Đồ Án Cuối Kì - Tổ Hợp Và Lý Thuyết Đồ Thị

Họ Tên: Lê Đức Long

Ngày 28 tháng 7 năm 2025

Mục lục

1 Bài toán 1

Thuật toán sinh phân hoạch giảm dần

Để sinh tất cả các phân hoạch $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_k) \in (\mathbb{N}^*)^k$ sao cho tổng bằng n, ta sử dụng thuật toán đệ quy với ràng buộc giảm dần ($\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$) nhằm tránh sinh trùng lặp.

- Mô tả thuật toán: Gọi hàm `generate_partitions`(n, k, `prefix`), trong đó:
  - n là tổng còn lại cần phân hoạch,
  - k là số phần tử còn lại cần chọn,
  - `prefix` là danh sách chứa các phần tử đã chọn.
- Ở mỗi bước, ta chọn phần tử tiếp theo i sao cho:
  - $i \leq$ `prefix`$[-1]$ (đảm bảo thứ tự giảm dần),
  - $i \leq n$ (vì không thể lấy phần tử lớn hơn tổng còn lại).

  Cụ thể, ta duyệt i từ `start` đến 1, trong đó:

$$\texttt{start} = \begin{cases} n, & \text{nếu } \texttt{prefix} \text{ rỗng} \\ \min(\texttt{prefix}[-1], n), & \text{ngược lại} \end{cases}$$

  Sau đó gọi đệ quy với $n - i$, $k - 1$, và thêm i vào `prefix`.
- Bước dừng:
  - Nếu $k = 0$ và $n = 0$: đã chọn đủ phần tử và tổng cũng đúng, sinh ra một phân hoạch hợp lệ.
  - Nếu $k = 0$ nhưng $n \neq 0$: loại bỏ nhánh này vì không còn phần tử nào để phân chia tiếp.

Biểu đồ Ferrers và chuyển vị

- Biểu đồ Ferrers F của phân hoạch $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_k)$ là lưới gồm k hàng, trong đó hàng thứ i có λ_i dấu `*`.
- Biểu đồ Ferrers chuyển vị $F^\top$ được tạo bằng cách hoán đổi hàng và cột của F, tức là:
  Hàng thứ j của $F^\top$ chứa số dấu `*` bằng với số hàng của F có độ dài $\geq j$.

Cài đặt Python

```python
def generate_partitions(n, k, prefix=[]):
    result = []
    if k == 0:
        if n == 0:
            result.append(prefix)
        return result

    # Gia tri bat dau: n hoac gia tri nho hon phan tu cuoi trong prefix
    start = n if not prefix else min(prefix[-1], n)

    # Duyet cac gia tri hop le cua so tiep theo
    for i in range(start, 0, -1):
        sub_partitions = generate_partitions(n - i, k - 1, prefix + [i])
        result.extend(sub_partitions)

    return result

def print_ferrers(partition):
    for row in partition:
        print('*' * row)
```

```python
def print_transpose(partition):
    max_len = max(partition)
    for i in range(max_len):
        for val in partition:
            print('*' if val > i else ' ', end='')
        print()

def main():
    n = int(input("Nhap n: "))
    k = int(input("Nhap k: "))
    print(f"Tat ca cac phan hoach cua {n} thanh {k} phan:\n")

    partitions = generate_partitions(n, k)
    count = 0

    for idx, part in enumerate(partitions, 1):
        print(f"Phan hoach {idx}: {part}")
        print("Bieu do Ferrers F:")
        print_ferrers(part)
        print("Bieu do Ferrers chuyen vi F^T:")
        print_transpose(part)
        print("-" * 30)
        count += 1

    print(f"Tong so phan hoach p_{k}({n}) = {count}")

# Goi ham main
if __name__ == "__main__":
    main()
```

Cài đặt C++

```cpp
#include <bits/stdc++.h>
using namespace std;

// Ham sinh phan hoach n thanh k phan
void generate_partitions(int n, int k, vector<int> prefix, vector<vector<int>>& result) {
    if (k == 0) {
        if (n == 0)
            result.push_back(prefix);
        return;
    }

    // Gia tri bat dau: n neu prefix rong, nguoc lai la min(prefix.back(), n)
    int start = prefix.empty() ? n : min(prefix.back(), n);

    // Duyet cac gia tri hop le
    for (int i = start; i >= 1; --i) {
        vector<int> new_prefix = prefix;
        new_prefix.push_back(i);
        generate_partitions(n - i, k - 1, new_prefix, result);
    }
}

// In bieu do Ferrers F
void print_ferrers(const vector<int>& partition) {
    for (int val : partition) {
        for (int i = 0; i < val; ++i)
            cout << '*';
        cout << '\n';
    }
```

```
}

// In bieu do Ferrers chuyen vi F^T
void print_transpose(const vector<int>& partition) {
    int max_len = *max_element(partition.begin(), partition.end());
    for (int i = 0; i < max_len; ++i) {
        for (int val : partition) {
            cout << (val > i ? '*' : ' ');
        }
        cout << '\n';
    }
}

int main() {
    int n, k;
    cout << "Nhap n: ";
    cin >> n;
    cout << "Nhap k: ";
    cin >> k;

    cout << "Tat ca cac phan hoach cua " << n << " thanh " << k << " phan:\n\n";

    vector<vector<int>> partitions;
    vector<int> prefix;
    generate_partitions(n, k, prefix, partitions);

    int count = 0;
    for (size_t idx = 0; idx < partitions.size(); ++idx) {
        const auto& part = partitions[idx];
        cout << "Phan hoach " << (idx + 1) << ": [";
        for (size_t i = 0; i < part.size(); ++i) {
            cout << part[i];
            if (i + 1 < part.size()) cout << ", ";
        }
        cout << "]\n";

        cout << "Bieu do Ferrers F:\n";
        print_ferrers(part);

        cout << "Bieu do Ferrers chuyen vi F^T:\n";
        print_transpose(part);

        cout << string(30, '-') << "\n";
        ++count;
    }

    cout << "Tong so phan hoach p_" << k << "(" << n << ") = " << count << "\n";

    return 0;
}
```

Sample input:

```
Nhap n: 5
Nhap k: 2
```

Sample output:

```
Tat ca cac phan hoach cua 5 thanh 2 phan:

Phan hoach 1: [4, 1]
Bieu do Ferrers F:
****
```

```
*
Bieu do Ferrers chuyen vi F^T:
**
*
*
*
------------------------------
Phan hoach 2: [3, 2]
Bieu do Ferrers F:
***
**
Bieu do Ferrers chuyen vi F^T:
**
**
*
------------------------------
Tong so phan hoach p_2(5) = 2
```

2 Bài toán 2

Thuật toán đếm số phân hoạch

Đề bài:

- Cho số tự nhiên n và k.
- Hỏi có bao nhiêu cách phân hoạch n thành **đúng k phần** (mỗi phần ≥ 1)

Bài toán 1: Đếm số phân hoạch $p_k(n)$ (đúng k phần)
Ta cần tìm số cách phân hoạch số nguyên dương n thành **đúng k phần tử**.
Công thức truy hồi:

$$p_k(n) = \begin{cases} 1, & n = 0, k = 0, \\ 0, & n < 0 \text{ hoặc } k = 0 \text{ (trừ trường hợp trên)}, \\ p_{k-1}(n-1) + p_k(n-k), & \text{ngược lại.} \end{cases}$$

Giải thích:

- $p_{k-1}(n-1)$: Lấy 1 đơn vị để tạo ra một phần mới, sau đó phân hoạch $n-1$ thành $k-1$ phần.
- $p_k(n-k)$: Giảm mỗi phần đi 1 đơn vị (tương đương đã dành ra k đơn vị), còn lại phân hoạch $n-k$ thành k phần.

Bài toán 2: Đếm số phân hoạch có phần tử lớn nhất đúng bằng k
Gọi $P_{\leq k}(n)$ là số phân hoạch của n với các phần tử không vượt quá k. Khi đó:

$$p_{\max}(n,k) = P_{\leq k}(n) - P_{\leq (k-1)}(n).$$

Công thức truy hồi cho $P_{\leq k}(n)$:

$$P_{\leq k}(n) = \begin{cases} 1, & n = 0, \\ 0, & k = 0, n > 0, \\ P_{\leq (k-1)}(n) + P_{\leq k}(n-k), & k > 0. \end{cases}$$

Ý nghĩa:

- Trường hợp $P_{\leq (k-1)}(n)$: không sử dụng số k trong phân hoạch.
- Trường hợp $P_{\leq k}(n-k)$: sử dụng ít nhất một số k, giảm n đi k và tiếp tục phân hoạch phần còn lại.

