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 17 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
  $$\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_k), \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k > 0, \ \sum_i \lambda_i = n$$
  gọi là tự liên hợp nếu sơ đồ Ferrers khi chuyển vị (transpose) vẫn cho lại chính nó, tức là đối xứng qua đường chéo chính.

- **Đặc trưng:**

  Khi phân hoạch tự liên hợp, ta nhìn lên Ferrers diagram và đánh dấu dãy độ dài mỗi "nhánh chéo" (hook trên đường chéo chính). Gọi a_i là số ô trên đường chéo chính hàng thứ i. Mỗi ô đó kéo ra hai cánh (hàng và cột) cộng thêm chính ô chéo, nên mỗi a_i đóng góp
  $$2\,a_i - 1$$
  ô. Vì có các nhánh chéo từ $i = 1$ đến k, tổng ô bằng
  $$n \ = \ \sum_{i=1}^{k} (2\,a_i - 1).$$
  Ví dụ, với $\boldsymbol{a} = (3,2,1)$, ta có $n = (2 \cdot 3 - 1) + (2 \cdot 2 - 1) + (2 \cdot 1 - 1) = 5 + 3 + 1 = 9$, và Ferrers diagram:

  ```
  ***   <- hàng 1 (3 ô)
  **    <- hàng 2 (2 ô)
  *     <- hàng 3 (1 ô)
  ```

  Đây là hình đối xứng (self-conjugate).

- **Công thức đệ quy:**

  Gọi $p_k^{\text{selfcjg}}(n)$ số phân hoạch tự liên hợp của n có k nhánh chéo (tức k phần). Để xây phân hoạch có k nhánh, ta chọn độ dài nhánh cuối cùng $a_k = i \geq 1$, nó chiếm $2i - 1$ ô, còn lại $n - i(2 \cdot k - 1)$ ô và $k - 1$ nhánh:
  $$p_k^{\text{selfcjg}}(n) = \sum_{i=1}^{\lfloor n/(2k-1) \rfloor} p_{k-1}^{\text{selfcjg}}\big(n - i\,(2k-1)\big),$$
  với khởi tạo $p_0^{\text{selfcjg}}(0) = 1$, còn $p_0^{\text{selfcjg}}(n > 0) = 0$.

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

  Chúng ta sinh đệ quy với tham số $(n, k, maxVal)$, trong đó $maxVal$ ban đầu đặt bằng $\lfloor n/(2k-1) \rfloor$, rồi mỗi bước:

  - Lấy i từ $maxVal$ xuống 1 (đảm bảo thứ tự không tăng).
  - Gán $a_k = i$, giảm $n \leftarrow n - i(2k-1)$, $k \leftarrow k - 1$, cập nhật $maxVal \leftarrow i$.
  - Khi $k = 0$ và $n = 0$, thu được phân hoạch hợp lệ.

  Vì chúng ta chọn i từ lớn xuống nhỏ, nên luôn giữ được $a_1 \geq a_2 \geq \cdots \geq a_k$.

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

  Định lý cổ điển: Số phân hoạch tự liên hợp của n bằng số phân hoạch của n có *số phần lẻ.*

  Chứng minh ngắn: - Mỗi phân hoạch tự liên hợp có Ferrers diagram đối xứng. - Nếu ta cắt Ferrers diagram thành đường chéo chính, hai nửa trên-dưới đối xứng và có cùng số phần. Chuyển đổi đó cho ra một phân hoạch có số phần lẻ (các hàng phía trên chéo). - Ngược lại, từ phân hoạch có lẻ phần, ta ghép hai nửa đối xứng để được self-conjugate.

  Ví dụ với $n = 5$:

```
Self-conjugate:      Odd-parts partition:

 * * *               * * *
 *                   *
 *                   *

 đây là (3,1,1)       đây là 3+1+1 (3 phần lẻ)
```

  Cả hai đều có đúng 3 hàng (3 phần), nên số lượng song hành.

Giải thích code:

- **C++/Python:** Hai chương trình đều dùng đệ quy để sinh các phân hoạch tự liên hợp, kiểm tra tổng và số phần.
- **Các biến quan trọng:**
  - n, k: số cần phân hoạch và số phần.
  - `current`: vector/list lưu dãy a_i hiện tại.
  - `result`: vector/list chứa các phân hoạch hợp lệ.
  - `min_val`: giá trị nhỏ nhất có thể chọn cho phần tiếp theo (đảm bảo không tăng).
- **Đếm số phân hoạch lẻ phần:** Dùng đệ quy sinh tất cả phân hoạch của n, đếm số phân hoạch có số phần lẻ.
- **Quy hoạch động:** Dùng mảng 2 chiều $dp[n][k]$ với $dp[0][0] = 1$, cập nhật theo công thức đệ quy ở trên.

Ví dụ: $n = 7, k = 2$

- Các phân hoạch tự liên hợp: $(3, 2)$ vì $3\times1+2\times3 = 3+6 = 9$ (không hợp lệ), $(2, 1)$ vì $2\times1+1\times3 = 2+3 = 5$ (không hợp lệ), ...

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.

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

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.*

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 ($u = v$).