

# Combinatorics And Graph Theory-Final

Phạm Phước Minh Hiếu - 2201700085

Ngày 22 tháng 7 năm 2025

## Project: Integer Partition – Đề Án: 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  $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_k) \in (\mathbb{N}^*)^k$  có định dạng các dấu chấm được biểu diễn bởi dấu  $*$ .

#### **Ví dụ:**

Cho  $n = 5, k = 2$ . Các phân hoạch của 5 thành đúng 2 phần tử là:

- $(4, 1)$
- $(3, 2)$

Biểu diễn Ferrers và Ferrers chuyển vị:

**Phân hoạch  $(4, 1)$**

**Ferrers:**

```
****
*
```

**Ferrers chuyển vị:**

```
**
*
*
*
```

**Phân hoạch  $(3, 2)$**

**Ferrers:**

```
***  
**
```

**Ferrers chuyển vị:**

```
**  
**  
*
```

**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  $p_k(n)$   
&  $p_{\max}(n, k)$ .

**Theo đề bài:** Cho  $n, k \in \mathbb{N}$ . Đếm:

- $p_k(n)$ : số phân hoạch của  $n$  thành đúng  $k$  số nguyên dương.
- $p_{\max}(n, k)$ : số phân hoạch của  $n$  sao cho phần tử lớn nhất đúng bằng  $k$ .

**Cho một ví dụ với  $n = 5$**

- Các phân hoạch thành đúng  $k = 2$  phần tử

$$(4, 1), (3, 2) \Rightarrow p_2(5) = 2$$

- Các phân hoạch của  $n = 5$  có phần tử lớn nhất là  $k = 2$ :

$$(2, 2, 1), (1, 1, 1, 2) \Rightarrow p_{\max}(5, 2) = 2$$

$$p_2(5) = 2 \quad \text{vs.} \quad p_{\max}(5, 2) = 2$$

**Bảng so sánh  $p_2(n)$  và  $p_{\max}(n, 2)$ :**

$n$	$p_2(n)$	$p_{\max}(n, 2)$
1	0	0
2	1	1
3	1	1
4	2	2
5	2	1
6	3	2
7	3	3
8	4	4
9	4	4
10	5	5

## 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  $p_k^{\text{selfcig}}(n)$ , rồi in ra các phân hoạch đó.
- (b) Đếm số phân hoạch của  $n$  có lẻ phần, rồi so sánh với  $p_k^{\text{selfcig}}(n)$ .
- (c) Thiết lập công thức truy hồi cho  $p_k^{\text{selfcig}}(n)$ , rồi implementation bằng:  
(i) đệ quy. (ii) quy hoạch động.

Cho  $n, k \in \mathbb{N}$ .

- (a) Đếm số phân hoạch tự liên hợp của  $n$  có đúng  $k$  phần, ký hiệu là  $p_k^{\text{selfcig}}(n)$  và liệt kê các phân hoạch đó.

**Ví dụ:** Với  $n = 7$ , các phân hoạch tự liên hợp gồm:

$$(4, 3), \quad (3, 3, 1), \quad (2, 2, 2, 1), \quad (1, 1, 1, 1, 1, 1, 1)$$

Khi lọc theo số phần  $k$ , ta chỉ giữ các phân hoạch có đúng  $k$  phần tử.

- (b) Đếm số phân hoạch của  $n$  có số phần tử là số lẻ. So sánh giá trị đó với  $p_k^{\text{selfcig}}(n)$ . Ký hiệu tổng số phân hoạch có số phần lẻ là  $q(n)$ .

**Định lý:** Tổng số phân hoạch tự liên hợp của  $n$  đúng bằng số phân hoạch có số phần tử lẻ (Euler).

- (c) **Thiết lập công thức truy hồi cho  $p_k^{\text{selfcig}}(n)$**

Gọi  $p_k^{\text{selfcig}}(n)$  là số phân hoạch **tự liên hợp** của  $n$  thành đúng  $k$  phần.

Do một phân hoạch tự liên hợp luôn gồm các số lẻ không tăng và không vượt quá  $k$ , do đối xứng qua đường chéo chính của biểu đồ Ferrers.