Cài đặt Python

```python
def count_partitions_exact_parts(n, k):
    dp = [[0] * (k + 1) for _ in range(n + 1)]
    dp[0][0] = 1  # Co 1 cach phan hoach 0 thanh 0 phan

    for i in range(1, n + 1):         # Tong can dat
        for j in range(1, k + 1):     # So phan
            if i >= j:
                dp[i][j] = dp[i - 1][j - 1] + dp[i - j][j]
            else:
                dp[i][j] = dp[i - 1][j - 1]
    return dp[n][k]


def count_partitions_max(n, k):
    def count_p_le_k(n, k):
        dp = [[0] * (k + 1) for _ in range(n + 1)]
        for j in range(k + 1):
            dp[0][j] = 1  # Co 1 cach phan hoach 0

        for i in range(1, n + 1):
            for j in range(1, k + 1):
                if i >= j:
                    dp[i][j] = dp[i][j - 1] + dp[i - j][j]
                else:
                    dp[i][j] = dp[i][j - 1]
        return dp[n][k]

    if k > n:
        return 0
    return count_p_le_k(n, k) - count_p_le_k(n, k - 1)

if __name__ == "__main__":
    n = int(input("Nhap n: "))
    k = int(input("Nhap k: "))

    p_k = count_partitions_exact_parts(n, k)
    p_max = count_partitions_max(n, k)

    print(f"\np_k({n}) = {p_k} ")
    print(f"p_max({n},{k}) = {p_max}")
```

Cài đặt C++

```cpp
#include <bits/stdc++.h>
using namespace std;

// Ham dem so phan hoach cua n thanh dung k phan tu
int count_partitions_exact_parts(int n, int k) {
    vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
    dp[0][0] = 1; // co 1 cach phan hoach 0 thanh 0 phan tu

    for (int i = 1; i <= n; i++) {        // tong can dat
        for (int j = 1; j <= k; j++) {    // so phan tu
            if (i >= j)
                dp[i][j] = dp[i - 1][j - 1] + dp[i - j][j];
            else
                dp[i][j] = dp[i - 1][j - 1];
        }
    }
    return dp[n][k];
```

```
}

// Ham phu dem so phan hoach cua n voi phan tu lon nhat khong vuot qua k
int count_p_le_k(int n, int k) {
    vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
    for (int j = 0; j <= k; j++)
        dp[0][j] = 1; // 1 cach phan hoach 0

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= k; j++) {
            if (i >= j)
                dp[i][j] = dp[i][j - 1] + dp[i - j][j];
            else
                dp[i][j] = dp[i][j - 1];
        }
    }
    return dp[n][k];
}

// Ham dem so phan hoach cua n sao cho phan tu lon nhat la k
int count_partitions_max(int n, int k) {
    if (k > n) return 0;
    return count_p_le_k(n, k) - count_p_le_k(n, k - 1);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cout << "Nhap n: ";
    cin >> n;
    cout << "Nhap k: ";
    cin >> k;

    int p_k = count_partitions_exact_parts(n, k);
    int p_max = count_partitions_max(n, k);

    cout << "\np_k(" << n << ") = " << p_k << "\n";
    cout << "p_max(" << n << "," << k << ") = " << p_max << "\n";

    return 0;
}
```

Sample input:

```
Nhap n: 5
Nhap k: 2
```

Sample output:

```
p_k(5) = 2
p_max(5,2) = 2
```

3 Bài toán 3

Thuật toán đếm số phân hoạch tự liên hợp

Định nghĩa. Cho $n \in \mathbb{N}$, gọi λ là một phân hoạch của n và λ' là phân hoạch liên hợp của λ. Nếu $\lambda = \lambda'$, thì λ được gọi là *phân hoạch tự liên hợp*.

Ý tưởng thuật toán.

1. Sinh tất cả các phân hoạch của n theo thứ tự không tăng dần.
2. Với mỗi phân hoạch, kiểm tra điều kiện tự liên hợp:
   - Dựng phân hoạch liên hợp λ' bằng cách hoán đổi hàng và cột trong biểu đồ Ferrers.
   - Nếu $\lambda = \lambda'$ thì giữ lại.
3. Đếm số phân hoạch tự liên hợp có đúng k phần tử.

Định lý quan trọng. Số phân hoạch tự liên hợp của n bằng số phân hoạch của n mà tất cả các phần tử đều là số lẻ và phân biệt:

$$\big|\{\text{self-conjugate partitions of } n\}\big| = \big|\{\text{odd distinct partitions of } n\}\big|.$$

Triển khai.

- *Phương pháp đệ quy:* Sinh tất cả các phân hoạch, sau đó lọc ra những phân hoạch thỏa tính chất tự liên hợp.
- *Phương pháp quy hoạch động:* Dùng bảng $\mathrm{dp}[i][j]$ đếm số phân hoạch tự liên hợp của i thành j phần.

Cài đặt Python:

```
def is_self_conjugate(partition):
    max_len = max(partition)
    conjugate = []

    for i in range(1, max_len + 1):
        count = 0
        for val in partition:
            if val >= i:
                count += 1
            else:
                break
        conjugate.append(count)

    return conjugate == partition


def generate_self_conjugate_partitions(n, k=None):
    result = []

    def backtrack(remaining, curr, max_val):
        if remaining == 0:
            if k is None or len(curr) == k:
                sorted_part = sorted(curr, reverse=True)
                if is_self_conjugate(sorted_part):
                    result.append(sorted_part)
            return
        for i in range(min(remaining, max_val), 0, -1):
            curr.append(i)
            backtrack(remaining - i, curr, i)
            curr.pop()

    backtrack(n, [], n)
    return result


def generate_odd_partitions(n):
    result = []

    def backtrack(remaining, curr, min_odd):
        if remaining == 0:
            result.append(curr.copy())
            return
        for i in range(min_odd, remaining + 1, 2):
```

```
            curr.append(i)
            backtrack(remaining - i, curr, i)
            curr.pop()

    backtrack(n, [], 1)
    return result


# (c.i) De quy
def p_k_selfconj_recursive(n, k, memo={}):
    key = (n, k)
    if key in memo:
        return memo[key]
    if n == 0 and k == 0:
        return 1
    if n <= 0 or k <= 0:
        return 0

    count = 0

    def backtrack(remaining, parts_left, curr, max_val):
        nonlocal count
        if remaining == 0 and parts_left == 0:
            sorted_part = sorted(curr, reverse=True)
            if is_self_conjugate(sorted_part):
                count += 1
            return
        if remaining <= 0 or parts_left <= 0:
            return
        for i in range(min(remaining, max_val), 0, -1):
            curr.append(i)
            backtrack(remaining - i, parts_left - 1, curr, i)
            curr.pop()

    backtrack(n, k, [], n)
    memo[key] = count
    return count


# (c.ii) Quy hoach dong
def p_k_selfconj_dp(n, k):
    dp = [[0] * (k + 1) for _ in range(n + 1)]
    dp[0][0] = 1

    for sum_val in range(1, n + 1):
        for parts in range(1, min(k, sum_val) + 1):
            count = 0

            def backtrack(remaining, parts_left, curr, max_val):
                nonlocal count
                if remaining == 0 and parts_left == 0:
                    sorted_part = sorted(curr, reverse=True)
                    if is_self_conjugate(sorted_part):
                        count += 1
                    return
                if remaining <= 0 or parts_left <= 0:
                    return
                for i in range(min(remaining, max_val), 0, -1):
                    curr.append(i)
                    backtrack(remaining - i, parts_left - 1, curr, i)
                    curr.pop()

            backtrack(sum_val, parts, [], sum_val)
            dp[sum_val][parts] = count
```

```
    return dp[n][k]

if __name__ == "__main__":
    n = int(input("Nhap n (so can phan hoach): "))
    k = int(input("Nhap k (so phan cua tu lien hop): "))

    # (a) Dem va in cac phan hoach tu lien hop co k phan
    print(f"\n(a) Cac phan hoach tu lien hop cua {n} co {k} phan:")
    sc_parts = generate_self_conjugate_partitions(n, k)
    for p in sc_parts:
        print("  ", p)
    print(f"=> So phan hoach tu lien hop p_k^selfcjg({n},{k}) = {len(sc_parts)}")

    # (b) So sanh voi so phan hoach co so luong phan tu la so le
    odd_parts = generate_odd_partitions(n)
    print(f"\n(b) So phan hoach cua {n} co so luong phan la so le:")
    for p in odd_parts:
        print("  ", p)
    print(f"=> Tong cong co {len(odd_parts)} phan hoach co so phan le")

    # (c.i) De quy
    res_recursive = p_k_selfconj_recursive(n, k)
    print(f"\n(c.i) Tinh bang de quy: p_k^selfcjg({n},{k}) = {res_recursive}")

    # (c.ii) Quy hoach dong
    res_dp = p_k_selfconj_dp(n, k)
    print(f"(c.ii) Tinh bang quy hoach dong: p_k^selfcjg({n},{k}) = {res_dp}")
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

// Ham kiem tra mot phan hoach co phai la tu lien hop hay khong
bool is_self_conjugate(const vector<int> &partition) {
    if (partition.empty()) return false;
    int max_len = partition[0]; // vi partition da sap xep giam dan
    vector<int> conjugate;
    for (int i = 1; i <= max_len; i++) {
        int count = 0;
        for (int val : partition) {
            if (val >= i)
                count++;
            else
                break;
        }
        conjugate.push_back(count);
    }
    return conjugate == partition;
}

// Sinh cac phan hoach tu lien hop cua n, neu k != -1 thi chi lay cac phan co k phan tu
void backtrack_self_conjugate(int remaining, vector<int> &curr, int max_val,
                              int n, int k, vector<vector<int>> &result) {
    if (remaining == 0) {
        if ((k == -1 || (int)curr.size() == k)) {
            vector<int> sorted_part = curr;
            sort(sorted_part.rbegin(), sorted_part.rend());
            if (is_self_conjugate(sorted_part)) {
                result.push_back(sorted_part);
            }
```

```
        }
        return;
    }
    for (int i = min(remaining, max_val); i >= 1; --i) {
        curr.push_back(i);
        backtrack_self_conjugate(remaining - i, curr, i, n, k, result);
        curr.pop_back();
    }
}

vector<vector<int>> generate_self_conjugate_partitions(int n, int k = -1) {
    vector<vector<int>> result;
    vector<int> curr;
    backtrack_self_conjugate(n, curr, n, n, k, result);
    return result;
}

// Sinh cac phan hoach cua n chi bao gom so le
void backtrack_odd(int remaining, vector<int> &curr, int min_odd,
                   vector<vector<int>> &result) {
    if (remaining == 0) {
        result.push_back(curr);
        return;
    }
    for (int i = min_odd; i <= remaining; i += 2) {
        curr.push_back(i);
        backtrack_odd(remaining - i, curr, i, result);
        curr.pop_back();
    }
}

vector<vector<int>> generate_odd_partitions(int n) {
    vector<vector<int>> result;
    vector<int> curr;
    backtrack_odd(n, curr, 1, result);
    return result;
}

// (c.i) De quy co nho
int p_k_selfconj_recursive(int n, int k, map<pair<int, int>, int> &memo) {
    pair<int, int> key = {n, k};
    if (memo.count(key)) return memo[key];
    if (n == 0 && k == 0) return 1;
    if (n <= 0 || k <= 0) return 0;

    int count = 0;
    vector<int> curr;

    function<void(int, int, vector<int> &, int)> backtrack = [&](int remaining, int
        parts_left, vector<int> &curr, int max_val) {
        if (remaining == 0 && parts_left == 0) {
            vector<int> sorted_part = curr;
            sort(sorted_part.rbegin(), sorted_part.rend());
            if (is_self_conjugate(sorted_part))
                count++;
            return;
        }
        if (remaining <= 0 || parts_left <= 0) return;
        for (int i = min(remaining, max_val); i >= 1; --i) {
            curr.push_back(i);
            backtrack(remaining - i, parts_left - 1, curr, i);
            curr.pop_back();
        }
    };
```

```
    backtrack(n, k, curr, n);
    memo[key] = count;
    return count;
}

// (c.ii) Quy hoach dong: Kiem tra tu lien hop cho tung phan hoach sinh ra
int p_k_selfconj_dp(int n, int k) {
    vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
    dp[0][0] = 1;
    for (int sum_val = 1; sum_val <= n; ++sum_val) {
        for (int parts = 1; parts <= min(k, sum_val); ++parts) {
            int count = 0;
            vector<int> curr;

            function<void(int, int, vector<int> &, int)> backtrack = [&](int remaining, int
                parts_left, vector<int> &curr, int max_val) {
                if (remaining == 0 && parts_left == 0) {
                    vector<int> sorted_part = curr;
                    sort(sorted_part.rbegin(), sorted_part.rend());
                    if (is_self_conjugate(sorted_part))
                        count++;
                    return;
                }
                if (remaining <= 0 || parts_left <= 0) return;
                for (int i = min(remaining, max_val); i >= 1; --i) {
                    curr.push_back(i);
                    backtrack(remaining - i, parts_left - 1, curr, i);
                    curr.pop_back();
                }
            };

            backtrack(sum_val, parts, curr, sum_val);
            dp[sum_val][parts] = count;
        }
    }
    return dp[n][k];
}

int main() {
    int n, k;
    cout << "Nhap n (so can phan hoach): ";
    cin >> n;
    cout << "Nhap k (so phan cua tu lien hop): ";
    cin >> k;

    // (a) Dem va in cac phan hoach tu lien hop co k phan
    cout << "\n(a) Cac phan hoach tu lien hop cua " << n << " co " << k << " phan:\n";
    auto sc_parts = generate_self_conjugate_partitions(n, k);
    for (auto &p : sc_parts) {
        cout << "  ";
        for (int x : p) cout << x << " ";
        cout << "\n";
    }
    cout << "=> So phan hoach tu lien hop p_k^selfcjg(" << n << "," << k << ") = " <<
        sc_parts.size() << "\n";

    // (b) So sanh voi phan hoach co cac so le
    auto odd_parts = generate_odd_partitions(n);
    cout << "\n(b) So phan hoach cua " << n << " co cac phan tu la so le:\n";
    for (auto &p : odd_parts) {
        cout << "  ";
        for (int x : p) cout << x << " ";
        cout << "\n";
```

```
    }
    cout << "=> Tong cong co " << odd_parts.size() << " phan hoach co so phan tu la so le\n";

    // (c.i) De quy
    map<pair<int, int>, int> memo;
    int res_recursive = p_k_selfconj_recursive(n, k, memo);
    cout << "\n(c.i) Tinh bang de quy: p_k^selfcjg(" << n << "," << k << ") = " <<
        res_recursive << "\n";

    // (c.ii) Quy hoach dong
    int res_dp = p_k_selfconj_dp(n, k);
    cout << "(c.ii) Tinh bang quy hoach dong: p_k^selfcjg(" << n << "," << k << ") = " <<
        res_dp << "\n";

    return 0;
}
```

