếu `lower_bound_cost >= best_solution.cost`, nhánh hiện tại sẽ bị cắt tỉa. Điều này có nghĩa là ngay cả với chi phí tối thiểu tuyệt đối trong tương lai, nhánh này cũng không thể tạo ra một lời giải tốt hơn lời giải tốt nhất đã biết, do đó không cần khám phá thêm.

Giải thích code:

- **Các biến số và cấu trúc dữ liệu quan trọng:**
  - `LAMBDA_NODE`, `DEL_COST`, `INS_COST`, `REP_COST`: Các hằng số định nghĩa nút ảo và chi phí các phép biến đổi.
  - `TreeNode`: Lớp đại diện cho một nút trong cây, với các thông tin như ID, nhãn, cha, con, độ sâu, và chỉ số preorder.
  - `Tree`: Lớp đại diện cho cây, quản lý các nút và các hàm tiền xử lý (tính preorder, độ sâu).
  - `SolutionState`: Một cấu trúc dữ liệu để lưu trữ thông tin của lời giải tốt nhất tìm được:
    - `cost`: Chi phí tối thiểu của lời giải.
    - `mapping`: Ánh xạ tương ứng với lời giải có chi phí tối thiểu đó.
    - `details`: Chi tiết số lượng các phép xóa, chèn, đổi nhãn.
- **Các hàm chính:**
  - `calculate_partial_cost(T1, T2, M_current)`:
    - **Chức năng:** Tính toán chi phí của một ánh xạ từng phần (`M_current`). Hàm này chỉ tính tổng chi phí của các phép xóa (khi một nút T1 được ánh xạ tới `LAMBDA_NODE`) và các phép đổi nhãn (khi một nút T1 được ánh xạ tới một nút T2 nhưng nhãn khác nhau).
    - **Vai trò trong Branch-and-Bound:** Kết quả của hàm này chính là `lower_bound_cost`, được sử dụng trong bước cắt tỉa.
  - `calculate_edit_distance(T1, T2, M)`:
    - **Chức năng:** Tính toán tổng chi phí chỉnh sửa cho một ánh xạ **hoàn chỉnh** `M`. Hàm này bao gồm cả chi phí xóa, chèn và đổi nhãn.
    - **Vai trò:** Được gọi khi một lời giải hoàn chỉnh được tìm thấy (trường hợp cơ sở của đệ quy) để xác định chi phí thực tế của nó.
  - `set_up_candidate_nodes(T1, T2)`:
    - **Chức năng:** Khởi tạo tập hợp các nút ứng viên (`C`) cho mỗi nút trong T_1. Ban đầu, một nút $v \in T_1$ có thể ánh xạ tới `LAMBDA_NODE` hoặc bất kỳ nút nào $w \in T_2$ có cùng độ sâu với v.
  - `refine_candidate_nodes(T1, T2, C_copy, v_id, w_id)`:
    - **Chức năng:** Lọc bỏ các ứng viên không hợp lệ từ tập `C_copy` dựa trên ánh xạ mới được thêm vào (`v_id -> w_id`) và ba ràng buộc chính (song ánh, bảo toàn cha-con, bảo toàn thứ tự anh em).
    - **Lưu ý:** `C_copy` được truyền dưới dạng tham chiếu để cho phép hàm này sửa đổi tập ứng viên cho các nút chưa được ánh xạ.
  - `branch_and_bound_extend_tree_edit(T1, T2, M_current, C_current, current_v_idx, best_solution)`:
    - Đây là hàm đệ quy cốt lõi của thuật toán Branch-and-Bound.
    - `M_current`: Ánh xạ từng phần đang được xây dựng.
    - `C_current`: Tập hợp các ứng viên hợp lệ cho các nút chưa được ánh xạ. Được truyền theo giá trị để đảm bảo mỗi nhánh đệ quy làm việc trên một bản sao độc lập.
    - `current_v_idx`: Chỉ số của nút T_1 hiện tại cần ánh xạ (theo thứ tự duyệt preorder).
    - `best_solution`: Một tham chiếu đến cấu trúc `SolutionState` toàn cục, chứa lời giải tốt nhất đã tìm thấy cho đến nay.
    - **Logic:**
      - **Cắt tỉa:** Gọi `calculate_partial_cost` để tính `lower_bound_cost`. Nếu `lower_bound_cost >= best_solution.cost`, hàm sẽ kết thúc ngay lập tức (cắt tỉa nhánh).
      - **Trường hợp cơ sở:** Nếu tất cả các nút của T_1 đã được ánh xạ (`current_v_idx` bằng số lượng nút của T_1), hàm sẽ tính toán chi phí cuối cùng bằng `calculate_edit_distance`. Nếu chi phí này tốt hơn `best_solution.cost`, `best_solution` sẽ được cập nhật.
      - **Bước đệ quy:** Duyệt qua các ứng viên cho nút T_1 hiện tại. Với mỗi ứng viên:
      - Gán ánh xạ tạm thời vào `M_current`.
      - Tạo một bản sao `C_next` từ `C_current` và tinh chỉnh nó bằng `refine_candidate_nodes`.
      - Gọi đệ quy cho nút T_1 tiếp theo (`current_v_idx + 1`) với `C_next` đã tinh chỉnh và `best_solution`.
      - (Implicit Backtracking): Khi đệ quy trở về, ánh xạ hiện tại trong `M_current` có thể bị ghi đè bởi ánh xạ mới trong vòng lặp tiếp theo, hoặc nếu là nhánh cuối cùng, nó giữ nguyên.