Ta có thể định nghĩa truy hồi như sau:

$$p_k^{\text{selfc}jg}(n) = \begin{cases} 1, & \text{nếu } n = 0 \text{ và } k = 0 \\ 0, & \text{nếu } n < 0 \text{ hoặc } k \leq 0 \\ \sum_{\substack{1 \leq m \leq \min(2k-1, n) \\ m \text{ lẻ}}} p_{k-1}^{\text{selfc}jg}(n-m), & \text{ngược lại} \end{cases}$$

→ Mỗi phần thêm vào là một số lẻ  $m$ , ta trừ nó khỏi  $n$  và giảm số phần đi 1.

→ Ràng buộc số lẻ xuất hiện là do cấu trúc đối xứng: mỗi điểm ở trên đường chéo cần đối xứng với một điểm dưới đường chéo, nên số phần tử ở mỗi dòng phải lẻ.

Ngoài ra, ta có thể thiết lập công thức không phụ thuộc vào  $k$ , khi đếm tổng số phân hoạch tự liên hợp (tức là  $p^{\text{selfc}jg}(n) = \sum_k p_k^{\text{selfc}jg}(n)$ ):

$$p^{\text{selfc}jg}(n) = \# \left\{ (\lambda_1, \lambda_2, \dots, \lambda_k) \mid \sum_{i=1}^k \lambda_i = n, \lambda_i \text{ lẻ}, \lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k, \lambda_i \geq i \right\}$$

Trong đó điều kiện  $\lambda_i \geq i$  đảm bảo đối xứng qua đường chéo chính.

**Công thức tạo hàm sinh:** Hàm sinh của số phân hoạch tự liên hợp là:

$$\sum_{n=0}^{\infty} p^{\text{selfc}jg}(n) q^n = \prod_{k=1}^{\infty} (1 + q^{2k-1})$$

Do mỗi phần tử trong phân hoạch tự liên hợp tương ứng với một chiều dài móc câu (hook length) lẻ.

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

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.

## Làm Problems 1.1–1.6 & Exercises 1.1–1.10.

**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.**

**1.1.1. Đồ thị đầy đủ  $K_n$ :** Đồ thị đầy đủ  $K_n$  là đồ thị đơn vô hướng trong đó mỗi cặp hai đỉnh phân biệt đều được nối với nhau bởi đúng một cạnh.

Tổng số cạnh trong  $K_n$  chính là số cách chọn ra 2 đỉnh bất kỳ từ  $n$  đỉnh để nối thành một cạnh:

$$\text{Số cạnh của } K_n = \binom{n}{2} = \frac{n(n-1)}{2}$$

**1.1.2. Đồ thị hai phía đầy đủ  $K_{p,q}$ :** Đồ thị hai phía đầy đủ  $K_{p,q}$  là đồ thị trong đó tập đỉnh được chia thành hai tập rời  $V_1$  và  $V_2$  với kích thước lần lượt là  $p$  và  $q$ , và mỗi đỉnh trong  $V_1$  được nối với mọi đỉnh trong  $V_2$ .

Do đó, tổng số cạnh là:

$$\text{Số cạnh của } K_{p,q} = p \cdot q$$

**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.**

**1.2.1. Đồ thị vòng  $C_n$ :** Đồ thị vòng  $C_n$  là một đồ thị mà các đỉnh được nối thành một vòng khép kín. Một đồ thị là hai phía khi và chỉ khi nó không chứa chu trình lẻ.

Như vậy:  $C_n$  chỉ chứa một chu trình có độ dài đúng bằng  $n$ , nên:

- Nếu  $n$  chẵn thì  $C_n$  không chứa chu trình lẻ  $\Rightarrow$  là đồ thị hai phía.
- Nếu  $n$  lẻ thì  $C_n$  chứa chu trình lẻ  $\Rightarrow$  không là đồ thị hai phía.