Sample input:

```
Nhap n (so can phan hoach): 12
Nhap k (so phan cua tu lien hop): 4
```

Sample output:

```
(a) Cac phan hoach tu lien hop cua 12 co 4 phan:
   [4, 4, 2, 2]
=> So phan hoach tu lien hop p_k^selfcjg(12,4) = 1

(b) So phan hoach cua 12 co so luong phan la so le:
   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
   [1, 1, 1, 1, 1, 1, 1, 1, 1, 3]
   [1, 1, 1, 1, 1, 1, 1, 5]
   [1, 1, 1, 1, 1, 1, 3, 3]
   [1, 1, 1, 1, 1, 7]
   [1, 1, 1, 1, 3, 5]
   [1, 1, 1, 3, 3, 3]
   [1, 1, 1, 9]
   [1, 1, 3, 7]
   [1, 1, 5, 5]
   [1, 3, 3, 5]
   [1, 11]
   [3, 3, 3, 3]
   [3, 9]
   [5, 7]
=> Tong cong co 15 phan hoach co so phan le

(c.i) Tinh bang de quy: p_k^selfcjg(12,4) = 1
(c.ii) Tinh bang quy hoach dong: p_k^selfcjg(12,4) = 1
```

4 Bài toán 4

Biểu diễn cây và đồ thị

Các cách biểu diễn: Có nhiều cách để biểu diễn cây và đồ thị trong máy tính, mỗi cách có ưu nhược điểm về thời gian xử lý và bộ nhớ.

- **Ma trận kề (Adjacency Matrix)**: Là một ma trận vuông $n \times n$, trong đó phần tử $A[i][j]$ biểu thị số lượng cạnh từ đỉnh i đến đỉnh j.

  - Kiểm tra hai đỉnh có kề nhau hay không, chỉ tốn $O(1)$.
  - Tuy nhiên chiếm bộ nhớ $O(n^2)$, phù hợp cho đồ thị dày (nhiều cạnh nối giữa các đỉnh).
- **Danh sách kề (Adjacency List)**: Mỗi đỉnh lưu một danh sách các đỉnh kề với nó.
  - Dễ dàng duyệt qua tất cả đỉnh kề, độ phức tạp $O(\deg(u))$.
  - Tiết kiệm bộ nhớ: $O(n+m)$ với m là số cạnh.
- **Danh sách cạnh (Edge List)**: Lưu toàn bộ các cạnh dưới dạng bộ ba (u, v, k), trong đó u là đỉnh đầu, v là đỉnh cuối và k là số cạnh song song từ u tới v.
  - Thuận tiện cho các thuật toán làm việc trực tiếp với danh sách cạnh.
  - Không phù hợp để truy vấn nhanh hai đỉnh có kề nhau hay không.
- **Bản đồ kề (Adjacency Map)**: Biểu diễn dựa vào ánh xạ (map hoặc dictionary). Với mỗi đỉnh v:
  - *incoming[v]*: lưu danh sách các đỉnh có cạnh đi vào v.
  - *outgoing[v]*: lưu danh sách các đỉnh có cạnh đi ra từ v.

  Dây là một cách mở rộng của danh sách kề, thường dùng cho đồ thị thưa.
- **Mảng cha (Array-of-Parents):** Áp dụng riêng cho cây:

$$P[v] = \begin{cases} \text{cha của v} & \text{nếu v không phải gốc} \\ \text{nil} & \text{nếu v là gốc} \end{cases}$$

- **Con-thứ-nhất và anh-em-tiếp-theo (First-child Next-sibling):** Biểu diễn cây bằng hai mảng:

$$F[v] = \text{con đầu tiên của } v, \qquad N[v] = \text{anh em kế tiếp của } v$$

  Cách này giúp biểu diễn cây đa nhánh thành cấu trúc tương tự cây nhị phân.

Thuật toán chuyển đổi biểu diễn

Input: Một trong bốn cấu trúc chính:
Ma trận kề, Danh sách cạnh, Danh sách kề, hoặc Bản đồ kề.
Output: Sinh ra ba cấu trúc còn lại.