- `branch_and_bound_tree_edit(T1, T2)`:
  - **Chức năng:** Hàm chính, khởi tạo các cấu trúc dữ liệu ban đầu (`M`, `C`, `best_solution`) và gọi `branch_and_bound_extend_tree_edit` để bắt đầu quá trình tìm kiếm.
  - **Kết quả:** Trả về cấu trúc `SolutionState` chứa lời giải tối ưu tìm được.

(c) Thuật toán Divide-and-Conquer
Giải thích và công thức:

- **Divide-and-Conquer (Chia để trị) với các phép toán chỉ trên lá:**
  - Divide-and-Conquer là một kỹ thuật thiết kế thuật toán đệ quy, phân tách một bài toán lớn thành các bài toán con tương tự nhỏ hơn, giải quyết các bài toán con này và sau đó kết hợp các lời giải để tạo ra lời giải cho bài toán ban đầu.
  - Trong ngữ cảnh Tree Edit Distance, đặc biệt là với ràng buộc "chỉ thao tác trên lá", thuật toán này kết hợp đệ quy với lập trình động (qua bảng memoization và DP con cho rừng).
  - **Ý tưởng chính:**
    - **Divide (Chia):** Đối với hai cây (hoặc cây con) đang xét, tập trung vào gốc của chúng. Bài toán được chia thành việc so sánh các chuỗi con của chúng (tức là các rừng cây con).
    - **Conquer (Trị):** Các bài toán con (tức là so sánh các cặp cây con hoặc rừng cây con) được giải quyết một cách đệ quy. Kết quả từ các bài toán con này sau đó được kết hợp lại sử dụng một bảng lập trình động (DP) để tính khoảng cách chỉnh sửa cho rừng cây con.
    - **Memoization (Ghi nhớ):** Lưu trữ kết quả của các bài toán con đã giải quyết để tránh tính toán lại, đặc trưng của lập trình động.
    - **Leaves-Only Operations (Thao tác chỉ trên lá):** Đây là một ràng buộc quan trọng. Các phép xóa và chèn đối với các nút không phải là lá được định nghĩa gián tiếp thông qua chi phí của tất cả các nút lá nằm trong cây con của chúng. Chỉ nút lá mới có chi phí xóa/chèn trực tiếp. Phép đổi nhãn áp dụng cho mọi nút.