$\rightarrow C_n$  là đồ thị hai phía khi và chỉ khi  $n$  là số chẵn:

$C_n \text{ là đồ thị hai phía khi } n \equiv 0 \pmod{2}$

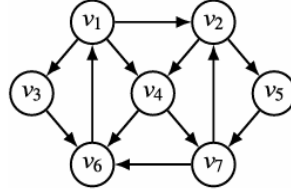
**1.2.2. Đồ thị đầy đủ  $K_n$ :** Đồ thị đầy đủ  $K_n$  là đồ thị mà mỗi cặp hai đỉnh phân biệt đều được nối bởi một cạnh.

Như vậy: một đồ thị hai phía không thể chứa tam giác (chu trình độ dài 3), nhưng  $K_n$  với  $n \geq 3$  luôn chứa tam giác (vì mọi bộ ba đỉnh bất kỳ đều tạo thành một tam giác).

- Với  $n = 1$ :  $K_1$  không có cạnh nào, nên là đồ thị hai phía.
- Với  $n = 2$ :  $K_2$  chỉ có 1 cạnh nối 2 đỉnh, rõ ràng là hai phía.
- Với  $n \geq 3$ :  $K_n$  chứa chu trình lẻ (tam giác), nên không phải hai phía.

Do đó:  $K_n$  là đồ thị hai phía khi và chỉ khi  $n \leq 2$

**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.



**Bước 1: Chuyển sang đồ thị vô hướng**

Ta bỏ hướng của tất cả các cung trong Hình 1.30 để thu được đồ thị vô hướng tương ứng. Tập các cạnh vô hướng là:

$$E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_1, v_6\}, \\ \{v_2, v_4\}, \{v_2, v_5\}, \\ \{v_3, v_1\}, \{v_3, v_6\}, \\ \{v_4, v_6\}, \{v_4, v_7\}, \\ \{v_6, v_7\}\}$$

Số đỉnh:  $n = 7$ , số cạnh:  $m = 10$ .

**Bước 2: Tính số cây khung bằng định lý Kirchhoff**

Bậc của các đỉnh được xác định như sau:

$$\deg(v_1) = 4, \quad \deg(v_2) = 3, \quad \deg(v_3) = 2, \quad \deg(v_4) = 4, \quad \deg(v_5) = 1, \quad \deg(v_6) = 5, \quad \deg(v_7) = 3$$

Từ đó, ma trận bậc:

$$D = \text{diag}(4, 3, 2, 4, 1, 5, 3)$$

Ma trận kề  $A$  tương ứng:

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

**Bước 3: Ma trận Laplace  $L = D - A$**

$$L = \begin{bmatrix} 4 & -1 & -1 & -1 & 0 & -1 & 0 \\ -1 & 3 & 0 & -1 & -1 & 0 & 0 \\ -1 & 0 & 2 & 0 & 0 & -1 & 0 \\ -1 & -1 & 0 & 4 & 0 & -1 & -1 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & -1 & 0 & 5 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 3 \end{bmatrix}$$

**Bước 4: Tính định thức ma trận con  $L'$**

Xóa hàng và cột thứ nhất (ứng với đỉnh  $v_1$ ) trong ma trận Laplace  $L$ , ta thu được ma trận con  $L' \in \mathbb{R}^{6 \times 6}$ . Số cây khung của đồ thị được tính bằng định thức:

$$\tau(G) = \det(L')$$

Tính tay bằng khai triển Laplace hoặc khử Gauss, ta thu được:

$$\tau(G) = 64$$

Như vậy số cây khung của đồ thị vô hướng tương ứng với Hình 1.30 là:

$$64$$

**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:  $G.del\_edge(v, w)$ ,  $G.edges()$ ,  $G.incoming(v)$ ,  $G.outgoing(v)$ ,  $G.source(v, w)$ , and  $G.target(v, w)$ .**

- **$G.del\_edge(v, w)$** : Xóa cạnh từ đỉnh  $v$  đến đỉnh  $w$ .

– Với ma trận kề  $A$ , thực hiện:

$$A[v][w] \leftarrow 0$$

- **$G.edges()$** : Trả về danh sách các cặp đỉnh  $(v, w)$  sao cho tồn tại cạnh từ  $v$  đến  $w$ .