1. **Từ Ma trận kề:**
   - Duyệt tất cả phần tử (i, j) của ma trận.
   - Nếu $A[i][j] = k > 0$:
     - Thêm j vào danh sách kề của i (lặp k lần).
     - Thêm (i, j, k) vào danh sách cạnh.
     - Cập nhật bản đồ kề (incoming/outgoing).
2. **Từ Danh sách cạnh (Edge List):**
   - Với mỗi cạnh (u, v, k):
     - Thêm v vào danh sách kề của u (k lần).
     - Tăng $A[u][v]$ thêm k.
     - Cập nhật adjacency map với k cạnh tương ứng.
3. **Từ Danh sách kề (Adjacency List):**
   - Với mỗi cặp (u, v) trong danh sách: tăng $A[u][v]$ thêm 1.
   - Với mỗi v duy nhất:

$$k = \text{count\_of(v trong adj[u])}$$

     Thêm (u, v, k) vào danh sách cạnh và cập nhật map.
4. **Từ Bản đồ kề (Adjacency Map):**
   - Duyệt qua outgoing để tạo edge list.
   - Từ edge list, sinh danh sách kề và ma trận kề.

Mã giả (Pseudocode) **Từ Ma trận kề:**

```
for i in 0..n-1:
    for j in 0..n-1:
        k = A[i][j]
        if k > 0:
            add (i,j,k) to edge_list
            repeat k times:
                adj_list[i].append(j)
                map_outgoing[i][j].append((i,j))
                map_incoming[j][i].append((i,j))
```

Từ Danh sách cạnh:

```
for (u,v,k) in edge_list:
    A[u][v] += k
    repeat k times:
        adj_list[u].append(v)
        map_outgoing[u][v].append((u,v))
        map_incoming[v][u].append((u,v))
```

Từ Danh sách kề:

```
for u in 0..n-1:
    for v in adj_list[u]:
        A[u][v] += 1
for u in 0..n-1:
    for v in unique(adj_list[u]):
        k = count(adj_list[u], v)
        edge_list.append((u,v,k))
        repeat k times:
            map_outgoing[u][v].append((u,v))
            map_incoming[v][u].append((u,v))
```

Từ Bản đồ kề:

```
edge_list = []
for u in 0..n-1:
    for v in outgoing[u]:
        k = len(outgoing[u][v])
        edge_list.append((u,v,k))
        A[u][v] = k
        repeat k times:
            adj_list[u].append(v)
```

Độ phức tạp

- Chuyển từ ma trận kề: $O(n^2)$
- Chuyển từ danh sách kề hoặc danh sách cạnh hoặc map: $O(n + m)$

4.1 From Adjacency Matrix

Cài đặt Python:

```python
from collections import defaultdict

# Nhap ma tran ke (do thi co huong)
n = int(input("Nhap so dinh n: "))
print("Nhap ma tran ke:")
matrix = []
for _ in range(n):
```

```
    row = list(map(int, input().split()))
    matrix.append(row)

# Adjacency List
adj_list = defaultdict(list)

# Edge List
edge_list = []

# Adjacency Map: moi dinh se map toi (incoming, outgoing)
adj_map = defaultdict(lambda: (defaultdict(list), defaultdict(list)))

for i in range(n):
    for j in range(n):
        k = matrix[i][j]  # So canh tu i den j
        if k > 0:
            adj_list[i].extend([j] * k)         # Them j vao danh sach ke cua i, lap k lan
            edge_list.append((i, j, k))         # Them (i, j, k): tu i den j co k canh

            for _ in range(k):
                adj_map[i][1][j].append((i, j))  # Outgoing tu i den j
                adj_map[j][0][i].append((i, j))  # Incoming den j tu i

# In danh sach ke
print("\n============Adjacency List:")
for u in range(n):
    print(f"{u}: {adj_list[u]}")

# In danh sach canh
print("\n============Edge List:")
for u, v, k in edge_list:
    print(f"{u} {v} {k}")

# In Adjacency Map
print("\n============Adjacency Map:")
for v in range(n):
    incoming, outgoing = adj_map[v][0], adj_map[v][1]
    print(f"Dinh {v}:")
    print("  Incoming:")
    for u in incoming:
        print(f"    {u}: {incoming[u]}")
    print("  Outgoing:")
    for w in outgoing:
        print(f"    {w}: {outgoing[w]}")
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cout << "Nhap so dinh n: ";
    cin >> n;

    // Nhap ma tran ke
    cout << "Nhap ma tran ke:\n";
    vector<vector<int>> matrix(n, vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> matrix[i][j];
        }
    }

    // Adjacency List: luu danh sach ke
```

```cpp
vector<vector<int>> adj_list(n);

// Edge List: luu danh sach canh (u, v, k)
struct Edge {
    int u, v, k;
};
vector<Edge> edge_list;

// Adjacency Map: map moi dinh v -> (incoming, outgoing)
// incoming[v][u] = vector cac canh tu u toi v
// outgoing[v][w] = vector cac canh tu v toi w
vector<unordered_map<int, vector<pair<int, int>>>> incoming(n), outgoing(n);

// Duyet qua ma tran ke
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        int k = matrix[i][j]; // so canh tu i den j
        if (k > 0) {
            // Them vao adjacency list (lap k lan neu co nhieu canh)
            for (int t = 0; t < k; t++) {
                adj_list[i].push_back(j);
            }

            // Them vao edge list
            edge_list.push_back({i, j, k});

            // Cap nhat adjacency map (incoming va outgoing)
            for (int t = 0; t < k; t++) {
                outgoing[i][j].push_back({i, j}); // canh tu i toi j
                incoming[j][i].push_back({i, j}); // canh toi j tu i
            }
        }
    }
}

// In danh sach ke
cout << "\n============ Adjacency List:\n";
for (int u = 0; u < n; u++) {
    cout << u << ": ";
    for (auto v : adj_list[u]) cout << v << " ";
    cout << "\n";
}

// In danh sach canh
cout << "\n============ Edge List:\n";
for (auto &e : edge_list) {
    cout << e.u << " " << e.v << " " << e.k << "\n";
}

// In adjacency map
cout << "\n============ Adjacency Map:\n";
for (int v = 0; v < n; v++) {
    cout << "Dinh " << v << ":\n";

    // Incoming
    cout << "  Incoming:\n";
    for (auto &kv : incoming[v]) {
        cout << "    " << kv.first << ": ";
        for (auto &edge : kv.second) {
            cout << "(" << edge.first << "," << edge.second << ") ";
        }
        cout << "\n";
    }
```

```
        // Outgoing
        cout << "  Outgoing:\n";
        for (auto &kv : outgoing[v]) {
            cout << "    " << kv.first << ": ";
            for (auto &edge : kv.second) {
                cout << "(" << edge.first << "," << edge.second << ") ";
            }
            cout << "\n";
        }
    }

    return 0;
}
```

Sample input:

```
Nhap so dinh n: 5
Nhap ma tran ke:
0 2 0 0 1
0 0 1 0 0
0 0 0 2 0
1 0 0 0 1
0 0 0 0 0
```

Sample output:

```
============Adjacency List:
0: [1, 1, 4]
1: [2]
2: [3, 3]
3: [0, 4]
4: []

============Edge List:
0 1 2
0 4 1
1 2 1
2 3 2
3 0 1
3 4 1

============Adjacency Map:
Dinh 0:
  Incoming:
    3: [(3, 0)]
  Outgoing:
    1: [(0, 1), (0, 1)]
    4: [(0, 4)]
Dinh 1:
  Incoming:
    0: [(0, 1), (0, 1)]
  Outgoing:
    2: [(1, 2)]
Dinh 2:
  Incoming:
    1: [(1, 2)]
  Outgoing:
    3: [(2, 3), (2, 3)]
Dinh 3:
  Incoming:
```

```
    2: [(2, 3), (2, 3)]
  Outgoing:
    0: [(3, 0)]
    4: [(3, 4)]
Dinh 4:
  Incoming:
    0: [(0, 4)]
    3: [(3, 4)]
  Outgoing:
```