- **Áp dụng Divide-and-Conquer cho Tree Edit Distance (Constrained):**
  - Thuật toán này tìm khoảng cách chỉnh sửa giữa hai cây có thứ tự.
  - **Các phép toán được xét:**
    - **Đổi nhãn (Relabeling):** Thay đổi nhãn của một nút. Chi phí là `REP_COST`.
    - **Xóa lá (Leaf Deletion):** Xóa một nút lá. Chi phí là `DEL_COST`.
    - **Chèn lá (Leaf Insertion):** Chèn một nút lá. Chi phí là `INS_COST`.
    - **Xóa/Chèn nút không phải lá:** Nếu một nút không phải lá (và cây con của nó) bị xóa hoặc chèn do sự khác biệt về cấu trúc, chi phí của thao tác này được tính bằng tổng chi phí xóa/chèn của **tất cả các nút lá** nằm trong cây con đó.
  - **Logic đệ quy:**
    - Hàm chính `constrained_ted_recursive(node1, node2, T1, T2)` so sánh hai cây con gốc `node1` và `node2`.
    - Chi phí của cặp gốc được tính là `relabel_cost` (nếu nhãn khác nhau) cộng với `ForestTED` của các tập con của chúng.
    - `ForestTED` là một bài toán con được giải quyết bằng một bảng DP ($F[i][j]$) tương tự như thuật toán Wagner-Fischer cho khoảng cách chỉnh sửa chuỗi. $F[i][j]$ là khoảng cách giữa i con đầu tiên của `node1` và j con đầu tiên của `node2`.
    - Mỗi ô $F[i][j]$ xem xét 3 khả năng:
      - Xóa cây con thứ i của u: $F[i-1][j] + \text{CostLeafOp}(u_i, \text{delete})$.
      - Chèn cây con thứ j của v: $F[i][j-1] + \text{CostLeafOp}(v_j, \text{insert})$.
      - Ghép cặp cây con thứ i của u với cây con thứ j của v: $F[i-1][j-1] + \text{constrained\_ted\_recursive}(u_i, v_j)$.
    - **Memoization**: Sử dụng một bảng `memo` (`std::map<std::pair<int, int>, int>`) để lưu trữ kết quả của các lời gọi `constrained_ted_recursive` nhằm tránh tính toán lặp lại.

Giải thích code:

- **Các biến số và cấu trúc dữ liệu quan trọng:**
  - `LAMBDA_NODE`, `DEL_COST`, `INS_COST`, `REP_COST`: Các hằng số định nghĩa nút ảo và chi phí các phép biến đổi.
  - `TreeNode`: Lớp đại diện cho một nút trong cây, với các thông tin như ID, nhãn, cha, con, độ sâu và chỉ số preorder.
  - `Tree`: Lớp đại diện cho cây, quản lý các nút và các hàm tiền xử lý (tính preorder, độ sâu). Sử dụng `std::unique_ptr` để quản lý bộ nhớ của các nút.
  - `memo`: Một `std::map` toàn cục được dùng làm bảng memoization, lưu trữ kết quả của các lời gọi đệ quy `constrained_ted_recursive` để tối ưu hóa hiệu suất.
- **Các hàm chính:**
  - `get_leaves_in_subtree(TreeNode* node, const Tree tree_obj)`:
    - **Chức năng:** Tìm và trả về danh sách tất cả các nút lá nằm trong cây con có gốc là `node` đã cho.
    - **Vai trò:** Là hàm hỗ trợ để tính toán chi phí xóa/chèn cho các cây con theo ràng buộc "chỉ thao tác trên lá".
  - `get_constrained_subtree_op_cost(TreeNode* node, const Tree tree_obj, const std::string operation_type)`:
    - **Chức năng:** Tính toán chi phí của việc xóa hoặc chèn toàn bộ một cây con. Chi phí này được xác định bằng số lượng nút lá trong cây con đó nhân với chi phí tương ứng (`DEL_COST` hoặc `INS_COST`).
    - **Vai trò:** Thực hiện việc định nghĩa chi phí cho các thao tác trên các nút không phải lá, dựa trên các nút lá con của chúng.
  - `constrained_ted_recursive(TreeNode* node1, TreeNode* node2, const Tree T1, const Tree T2)`:
    - Đây là hàm đệ quy cốt lõi của thuật toán Divide-and-Conquer.
    - **Trường hợp cơ sở:** Xử lý khi một hoặc cả hai nút đầu vào là `nullptr` (biểu thị cây con rỗng).
    - **Chi phí đổi nhãn:** Tính toán chi phí đổi nhãn giữa `node1` và `node2` nếu nhãn của chúng khác nhau.
    - **Bảng DP cho rừng (`forest_dp_table`):** Một ma trận 2D được sử dụng để tính khoảng cách chỉnh sửa giữa các rừng con (danh sách các con của `node1` và `node2`). Đây là nơi các kết quả của các bài toán con được kết hợp.
      - Bảng được khởi tạo với các trường hợp cơ sở (xóa/chèn toàn bộ các con ở đầu).
      - Các ô còn lại được điền bằng cách lấy giá trị nhỏ nhất trong ba tùy chọn: xóa một con, chèn một con, hoặc ghép cặp hai con (sử dụng lời gọi đệ quy).
      - **Lưu ý về `std::min`:** Để đảm bảo khả năng tương thích với nhiều phiên bản trình biên dịch C++ (đặc biệt là trước C++11 không hỗ trợ `std::min` với initializer list), việc tính toán `min` được thực hiện lồng nhau, ví dụ: `std::min(option1, std::min(option2, option3))`.
    - **Memoization:** Kết quả cuối cùng cho cặp (`node1`, `node2`) được lưu trữ trong bản đồ `memo` trước khi hàm trả về.
  - `divide_and_conquer_constrained_tree_edit_distance(Tree& T1, Tree& T2)`:
    - **Chức năng:** Hàm công khai chính để khởi chạy thuật toán. Nó xóa bộ nhớ đệm `memo`, tính toán các siêu dữ liệu cần thiết cho cây (như thứ tự duyệt trước, độ sâu) và sau đó bắt đầu quá trình đệ quy từ gốc của hai cây T_1 và T_2.
    - **Kết quả:** Trả về khoảng cách chỉnh sửa tối thiểu tìm được. Chi tiết về số lượng các phép xóa, chèn và đổi nhãn thường không được cung cấp trực tiếp từ cách triển khai lập trình động này mà không có thêm logic truy vết.