– Duyệt toàn bộ ma trận kề  $A$ , với mỗi  $v, w$  thỏa:

$$A[v][w] \neq 0 \Rightarrow \text{thêm } (v, w) \text{ vào danh sách}$$

- **$G.incoming(v)$** : Trả về danh sách các đỉnh  $u$  sao cho tồn tại cạnh từ  $u$  đến  $v$ .

- Duyệt cột  $v$  trong ma trận kề  $A$ :

$$A[u][v] \neq 0 \Rightarrow u \in \text{incoming}(v)$$

- **G.outgoing(v)**: Trả về danh sách các đỉnh  $w$  sao cho tồn tại cạnh từ  $v$  đến  $w$ .

- Duyệt hàng  $v$  trong ma trận kề  $A$ :

$$A[v][w] \neq 0 \Rightarrow w \in \text{outgoing}(v)$$

- **G.source(v, w)**: Trả về đỉnh đầu của cạnh nối từ  $v$  đến  $w$  (nếu cạnh tồn tại).

- Nếu  $A[v][w] \neq 0$ , thì:

return  $v$

- **G.target(v, w)**: Trả về đỉnh cuối của cạnh nối từ  $v$  đến  $w$  (nếu cạnh tồn tại).

- Nếu  $A[v][w] \neq 0$ , thì:

return  $w$

**1.5 Extend the first-child, next-sibling tree representation, in order to support the collection of basic operations but  $T.root()$ ,  $T.number\_of\_children(v)$ , and  $T.children(v)$  in  $O(1)$  time.**

Biểu diễn cây theo mô hình **con đầu tiên - anh em kế tiếp (first-child, next-sibling)** là một cách hiệu quả để biểu diễn cây đa phân nhánh (n-ary tree) bằng cấu trúc dữ liệu nhị phân. Trong mô hình này, mỗi nút chỉ giữ hai con trỏ: một trỏ đến người con đầu tiên của nó, và một trỏ đến người anh em kế tiếp trong danh sách các con.

Tuy nhiên, với biểu diễn cơ bản này, một số thao tác thường gặp lại không thể thực hiện trong thời gian hằng số  $O(1)$ , chẳng hạn như:

- **T.root()**: lấy nút gốc của cây.
- **T.number\_of\_children(v)**: đếm số con trực tiếp của một nút  $v$ .
- **T.children(v)**: truy xuất danh sách các con của nút  $v$ .

Nguyên nhân là bởi vì để lấy danh sách con hoặc đếm số lượng con của một nút, ta cần duyệt qua chuỗi các nút thông qua con trỏ **next-sibling**, nên độ phức tạp thời gian là  $O(k)$  với  $k$  là số con.

Để thực hiện các thao tác trên trong thời gian  $O(1)$ , ta có thể mở rộng cấu trúc của mỗi nút trong cây bằng cách bổ sung thêm thông tin phụ trợ:

- **num\_children**: một biến nguyên lưu số lượng con trực tiếp của nút.



- `children_list` (tùy chọn): một danh sách (hoặc mảng) chứa con trỏ đến tất cả các con của nút, được cập nhật mỗi khi thêm hoặc xóa nút con.
- `parent` (nếu cần): con trỏ đến nút cha, giúp thuận tiện cho việc truy xuất gốc hoặc duyệt ngược lên trên cây.

Khi đó:

- `T.root()`: có thể lưu con trỏ đến nút gốc ngay trong cấu trúc cây  $\Rightarrow O(1)$ .
- `T.number_of_children(v)`: trả về giá trị `v.num_children`  $\Rightarrow O(1)$ .
- `T.children(v)`: truy xuất trực tiếp `v.children_list` nếu tồn tại  $\Rightarrow O(1)$ .

Như vậy, với một chút mở rộng hợp lý trong cấu trúc của mỗi nút, ta hoàn toàn có thể giữ nguyên lợi ích biểu diễn đơn giản của mô hình *first-child, next-sibling* mà vẫn đảm bảo hiệu năng cao cho các thao tác truy xuất thông dụng, đạt độ phức tạp thời gian  $O(1)$ .