4.2 From Edge List

Cài đặt Python:

```python
from collections import defaultdict

# Nhap danh sach canh (do thi co huong)
n, m = map(int, input("Nhap so dinh va so cap dinh co canh noi (n m): ").split())
print("Nhap m dong: u v k (co k canh tu u den v):")
edge_list = []
for _ in range(m):
    u, v, k = map(int, input().split())
    edge_list.append((u, v, k))

# Tao danh sach ke
adj_list = defaultdict(list)

# Tao ma tran ke (ban dau la 0 het)
adj_matrix = [[0] * n for _ in range(n)]

# Tao adjacency map: moi dinh map toi (incoming, outgoing)
adj_map = defaultdict(lambda: (defaultdict(list), defaultdict(list)))

# Xu ly edge list
for u, v, k in edge_list:
    # Cap nhat danh sach ke
    adj_list[u].extend([v] * k)

    # Cap nhat ma tran ke
    adj_matrix[u][v] += k

    # Cap nhat adjacency map
    for _ in range(k):
        adj_map[u][1][v].append((u, v))  # Outgoing tu u den v
        adj_map[v][0][u].append((u, v))  # Incoming den v tu u

# In danh sach ke
print("\n============Adjacency List:")
for u in range(n):
    print(f"{u}: {adj_list[u]}")

# In ma tran ke
print("\n============Adjacency Matrix:")
for row in adj_matrix:
    print(" ".join(map(str, row)))

# In adjacency map
print("\n============Adjacency Map:")
for v in range(n):
    incoming, outgoing = adj_map[v][0], adj_map[v][1]
    print(f"Dinh {v}:")
    print("  Incoming:")
    for u in incoming:
```

```
        print(f"    {u}: {incoming[u]}")
    print("  Outgoing:")
    for w in outgoing:
        print(f"    {w}: {outgoing[w]}")
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cout << "Nhap so dinh va so cap dinh co canh noi (n m): ";
    cin >> n >> m;

    cout << "Nhap m dong: u v k (co k canh tu u den v):\n";
    struct Edge { int u, v, k; };
    vector<Edge> edge_list;

    for (int i = 0; i < m; i++) {
        int u, v, k;
        cin >> u >> v >> k;
        edge_list.push_back({u, v, k});
    }

    // Adjacency list: vector<int> cho moi dinh
    vector<vector<int>> adj_list(n);

    // Adjacency matrix: khoi tao n x n voi gia tri 0
    vector<vector<int>> adj_matrix(n, vector<int>(n, 0));

    // Adjacency map: su dung 2 map cho moi dinh (incoming va outgoing)
    // incoming[v][u] = danh sach cac cap (u,v)
    // outgoing[u][v] = danh sach cac cap (u,v)
    vector<unordered_map<int, vector<pair<int,int>>>> incoming(n), outgoing(n);

    // Xu ly edge list
    for (auto &e : edge_list) {
        int u = e.u, v = e.v, k = e.k;

        // Cap nhat danh sach ke
        for (int i = 0; i < k; i++) {
            adj_list[u].push_back(v);
        }

        // Cap nhat ma tran ke
        adj_matrix[u][v] += k;

        // Cap nhat adjacency map
        for (int i = 0; i < k; i++) {
            outgoing[u][v].push_back({u, v});
            incoming[v][u].push_back({u, v});
        }
    }

    // In danh sach ke
    cout << "\n============ Adjacency List:\n";
    for (int u = 0; u < n; u++) {
        cout << u << ": ";
        for (auto v : adj_list[u]) cout << v << " ";
        cout << "\n";
    }
```

```
    // In ma tran ke
    cout << "\n============ Adjacency Matrix:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << adj_matrix[i][j] << " ";
        }
        cout << "\n";
    }

    // In adjacency map
    cout << "\n============ Adjacency Map:\n";
    for (int v = 0; v < n; v++) {
        cout << "Dinh " << v << ":\n";
        // Incoming
        cout << "  Incoming:\n";
        for (auto &kv : incoming[v]) {
            cout << "    " << kv.first << ": ";
            for (auto &edge : kv.second) {
                cout << "(" << edge.first << "," << edge.second << ") ";
            }
            cout << "\n";
        }
        // Outgoing
        cout << "  Outgoing:\n";
        for (auto &kv : outgoing[v]) {
            cout << "    " << kv.first << ": ";
            for (auto &edge : kv.second) {
                cout << "(" << edge.first << "," << edge.second << ") ";
            }
            cout << "\n";
        }
    }
    return 0;
}
```

Sample input:

```
Nhap so dinh va so cap dinh co canh noi (n m): 6 9
Nhap m dong: u v k (co k canh tu u den v):
0 1 2
0 2 1
1 2 1
1 3 2
2 4 1
3 4 1
3 5 1
4 5 1
5 3 1
```

Sample output:

```
============Adjacency List:
0: [1, 1, 2]
1: [2, 3, 3]
2: [4]
3: [4, 5]
4: [5]
5: [3]

============Adjacency Matrix:
0 2 1 0 0 0
```

```
0 0 1 2 0 0
0 0 0 0 1 0
0 0 0 0 1 1
0 0 0 0 0 1
0 0 0 1 0 0

============Adjacency Map:
Dinh 0:
  Incoming:
  Outgoing:
    1: [(0, 1), (0, 1)]
    2: [(0, 2)]
Dinh 1:
  Incoming:
    0: [(0, 1), (0, 1)]
  Outgoing:
    2: [(1, 2)]
    3: [(1, 3), (1, 3)]
Dinh 2:
  Incoming:
    0: [(0, 2)]
    1: [(1, 2)]
  Outgoing:
    4: [(2, 4)]
Dinh 3:
  Incoming:
    1: [(1, 3), (1, 3)]
    5: [(5, 3)]
  Outgoing:
    4: [(3, 4)]
    5: [(3, 5)]
Dinh 4:
  Incoming:
    2: [(2, 4)]
    3: [(3, 4)]
  Outgoing:
    5: [(4, 5)]
Dinh 5:
  Incoming:
    3: [(3, 5)]
    4: [(4, 5)]
  Outgoing:
    3: [(5, 3)]
```

4.3 From Adjacency List

Cài đặt Python:

```python
from collections import defaultdict

# Nhap danh sach ke
n = int(input("Nhap so dinh n: "))
adj_list = defaultdict(list)
print("Nhap danh sach ke (nhap xong moi dinh thi Enter):")

for u in range(n):
    row = input(f"{u}: ").strip()
    if row:
        adj_list[u] = list(map(int, row.split()))
```

```
adj_matrix = [[0]*n for _ in range(n)]
edge_list = []
adj_map = defaultdict(lambda: (defaultdict(list), defaultdict(list)))

for u in range(n):
    for v in adj_list[u]:
        adj_matrix[u][v] += 1
        # Chi luu danh sach canh khi dem duoc so canh sau
        # Tam dem sau de tranh trung lap khi convert
for u in range(n):
    for v in set(adj_list[u]):
        k = adj_list[u].count(v)
        edge_list.append((u, v, k))
        for _ in range(k):
            adj_map[u][1][v].append((u, v))  # Outgoing
            adj_map[v][0][u].append((u, v))  # Incoming

# In ma tran ke
print("\n============Ma tran ke:")
for row in adj_matrix:
    print(' '.join(map(str, row)))

# In danh sach canh
print("\n============Edge List:")
for u, v, k in edge_list:
    print(f"{u} {v} {k}")

# In adjacency map
print("\n============Adjacency Map:")
for v in range(n):
    incoming, outgoing = adj_map[v][0], adj_map[v][1]
    print(f"Dinh {v}:")
    print("  Incoming:")
    for u in incoming:
        print(f"    {u}: {incoming[u]}")
    print("  Outgoing:")
    for w in outgoing:
        print(f"    {w}: {outgoing[w]}")
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cout << "Nhap so dinh n: ";
    cin >> n;
    cin.ignore(); // bo qua ky tu xuong dong sau khi nhap n

    // Danh sach ke
    vector<vector<int>> adj_list(n);
    cout << "Nhap danh sach ke (moi dong la cac dinh ke cua dinh u, enter neu rong):\n";
    for (int u = 0; u < n; u++) {
        cout << u << ": ";
        string line;
        getline(cin, line);
        if (!line.empty()) {
            stringstream ss(line);
            int v;
            while (ss >> v) {
                adj_list[u].push_back(v);
```

```
        }
    }
}

// Ma tran ke khoi tao toan 0
vector<vector<int>> adj_matrix(n, vector<int>(n, 0));

// Danh sach canh (edge list)
struct Edge { int u, v, k; };
vector<Edge> edge_list;

// Adjacency map: incoming[v][u] va outgoing[u][v]
vector<unordered_map<int, vector<pair<int,int>>>> incoming(n), outgoing(n);

// Buoc 1: cap nhat ma tran ke
for (int u = 0; u < n; u++) {
    for (int v : adj_list[u]) {
        adj_matrix[u][v] += 1;
    }
}

// Buoc 2: tao danh sach canh va cap nhat adjacency map
for (int u = 0; u < n; u++) {
    // dung set de tranh lap khi dem
    set<int> neighbors(adj_list[u].begin(), adj_list[u].end());
    for (int v : neighbors) {
        int k = count(adj_list[u].begin(), adj_list[u].end(), v); // dem so canh tu u den
             v
        edge_list.push_back({u, v, k});
        for (int i = 0; i < k; i++) {
            outgoing[u][v].push_back({u, v}); // Outgoing
            incoming[v][u].push_back({u, v}); // Incoming
        }
    }
}

// In ma tran ke
cout << "\n============ Ma tran ke:\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << adj_matrix[i][j] << " ";
    }
    cout << "\n";
}

// In danh sach canh
cout << "\n============ Edge List:\n";
for (auto &e : edge_list) {
    cout << e.u << " " << e.v << " " << e.k << "\n";
}

// In adjacency map
cout << "\n============ Adjacency Map:\n";
for (int v = 0; v < n; v++) {
    cout << "Dinh " << v << ":\n";
    cout << "  Incoming:\n";
    for (auto &kv : incoming[v]) {
        cout << "    " << kv.first << ": ";
        for (auto &edge : kv.second) {
            cout << "(" << edge.first << "," << edge.second << ") ";
        }
        cout << "\n";
    }
    cout << "  Outgoing:\n";
```

```
        for (auto &kv : outgoing[v]) {
            cout << "    " << kv.first << ": ";
            for (auto &edge : kv.second) {
                cout << "(" << edge.first << "," << edge.second << ") ";
            }
            cout << "\n";
        }
    }

    return 0;
}
```

Sample input:

```
Nhap so dinh n: 4
Nhap danh sach ke (nhap xong moi dinh thi Enter):
0: 1 1 2
1: 2 3
2: 3
3:
```

Sample output:

```
============Ma tran ke:
0 2 1 0
0 0 1 1
0 0 0 1
0 0 0 0

============Edge List:
0 1 2
0 2 1
1 2 1
1 3 1
2 3 1

============Adjacency Map:
Dinh 0:
  Incoming:
  Outgoing:
    1: [(0, 1), (0, 1)]
    2: [(0, 2)]
Dinh 1:
  Incoming:
    0: [(0, 1), (0, 1)]
  Outgoing:
    2: [(1, 2)]
    3: [(1, 3)]
Dinh 2:
  Incoming:
    0: [(0, 2)]
    1: [(1, 2)]
  Outgoing:
    3: [(2, 3)]
Dinh 3:
  Incoming:
    1: [(1, 3)]
    2: [(2, 3)]
  Outgoing:
```