(d) Phương pháp Dynamic Programming
Giải thích và công thức:

- **Dynamic Programming (Quy hoạch động):**
  - Quy hoạch động là một kỹ thuật tối ưu hóa được sử dụng để giải quyết các vấn đề phức tạp bằng cách chia chúng thành các bài toán con đơn giản hơn, giải quyết từng bài toán con một lần và lưu trữ kết quả để tránh tính toán lặp lại.
  - Nó thường được áp dụng cho các bài toán có hai đặc điểm chính:
    * **Cấu trúc con tối ưu (Optimal Substructure):** Giải pháp tối ưu cho vấn đề lớn có thể được xây dựng từ giải pháp tối ưu của các bài toán con.
    * **Các bài toán con trùng lặp (Overlapping Subproblems):** Các bài toán con tương tự được giải quyết nhiều lần.
- **Áp dụng Dynamic Programming cho Tree Edit Distance:**
  - Ý tưởng chính là định nghĩa một hàm khoảng cách $D(T_a, T_b)$ là chi phí tối thiểu để biến đổi cây con T_a (gốc tại nút a) thành cây con T_b (gốc tại nút b).
  - Chúng ta xây dựng lời giải cho các cây con lớn hơn dựa trên lời giải của các cây con nhỏ hơn.
  - Ba phép toán cơ bản (xóa, chèn, đổi nhãn) được xem xét tại mỗi bước:
    * **Đổi nhãn / Khớp nút gốc:** Nếu chúng ta khớp (hoặc đổi nhãn) nút gốc của T_a và T_b, chi phí là chi phí đổi nhãn (0 nếu cùng nhãn, 1 nếu khác nhãn) cộng với khoảng cách chỉnh sửa giữa các rừng con (sequences of children) của T_a và T_b.
    * **Xóa nút gốc:** Xóa nút gốc của T_a. Chi phí là 1 (chi phí xóa nút gốc) cộng với chi phí để biến đổi phần còn lại của T_a thành T_b. (Trong triển khai đệ quy, điều này thường được xử lý khi một cây con trống được so sánh với một cây con không trống).
    * **Chèn nút gốc:** Chèn nút gốc của T_b. Chi phí là 1 (chi phí chèn nút gốc) cộng với chi phí để biến đổi T_a thành phần còn lại của T_b. (Tương tự như xóa nút gốc, xử lý khi một cây con không trống được so sánh với một cây con trống).
  - **Khoảng cách chỉnh sửa rừng (Forest Edit Distance):** Đây là một bài toán con quan trọng. Để tính khoảng cách giữa hai rừng (một chuỗi các cây con), chúng ta thường sử dụng một bảng quy hoạch động 2D tương tự như khoảng cách Levenshtein cho chuỗi. Bảng này tính toán chi phí biến đổi một tiền tố của rừng thứ nhất thành một tiền tố của rừng thứ hai, bằng cách xem xét các thao tác trên các cây con (xóa cây con, chèn cây con, hoặc khớp/đổi nhãn hai cây con).