### 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.

Khi một cây được biểu diễn dưới dạng đồ thị có hướng (graph-based representation), đặc biệt là bằng danh sách cạnh hoặc danh sách kề, chúng ta cần một cách để kiểm tra xem cấu trúc đó có thực sự là một cây hay không. Ta cần thực hiện phép kiểm tra này trong thời gian tuyến tính theo kích thước của cây (số đỉnh và số cạnh).

Nếu một đồ thị có hướng là một cây nếu thỏa mãn đồng thời các điều kiện sau:

1. Đồ thị có đúng  $n$  đỉnh và  $n - 1$  cạnh.
2. Có đúng một nút gốc (root): chỉ có một đỉnh có bậc vào bằng 0.
3. Mỗi đỉnh (trừ gốc) có đúng một nút cha: tức là mỗi đỉnh có bậc vào đúng bằng 1.
4. Đồ thị không chứa chu trình và liên thông (mọi đỉnh đều được nối với gốc qua một chuỗi cạnh).

Giả sử đồ thị được biểu diễn bằng danh sách kề với  $n$  đỉnh và  $m$  cạnh.

1. Kiểm tra số cạnh: Nếu  $m \neq n - 1$ , thì không thể là cây.
2. Tính bậc vào (in-degree) của mỗi đỉnh:
  - Duyệt qua tất cả các cạnh  $(u, v)$ , với mỗi cạnh ta tăng `in_degree[v]` thêm 1.

- Sau đó:
  - Đếm số đỉnh có  $in\_degree = 0$  (gốc): phải đúng 1 đỉnh.
  - Kiểm tra các đỉnh còn lại có  $in\_degree = 1$ .
  - Nếu không đúng, thì không phải cây.

3. Kiểm tra liên thông và không có chu trình:

- Bắt đầu từ đỉnh gốc, thực hiện duyệt đồ thị (DFS hoặc BFS).
- Trong quá trình duyệt:
  - Nếu gặp lại một đỉnh đã thăm  $\Rightarrow$  có chu trình  $\Rightarrow$  không phải cây.
  - Sau khi duyệt xong, kiểm tra số lượng đỉnh đã thăm phải đúng bằng  $n$  (tức là liên thông).

**Độ phức tạp thời gian:** Mọi bước trong giải thuật đều chạy trong thời gian  $O(n + m)$ , mà với cây thì  $m = n - 1$ , do đó tổng thể là  $O(n)$ .

Như vậy, ta có thể kiểm tra một cách chắc chắn rằng một đồ thị có hướng có biểu diễn đúng là một cây hay không bằng các bước kiểm tra đơn giản về số cạnh, bậc vào, chu trình và liên thông, tất cả đều thực hiện được trong thời gian tuyến tính.

**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.**

- **Đồ thị đường đi  $P_n$**  là một chuỗi các đỉnh nối liên tiếp, tức là có  $n$  đỉnh và  $n - 1$  cạnh, với mỗi cặp đỉnh  $v_i$  và  $v_{i+1}$  được nối bằng một cạnh.
- **Đồ thị chu trình  $C_n$**  là một đồ thị đường đi  $P_n$  được “đóng vòng”, nghĩa là thêm cạnh nối giữa đỉnh đầu tiên và đỉnh cuối cùng.
- **Đồ thị bánh xe  $W_n$**  được xây dựng từ đồ thị chu trình  $C_{n-1}$  bằng cách thêm một đỉnh trung tâm (gọi là trục bánh xe) và nối nó với tất cả các đỉnh trên chu trình.

Giả sử chúng ta có sẵn các phép toán trừu tượng như:

- $G.add\_vertex()$ : thêm đỉnh mới và trả về định danh của đỉnh.
- $G.add\_edge(u, v)$ : thêm cạnh nối hai đỉnh  $u$  và  $v$ .

**Với thuật toán sinh  $P_n$ :**

1. Khởi tạo đồ thị rỗng  $G$ .
2. Thêm  $n$  đỉnh:  $v_0, v_1, \dots, v_{n-1}$ .

3. Với mỗi  $i$  từ 0 đến  $n - 2$ , thêm cạnh từ  $v_i$  đến  $v_{i+1}$ .

**Với thuật toán sinh  $C_n$ :**

- Thực hiện giống như  $P_n$ .
- Thêm cạnh từ  $v_{n-1}$  đến  $v_0$  để tạo chu trình.