4.4 From Adjacency Map

Cài đặt Python:

```python
from collections import defaultdict

# Moi dinh se map toi (incoming, outgoing)
adj_map = defaultdict(lambda: (defaultdict(list), defaultdict(list)))

# Nhap so dinh
n = int(input("Nhap so dinh n: "))

# Nhap incoming edges
print("Nhap canh incoming theo dang: to_dinh from_dinh")
print("Nhap x khi ket thuc")
while True:
    s = input("Incoming: ")
    if s.strip().lower() == 'x':
        break
    v, u = map(int, s.split())
    adj_map[v][0][u].append((u, v))  # incoming cua v tu u
    adj_map[u][1][v].append((u, v))  # dong thoi cap nhat outgoing cua u toi v

# Danh sach canh
edge_list = []
for u in range(n):
    outgoing = adj_map[u][1]
    for v in outgoing:
        k = len(outgoing[v])
        if k > 0:
            edge_list.append((u, v, k))

# Danh sach ke
adj_list = defaultdict(list)
for u, v, k in edge_list:
    adj_list[u].extend([v] * k)

# Ma tran ke
adj_matrix = [[0]*n for _ in range(n)]
for u, v, k in edge_list:
    adj_matrix[u][v] = k

# In danh sach canh
print("\n============Edge List:")
for u, v, k in edge_list:
    print(f"{u} {v} {k}")

# In danh sach ke
print("\n============Adjacency List:")
for u in range(n):
    print(f"{u}: {adj_list[u]}")

# In ma tran ke
print("\n============Ma tran ke:")
for row in adj_matrix:
    print(' '.join(map(str, row)))
```

Cài đặt C++:

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int u, v, k;
};
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cout << "Nhap so dinh n: ";
    cin >> n;

    // Dung map de dam bao thu tu tang dan cua key
    vector<map<int, vector<pair<int,int>>>> incoming(n), outgoing(n);

    cout << "Nhap canh incoming theo dang: to_dinh from_dinh\n";
    cout << "Nhap x khi ket thuc\n";

    while (true) {
        cout << "Incoming: ";
        string s;
        cin >> s;
        if (s == "x" || s == "X") break;
        int v = stoi(s);
        int u;
        cin >> u;
        incoming[v][u].push_back({u, v});
        outgoing[u][v].push_back({u, v});
    }

    // Tao danh sach canh
    vector<Edge> edge_list;
    for (int u = 0; u < n; u++) {
        for (auto &kv : outgoing[u]) {
            int v = kv.first;
            int k = kv.second.size();
            if (k > 0) {
                edge_list.push_back({u, v, k});
            }
        }
    }

    // Sap xep edge_list theo u roi v (giong Python)
    sort(edge_list.begin(), edge_list.end(), [](const Edge &a, const Edge &b) {
        if (a.u != b.u) return a.u < b.u;
        return a.v < b.v;
    });

    // Tao danh sach ke tu edge_list (theo thu tu da sap xep)
    vector<vector<int>> adj_list(n);
    for (auto &e : edge_list) {
        for (int i = 0; i < e.k; i++) {
            adj_list[e.u].push_back(e.v);
        }
    }

    // Tao ma tran ke
    vector<vector<int>> adj_matrix(n, vector<int>(n, 0));
    for (auto &e : edge_list) {
        adj_matrix[e.u][e.v] = e.k;
    }

    // In danh sach canh
    cout << "\n============ Edge List:\n";
    for (auto &e : edge_list) {
        cout << e.u << " " << e.v << " " << e.k << "\n";
    }
```

```
    // In danh sach ke
    cout << "\n============ Adjacency List:\n";
    for (int u = 0; u < n; ++u) {
        cout << u << ": ";
        for (auto v : adj_list[u]) cout << v << " ";
        cout << "\n";
    }

    // In ma tran ke
    cout << "\n============ Ma tran ke:\n";
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; j++) {
            cout << adj_matrix[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Sample input:

```
Nhap so dinh n: 4
Nhap canh incoming theo dang: to_dinh from_dinh
Nhap x khi ket thuc
Incoming: 1 0
Incoming: 1 0
Incoming: 2 0
Incoming: 2 1
Incoming: 3 1
Incoming: 3 2
Incoming: x
```

Sample output:

```
============Edge List:
0 1 2
0 2 1
1 2 1
1 3 1
2 3 1

============Adjacency List:
0: [1, 1, 2]
1: [2, 3]
2: [3]
3: []

============Ma tran ke:
0 2 1 0
0 0 1 1
0 0 0 1
0 0 0 0
```

4.5 From Array of parents

Cài đặt Python:

```
from collections import defaultdict
```

```python
def build_children_list(parents):
    n = len(parents)
    children = defaultdict(list)
    root = -1
    for i in range(n):
        if parents[i] == -1:
            root = i
        else:
            children[parents[i]].append(i)
    return root, children

def fcns_representation(root, children, n):
    first_child = [-1] * n
    next_sibling = [-1] * n

    def dfs(node):
        if not children[node]:
            return

        first_child[node] = children[node][0]

        for i in range(len(children[node]) - 1):
            next_sibling[children[node][i]] = children[node][i + 1]
        for child in children[node]:
            dfs(child)

    dfs(root)
    return first_child, next_sibling

def edge_list(first_child, next_sibling):
    edges = []
    for i in range(len(first_child)):
        if first_child[i] != -1:
            edges.append((i, first_child[i]))
        if next_sibling[i] != -1:
            edges.append((i, next_sibling[i]))
    return edges

parents = [-1, 0, 0, 1, 1, 1, 3, 3]
#         0
#       /   \
#      1     2
#    / | \
#   3  4  5
#  / \
#  6 7

n = len(parents)
root, children = build_children_list(parents)
first_child, next_sibling = fcns_representation(root, children, n)
edges = edge_list(first_child, next_sibling)

print("First-Child array:", first_child)
print("Next-Sibling array:", next_sibling)
print("Edge list:")
for u, v in edges:
    print(f"{u} -> {v}")
```

Cài đặt C++:

```cpp
#include <bits/stdc++.h>
using namespace std;

// Ham xay dung danh sach children tu mang cha
```

```cpp
pair<int, vector<vector<int>>> build_children_list(const vector<int>& parents) {
    int n = parents.size();
    vector<vector<int>> children(n);
    int root = -1;
    for (int i = 0; i < n; i++) {
        if (parents[i] == -1) {
            root = i; // dinh goc
        } else {
            children[parents[i]].push_back(i);
        }
    }
    return {root, children};
}

// Chuyen sang bieu dien First-Child Next-Sibling
pair<vector<int>, vector<int>> fcns_representation(int root,
                                                   const vector<vector<int>>& children,
                                                   int n) {
    vector<int> first_child(n, -1);    // first_child[v]: con dau tien cua v, -1 neu khong co
    vector<int> next_sibling(n, -1);   // next_sibling[v]: anh em ke tiep cua v, -1 neu khong
        co

    // DFS de xay dung 2 mang
    function<void(int)> dfs = [&](int node) {
        if (children[node].empty()) return;

        // Con dau tien cua node
        first_child[node] = children[node][0];

        // Lien ket anh em ke tiep giua cac con
        for (int i = 0; i + 1 < (int)children[node].size(); i++) {
            int u = children[node][i];
            int v = children[node][i + 1];
            next_sibling[u] = v;
        }

        // De quy cho tung con
        for (int child : children[node]) {
            dfs(child);
        }
    };

    dfs(root);
    return {first_child, next_sibling};
}

// Tao danh sach canh tu 2 mang first_child va next_sibling
vector<pair<int,int>> edge_list(const vector<int>& first_child, const vector<int>&
    next_sibling) {
    vector<pair<int,int>> edges;
    int n = first_child.size();
    for (int i = 0; i < n; i++) {
        if (first_child[i] != -1)
            edges.push_back({i, first_child[i]});
        if (next_sibling[i] != -1)
            edges.push_back({i, next_sibling[i]});
    }
    return edges;
}

int main() {
    // Mang cha: -1 la goc
    vector<int> parents = {-1, 0, 0, 1, 1, 1, 3, 3};
    // Cay tuong ung:
```

```
    //         0
    //       /   \
    //      1     2
    //    / | \
    //  3  4  5
    // / \
    // 6 7

    int n = parents.size();

    // Buoc 1: xay dung danh sach children va tim goc
    auto [root, children] = build_children_list(parents);

    // Buoc 2: tao bieu dien First-Child Next-Sibling
    auto [first_child, next_sibling] = fcns_representation(root, children, n);

    // Buoc 3: tao danh sach canh tu hai mang
    auto edges = edge_list(first_child, next_sibling);

    // In ket qua
    cout << "First-Child array: ";
    for (int x : first_child) cout << x << " ";
    cout << "\nNext-Sibling array: ";
    for (int x : next_sibling) cout << x << " ";
    cout << "\nEdge list:\n";
    for (auto [u, v] : edges) {
        cout << u << " -> " << v << "\n";
    }

    return 0;
}
```