Giải thích code:

- **Cấu trúc dữ liệu và Hàm trợ giúp:**
  - `Node` và `Tree` classes: Tương tự như phương pháp backtracking, biểu diễn cấu trúc cây và các nút, bao gồm cả việc tính toán thứ tự preorder và độ sâu.
  - `memo`: Một `std::map<std::pair<int, int>, int>` được sử dụng làm bảng ghi nhớ (memoization table). Khóa là một cặp (`node1_id`, `node2_id`), và giá trị là khoảng cách chỉnh sửa đã được tính toán giữa cây con gốc tại `node1_id` và cây con gốc tại `node2_id`. Điều này ngăn chặn việc tính toán lặp lại cho các bài toán con trùng lặp.
- **Hàm chính `tree_edit_distance_dp(Tree& T1, Tree& T2)`:**
  - Đây là điểm bắt đầu của thuật toán. Nó xóa bảng `memo` để đảm bảo tính toán mới và gọi hàm đệ quy `calculate_distance_recursive` với các ID nút gốc của T_1 và T_2.
  - Xử lý các trường hợp đặc biệt khi một hoặc cả hai cây đều trống.
- **Hàm đệ quy `calculate_distance_recursive(Tree& T1, Tree& T2, int n1_id, int n2_id)`:**
  - Đây là trái tim của thuật toán quy hoạch động, tính toán khoảng cách chỉnh sửa giữa cây con gốc tại `n1_id` và cây con gốc tại `n2_id`.

- **Kiểm tra ghi nhớ:** Nếu kết quả cho cặp (`n1_id`, `n2_id`) đã có trong `memo`, trả về giá trị đã lưu.
- **Trường hợp cơ sở:**
  - Nếu cả `node1` và `node2` đều là `nullptr` (tức là cây con rỗng), chi phí là 0.
  - Nếu chỉ `node1` là `nullptr` (cây con T_1 rỗng): Chi phí là tổng số nút trong cây con T_2 (tính bằng BFS) vì tất cả chúng cần được chèn.
  - Nếu chỉ `node2` là `nullptr` (cây con T_2 rỗng): Chi phí là tổng số nút trong cây con T_1 (tính bằng BFS) vì tất cả chúng cần được xóa.
- **Bước đệ quy (Đối với các nút không rỗng):**
  - **Chi phí đổi nhãn gốc (`relabel_cost`):** 0 nếu nhãn của `node1` và `node2` giống nhau, 1 nếu khác nhau.
  - **Tính toán Khoảng cách chỉnh sửa rừng con (`dp_forest`):**
    - Một bảng 2D `dp_forest` được tạo để tính khoảng cách giữa danh sách con của `node1` và danh sách con của `node2`.
    - Các ô trong `dp_forest[x][y]` biểu thị chi phí biến đổi x con đầu tiên của `node1` thành y con đầu tiên của `node2`.
    - Mỗi ô `dp_forest[x][y]` được tính là giá trị nhỏ nhất của ba trường hợp:
      1. Xóa cây con thứ x của `node1` (chi phí 1 + `calculate_distance_recursive(T1, T2, child1_id, -1)`).
      2. Chèn cây con thứ y của `node2` (chi phí 1 + `calculate_distance_recursive(T1, T2, -1, child2_id)`).
      3. Khớp/Đổi nhãn cây con thứ x của `node1` với cây con thứ y của `node2` (chi phí là tổng của `dp_forest[x-1][y-1]` và kết quả đệ quy của `calculate_distance_recursive` cho hai cây con này).
  - **Tổng chi phí cho cặp gốc:** Chi phí cuối cùng cho việc khớp/đổi nhãn các nút gốc là `relabel_cost` cộng với giá trị `dp_forest[m][k]` (tổng chi phí cho việc chuyển đổi rừng con).
  - **Lưu kết quả:** Kết quả cuối cùng được lưu vào bảng `memo`.

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 ($u = v$).