**Với thuật toán sinh  $W_n$ :**

1. Tạo chu trình  $C_{n-1}$  gồm các đỉnh  $v_0, \dots, v_{n-2}$ .
2. Thêm đỉnh trung tâm  $c$ .
3. Với mỗi đỉnh  $v_i$  trên chu trình, thêm cạnh giữa  $c$  và  $v_i$ .

Như vậy, bằng cách sử dụng các thao tác trừu tượng như thêm đỉnh và thêm cạnh, ta có thể xây dựng ba loại đồ thị  $P_n$ ,  $C_n$ , và  $W_n$  một cách tuần tự, đơn giản và rõ ràng, với chi phí thời gian tuyến tính theo số đỉnh.

**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.**

- Với đồ thị đầy đủ  $K_n$ , ta:
  1. Tạo  $n$  đỉnh bằng các lệnh `G.add_vertex()`.
  2. Duyệt qua tất cả các cặp  $(i, j)$  với  $i \neq j$ , rồi thêm cạnh nối giữa chúng bằng `G.add_edge(i, j)`.
- Với đồ thị hai phía đầy đủ  $K_{p,q}$ , ta:
  1. Tạo  $p + q$  đỉnh:  $p$  đỉnh cho tập trái  $U$ , và  $q$  đỉnh cho tập phải  $V$ .
  2. Duyệt tất cả các cặp  $(u, v)$  với  $u \in U, v \in V$ , rồi thêm cạnh nối giữa  $u$  và  $v$ .

Độ phức tạp thời gian:

- Với  $K_n$ :  $\mathcal{O}(n^2)$  do cần duyệt tất cả cặp đỉnh.
- Với  $K_{p,q}$ :  $\mathcal{O}(pq)$  vì có đúng  $pq$  cạnh cần thêm.

**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**

Ý tưởng:

Ta xây dựng một lớp `Graph` với các đặc điểm sau:

- Sử dụng ma trận kề dạng danh sách 2 chiều (list of lists) để lưu các cạnh.

- Với đồ thị có  $n$  đỉnh, ma trận kề là ma trận  $n \times n$ , phần tử tại vị trí  $(i, j)$  có giá trị **True** nếu có cạnh từ đỉnh  $i$  đến đỉnh  $j$ .
- Dựa vào ý tưởng đó, ta có thể có các thao tác sau:
  1. `add_edge(v, w)`: thêm cạnh từ  $v$  đến  $w$
  2. `del_edge(v, w)`: xoá cạnh từ  $v$  đến  $w$
  3. `edges()`: trả về danh sách các cạnh dưới dạng cặp  $(v, w)$
  4. `incoming(v)`: trả về danh sách các đỉnh  $u$  sao cho có cạnh từ  $u \rightarrow v$
  5. `outgoing(v)`: trả về danh sách các đỉnh  $w$  sao cho có cạnh từ  $v \rightarrow w$
  6. `source(v, w)`: trả về đỉnh nguồn của cạnh  $(v, w)$  nếu tồn tại
  7. `target(v, w)`: trả về đỉnh đích của cạnh  $(v, w)$  nếu tồn tại

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

- Đồ thị  $K_{p,q}$  gồm hai tập đỉnh  $U$  và  $V$  với:
  - $|U| = p, |V| = q$
  - Mỗi đỉnh trong  $U$  nối với mọi đỉnh trong  $V$
- Ghép cặp hoàn hảo (perfect matching) là tập các cạnh sao cho:
  - Mỗi đỉnh chỉ thuộc đúng một cạnh
  - Tức là toàn bộ các đỉnh được ghép đôi hoàn toàn giữa hai tập
- Điều kiện cần: **phải có**  $p = q$ , nếu không sẽ không thể có ghép cặp hoàn hảo.