From First child, Next sibling

Cài đặt Python:

```
def fcns_to_parents_and_tree_edges(root, first_child, next_sibling):
    n = len(first_child)
    parents = [-1] * n
    tree_edges = []

    def traverse(node):
        child = first_child[node]
        while child != -1:
            parents[child] = node
            tree_edges.append((node, child))
            traverse(child)
            child = next_sibling[child]

    traverse(root)
    return parents, tree_edges

first_child = [1, 3, -1, 6, -1, -1, -1, -1]
next_sibling = [-1, 2, -1, 4, 5, -1, 7, -1]
root = 0

parents, edge_list = fcns_to_parents_and_tree_edges(root, first_child, next_sibling)

print("\nArray of parents:")
print(parents)
print("\nEdge List:")
for u, v in edge_list:
    print(f"{u} -> {v}")
```

Cài đặt C++:

```cpp
#include <bits/stdc++.h>
using namespace std;

// H m  c h u y n  t  b i u  d i n  First-Child Next-Sibling (FCNS)
// sang  m ng  cha (parents array) v  danh s ch  c nh  (tree_edges)
pair<vector<int>, vector<pair<int,int>>>
fcns_to_parents_and_tree_edges(int root,
                               const vector<int>& first_child,
                               const vector<int>& next_sibling) {
    int n = first_child.size();
    vector<int> parents(n, -1);                 //  M ng  cha,  m c  nh  -1
    vector<pair<int,int>> tree_edges;           // Danh s ch  c nh  (u, v)

    // H m  quy  d u y t  c y  t  n t root
    function<void(int)> traverse = [&](int node) {
        int child = first_child[node];          //  L y  con  u  ti n  c a  node
        while (child != -1) {                   //  D u y t  l n  l  t  c c con v  anh em
            parents[child] = node;              // G n cha  c a  node con
            tree_edges.push_back({node, child}); // L u  c nh  (node -> child)
            traverse(child);                    //  G i  quy cho node con
            child = next_sibling[child];        // Sang anh em  t i ỉp  theo  c a  child
        }
    };

    traverse(root); //  B t  u  t  g c
    return {parents, tree_edges};
}

int main() {
    //  D  l i u  v  d
    vector<int> first_child = {1, 3, -1, 6, -1, -1, -1, -1};
    vector<int> next_sibling = {-1, 2, -1, 4, 5, -1, 7, -1};
    int root = 0;

    //  G i  h m  c h u y n  i  t  FCNS sang parents + edge list
    auto [parents, edge_list] = fcns_to_parents_and_tree_edges(root, first_child,
        next_sibling);

    // In  k ỉt  q u
    cout << "\nArray of parents:\n";
    for (int p : parents) cout << p << " ";
    cout << "\n\nEdge List:\n";
    for (auto [u, v] : edge_list) {
        cout << u << " -> " << v << "\n";
    }

    return 0;
}
```

From Edge List

Cài đặt Python:

```python
from collections import defaultdict

def edge_list_to_parents_and_fcns(edges, n, root):
    children = defaultdict(list)
    parents = [-1] * n

    for u, v in edges:
        children[u].append(v)
        parents[v] = u
```

```
    parents[root] = -1

    first_child = [-1] * n
    next_sibling = [-1] * n

    for u in range(n):
        if children[u]:
            first_child[u] = children[u][0]
            for i in range(len(children[u]) - 1):
                next_sibling[children[u][i]] = children[u][i + 1]

    return parents, first_child, next_sibling

edges = [
    (0, 1),
    (1, 3),
    (3, 6),
    (3, 7),
    (1, 4),
    (1, 5),
    (0, 2)
]
n = 8
root = 0

parents, first_child, next_sibling = edge_list_to_parents_and_fcns(edges, n, root)

print("Parents array:")
print(parents)

print("\nFirst-Child array:")
print(first_child)

print("Next-Sibling array:")
print(next_sibling)
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

// Ham chuyen tu danh sach canh sang array-of-parents va FCNS
tuple<vector<int>, vector<int>, vector<int>> edge_list_to_parents_and_fcns(
    const vector<pair<int,int>>& edges, int n, int root) {

    // 1. Xay dung danh sach children cho moi nut
    vector<vector<int>> children(n);
    vector<int> parents(n, -1); // Khoi tao tat ca cha = -1

    for (auto [u, v] : edges) {
        children[u].push_back(v);
        parents[v] = u; // Gan cha cua v la u
    }

    parents[root] = -1; // Nut goc khong co cha

    // 2. Khoi tao 2 mang FCNS
    vector<int> first_child(n, -1);
    vector<int> next_sibling(n, -1);

    // 3. Xay dung mang first_child va next_sibling
    for (int u = 0; u < n; u++) {
        if (!children[u].empty()) {
            // Con dau tien cua u
```

```
            first_child[u] = children[u][0];
            // Lien ket cac anh em
            for (int i = 0; i + 1 < (int)children[u].size(); i++) {
                int curr = children[u][i];
                int nxt = children[u][i + 1];
                next_sibling[curr] = nxt;
            }
        }
    }

    return {parents, first_child, next_sibling};
}

int main() {
    // Danh sach canh (edge list)
    vector<pair<int,int>> edges = {
        {0, 1},
        {1, 3},
        {3, 6},
        {3, 7},
        {1, 4},
        {1, 5},
        {0, 2}
    };
    int n = 8;
    int root = 0;

    // Goi ham chuyen doi
    auto [parents, first_child, next_sibling] = edge_list_to_parents_and_fcns(edges, n, root)
        ;

    // In ket qua
    cout << "Parents array:\n";
    for (int x : parents) cout << x << " ";
    cout << "\n\nFirst-Child array:\n";
    for (int x : first_child) cout << x << " ";
    cout << "\n\nNext-Sibling array:\n";
    for (int x : next_sibling) cout << x << " ";
    cout << "\n";

    return 0;
}
```

---

5 Bài toán 5

Bài giải Problems 1.1–1.6 (Valiente 2021)

Problem 1.1. Trong đồ thị đầy đủ K_n, có một cạnh giữa mỗi cặp đỉnh phân biệt. Số cạnh của K_n là:

$$|E(K_n)| = \binom{n}{2} = \frac{n(n-1)}{2}.$$

Trong đồ thị hai phía đầy đủ $K_{p,q}$ với hai tập đỉnh có kích thước p và q, mỗi đỉnh của tập thứ nhất nối với mọi đỉnh của tập thứ hai, nên số cạnh là:

$$|E(K_{p,q})| = p \cdot q.$$

Problem 1.2. Một đồ thị là hai phía khi và chỉ khi nó không chứa chu trình có độ dài lẻ. Do đó:

- C_n là hai phía khi n chẵn.
- K_n chỉ là hai phía nếu $n = 1$ hoặc $n = 2$.

Problem 1.3. Các spanning tree sinh ra từ duyệt theo chiều sâu (DFS) và duyệt theo chiều rộng (BFS) được biểu diễn ở Hình B.1 trong sách. Tổng số spanning trees của đồ thị đã cho có thể được tính bằng định thức của ma trận Laplacian:

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

trong đó L' là ma trận Laplacian loại bỏ một hàng và một cột. Kết quả:

- Với đồ thị có hướng, số spanning arborescences là 45.
- Với đồ thị vô hướng cơ sở, số spanning trees là 288.

Problem 1.4. Mở rộng biểu diễn ma trận kề bằng cách hiện thực các thao tác:

- `G.del_edge(v,w)`: đặt $A[v.index, w.index] = \text{false}$.
- `G.edges()`: trả về danh sách (v, w) sao cho $A[v.index, w.index]$ đúng.
- `G.incoming(v)`: tập đỉnh u với $A[u.index, v.index] = \text{true}$.
- `G.outgoing(v)`: tập đỉnh w với $A[v.index, w.index] = \text{true}$.
- `G.source(v,w)`: trả về v.
- `G.target(v,w)`: trả về w.

Problem 1.5. Mở rộng biểu diễn cây dạng *first-child/next-sibling*:

- Lưu thêm hai mảng $L[v]$ (last child) và $S[v]$ (previous sibling).
- Khi đó:

$$T.\text{last\_child}(v) = L[v], \quad T.\text{previous\_sibling}(v) = S[v], \quad T.\text{is\_first\_child}(v) = (S[v] = \text{nil}).$$

Problem 1.6. Để kiểm tra biểu diễn của một cây có thật sự là cây:

1. Kiểm tra rằng số đỉnh $|V|$ đúng bằng số cạnh cộng 1.
2. Duyệt BFS hoặc DFS từ gốc, đánh dấu các đỉnh đã thăm.
3. Nếu sau khi duyệt, tất cả các đỉnh đều đã được thăm, thì đây là một cây.

Mã giả:

```
function is_tree(G):
    if |V| != |E| + 1: return false
    mark all vertices unvisited
    choose root r
    DFS/BFS from r
    if exists vertex unvisited: return false
    return true
```

Lời giải các bài tập Exercises 1.1 – 1.10 [Val21, pp. 39–40]

Exercise 1.1.
Yêu cầu: Đọc/ghi đồ thị ở định dạng DIMACS.
Ý tưởng:

- Định dạng DIMACS:

```
p edge n m
e i j
e i j
...
```

- Đọc file: đầu tiên đọc dòng bắt đầu bằng `p` để lấy n và m, sau đó đọc m dòng chứa cặp (i, j) để thêm cạnh vào đồ thị.
- Khi ghi: xuất ngược lại theo đúng định dạng.