Ta tìm số lượng ghép cặp hoàn hảo như sau:

Với  $p = q = n$ , số ghép cặp hoàn hảo trong  $K_{n,n}$  chính là số hoán vị của  $n$  phần tử:

$$\text{Số lượng} = n!$$

Mỗi hoán vị tương ứng với một cách ghép từng đỉnh  $u_i \in U$  với một đỉnh  $v_{\sigma(i)} \in V$ .

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

Bài toán yêu cầu xây dựng cây nhị phân đầy đủ với đúng  $n$  nút, sử dụng bộ 13 phép toán trừu tượng được đề cập trong Mục 1.3. Một cây nhị phân đầy đủ (complete binary tree) là cây trong đó các mức, ngoại trừ mức cuối cùng, đều được điền đầy đủ; các nút ở mức cuối nằm càng bên trái càng tốt.

- Bắt đầu bằng cách tạo nút gốc (root).

- Sử dụng một hàng đợi để duyệt các nút theo thứ tự mức (level-order traversal).
- Với mỗi nút trong hàng đợi, lần lượt chèn nút con trái và nút con phải (nếu tổng số nút chưa đạt  $n$ ).
- Dừng lại khi đã tạo đủ  $n$  nút.

Ta có thể mô phỏng trừu tượng như sau:

1. Gọi `T.Create()` để tạo cây rỗng.
2. Gọi `r = T.AddRoot()` để thêm nút gốc, tăng số lượng nút hiện tại lên 1.
3. Khởi tạo hàng đợi `Q` và thêm nút gốc `r` vào hàng đợi.
4. Trong khi số nút hiện tại nhỏ hơn  $n$ , thực hiện:
  - Lấy nút `v` từ hàng đợi (`v = Q.pop()`).
  - Nếu tổng số nút chưa đủ  $n$ , chèn nút trái: `l = T.AddLeft(v)`, thêm `l` vào hàng đợi.
  - Nếu tổng số nút chưa đủ  $n$ , chèn nút phải: `r = T.AddRight(v)`, thêm `r` vào hàng đợi.

**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**

Ta sẽ sinh tuần tự  $n$  nút bằng các phép toán trừu tượng, và liên kết chúng lại thành một cây bằng cách chọn ngẫu nhiên cha cho từng nút mới sinh. Thuật toán thực hiện như sau:

- Bước 1: Khởi tạo nút gốc  $r \leftarrow \text{makeNode}()$ .
- Bước 2: Khởi tạo danh sách các nút đã có:  $V \leftarrow [r]$ .
- Bước 3: Với mỗi  $i$  từ 2 đến  $n$ :
  - Sinh nút mới:  $u \leftarrow \text{makeNode}()$
  - Chọn ngẫu nhiên một nút  $v \in V$
  - Gắn  $u$  làm con của  $v$ : `addChild(v, u)`
  - Thêm  $u$  vào  $V$

**Exercise 1.9 Give an implementation of operation  $T.previous\_sibling(v)$  using the array-of-parents tree representation.**

Ta cần thao tác  $previous\_sibling(v)$  để tìm đỉnh anh/chị em nằm bên trái (trước) của đỉnh  $v$  trên cây.

- Gọi  $p = P[v]$  là cha của đỉnh  $v$ .
- Duyệt qua tất cả các đỉnh  $u$  từ 1 đến  $v - 1$ :
  - Nếu  $P[u] = p$  thì  $u$  là anh/chị em của  $v$  và đứng trước  $v$ .
  - Ghi nhận giá trị  $u$  lớn nhất thỏa điều kiện này.
- Kết quả là  $u$  lớn nhất nhỏ hơn  $v$  có cùng cha với  $v$ .

**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**

Để đạt được thời gian  $O(1)$  cho các thao tác trên, ta mở rộng cấu trúc dữ liệu như sau:

- Với mỗi nút  $v$ , lưu:
  - `first_child[v]`: con đầu tiên của  $v$
  - `next_sibling[v]`: anh em kế tiếp của  $v$
  - `parent[v]`: cha của  $v$
  - `num_children[v]`: số con của  $v$  (cập nhật khi thêm con)
  - `children_list[v]`: danh sách con của  $v$  để truy cập  $O(1)$
- Duy trì `root`: đỉnh gốc