Exercise 1.2.
Yêu cầu: Đọc/ghi đồ thị ở định dạng Stanford GraphBase (SGB).
Ý tưởng:

- Định dạng bắt đầu bằng dòng `* GraphBase graph(utiltypes..., nV, mA)`.
- Tiếp theo:
  - Dòng mô tả.
  - `* Vertices` và danh sách n đỉnh: `label, Ai, 0, 0`.
  - `* Arcs` và m dòng cạnh: `V j, Ai, label, 0`.
  - Cuối cùng là `* Checksum ...`.
- Cần viết parser để đọc và hàm ghi lại cấu trúc này.

Exercise 1.3.
Sinh đồ thị đường P_n, chu trình C_n, bánh xe W_n:

- P_n: thêm các cạnh $(i, i+1)$.
- C_n: thêm các cạnh của P_n và thêm $(n, 1)$.
- W_n: thêm một đỉnh trung tâm 0, nối với tất cả các đỉnh vòng tròn $1 \dots n$ và thêm các cạnh vòng tròn.

Exercise 1.4.
Sinh K_n và $K_{p,q}$:

- K_n: thêm mọi cạnh (i, j) với $1 \le i < j \le n$.
- $K_{p,q}$: tập đỉnh tách thành X và Y, thêm cạnh (x, y) với $x \in X$ và $y \in Y$.

Exercise 1.5.
Extended adjacency matrix:

- Lưu đồ thị bằng ma trận $n \times n$.
- Viết class Python với các phương thức: `del_edge`, `edges()`, `incoming(v)`, `outgoing(v)`.

Exercise 1.6.
Liệt kê perfect matchings trong $K_{p,q}$:

- Với $p = q$, số perfect matchings là $p!$ (tất cả các hoán vị).
- Dùng backtracking để sinh các ánh xạ song ánh từ X sang Y.

Exercise 1.7.
Sinh cây nhị phân đầy đủ n đỉnh:

- Đỉnh 0 là gốc.
- Với đỉnh i, con trái là $2i + 1$, con phải là $2i + 2$ (nếu chỉ số còn trong phạm vi).

Exercise 1.8.
Sinh cây ngẫu nhiên với n đỉnh:

- Dùng thuật toán Prüfer sequence:
  - Sinh dãy ngẫu nhiên độ dài $n - 2$.
  - Chuyển đổi thành cây với n đỉnh.
- Độ phức tạp $O(n \log n)$.

Exercise 1.9.
Hàm `T.previous_sibling(v)`:

- Cho mảng cha $P[0..n-1]$.
- Với đỉnh v, duyệt tất cả đỉnh u có cùng cha $P[u] = P[v]$, lấy $u < v$ lớn nhất.

Exercise 1.10.
Cài đặt extended first-child next-sibling:

- Mỗi nút lưu chỉ số con-thứ-nhất (F) và anh-em-tiếp-theo (N).
- Class Python quản lý các thao tác `root()`, `first_child(v)`, `next_sibling(v)`.

6 Bài toán 6

Tree Edit Distance – Khoảng cách chỉnh sửa cây

Định nghĩa: Cho hai cây có thứ tự $T_1 = (V_1, E_1)$ và $T_2 = (V_2, E_2)$. Khoảng cách chỉnh sửa cây (Tree Edit Distance, TED) là **chi phí tối thiểu** để biến đổi T_1 thành T_2 thông qua một dãy các *phép chỉnh sửa cơ bản*:

- **Xoá nút (delete):** Xóa một nút trong cây.
- **Chèn nút (insert):** Thêm một nút vào cây.
- **Thay nhãn (substitute):** Đổi nhãn của một nút này thành nút khác.

Hàm chi phí $\gamma(v, w)$ thoả mãn:

$$\gamma(v,w) = \begin{cases} 0, & v \neq \lambda, w \neq \lambda \quad \text{(thay thế nhãn miễn phí)} \\ 1, & v = \lambda \text{ hoặc } w = \lambda \quad \text{(chèn/xóa)} \end{cases}$$

Trong đó λ biểu thị một nút rỗng (null).

Khoảng cách chỉnh sửa: Một phép biến đổi hợp lệ $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ của T_1 sang T_2 có **tổng chi phí**:

$$\gamma(E) = \sum_{(v,w)\in E} \gamma(v,w)$$

Khoảng cách chỉnh sửa giữa T_1 và T_2 được định nghĩa là:

$$\delta(T_1, T_2) = \min\{\gamma(E) \mid E \text{ là một phép biến đổi hợp lệ của } T_1 \text{ sang } T_2\}.$$

Ý tưởng thuật toán: Để tính TED:

1. **Xây dựng cây:** Mỗi cây được biểu diễn bằng danh sách cha (parent array), từ đó tạo cây gốc.
2. **Hàm đệ quy** `compute(node1,node2)`:
   - Nếu cả hai nút rỗng: chi phí = 0.
   - Nếu một nút rỗng: chi phí = 1 + tổng chi phí chuyển đổi tất cả các con của nút còn lại.
   - Ngược lại:
     - Tính chi phí cho cặp $(node1, node2)$.
     - Duyệt tất cả cách ghép các con của node1 với con của node2 bằng backtracking:
       - Ghép đôi con (match).
       - Xóa một con.
       - Thêm một con.
     - Lấy tổng chi phí nhỏ nhất.

Độ phức tạp: Do phương pháp backtracking duyệt toàn bộ tổ hợp, thuật toán đơn giản này có độ phức tạp *hàm mũ* theo số con. Các thuật toán nâng cao (Zhang-Shasha) sử dụng quy hoạch động để đạt độ phức tạp $O(n^3)$ cho cây có thứ tự.

Backtracking

Thuật toán Tree Edit Distance bằng Backtracking

Mục tiêu. Cho hai cây có thứ tự T_1 và T_2 (mỗi nút có một danh sách các nút con theo thứ tự). Khoảng cách chỉnh sửa cây (Tree Edit Distance) là chi phí nhỏ nhất để biến đổi T_1 thành T_2 bằng ba phép:

- Xoá một nút (delete).
- Thêm một nút (insert).
- Thay thế một nút này bằng nút khác (substitute).

Trong phiên bản đơn giản này, phép *substitute* không tính chi phí nếu hai nút đều tồn tại (vì không thay đổi nhãn nút).

Ý tưởng thuật toán. Phương pháp **đệ quy quay lui (backtracking)** thực hiện:

1. So khớp gốc của hai cây (nếu một trong hai cây rỗng thì toàn bộ cây kia phải được thêm hoặc xoá).
2. Với danh sách các nút con:
   - Duyệt lần lượt từng cặp vị trí (i, j) trong danh sách con của hai nút hiện tại.
   - Thử tất cả các lựa chọn:
     (a) Ghép con $c1[i]$ với con $c2[j]$ (match).
     (b) Xoá $c1[i]$ (align con của cây thứ nhất mà không tăng chỉ số j).
     (c) Thêm $c2[j]$ (align con của cây thứ hai mà không tăng chỉ số i).
   - Tính chi phí của mỗi lựa chọn bằng cách đệ quy gọi thuật toán trên cặp nút tương ứng.
   - Lấy giá trị nhỏ nhất trong tất cả các khả năng.

Xử lý trường hợp cơ sở.

- Nếu cả hai nút đều rỗng: chi phí = 0.
- Nếu một nút rỗng, nút kia không rỗng:

$$1 + \sum \text{tree\_edit\_distance}(\lambda, \text{child})$$

  nghĩa là phải thêm hoặc xoá nút và toàn bộ cây con của nút kia.

Mã giả.

```
function TED(node1, node2):
    if node1 == null and node2 == null:
        return 0
    if node1 == null:
        return 1 + sum(TED(null, child) for child in node2.children)
    if node2 == null:
        return 1 + sum(TED(child, null) for child in node1.children)

    cost = 0   # không tính thay thế nếu cả hai nút tồn tại
    C1 = node1.children
    C2 = node2.children
    n = len(C1), m = len(C2)
    min_cost = +infinity

    procedure align(i, j, acc):
        if i == n and j == m:
            min_cost = min(min_cost, acc)
```

```
            return
        if i == n:
            extra = sum(TED(null, C2[k]) for k=j..m-1)
            min_cost = min(min_cost, acc + extra)
            return
        if j == m:
            extra = sum(TED(C1[k], null) for k=i..n-1)
            min_cost = min(min_cost, acc + extra)
            return

        # 1. Ghép C1[i] và C2[j]
        align(i+1, j+1, acc + TED(C1[i], C2[j]))
        # 2. Xóa C1[i]
        align(i+1, j, acc + TED(C1[i], null))
        # 3. Thêm C2[j]
        align(i, j+1, acc + TED(null, C2[j]))

    align(0, 0, 0)
    return cost + min_cost
```

Đặc điểm.

- Thuật toán thử tất cả các cách ghép cặp các nút con giữa hai cây nên có độ phức tạp cao (tăng theo cấp số mũ trong trường hợp xấu nhất).
- Kết quả tìm được là khoảng cách chỉnh sửa cây tối ưu (đúng nhất).
- Phiên bản này phù hợp cho cây nhỏ; với cây lớn nên dùng thuật toán quy hoạch động (Zhang-Shasha).

Cài đặt Python:

```python
class TreeNode:
    def __init__(self, id):
        self.id = id
        self.children = []

def build_tree(parents):
    nodes = {}
    root = -1
    for i, p in enumerate(parents):
        if i not in nodes:
            nodes[i] = TreeNode(i)
        if p == -1:
            root = i
        else:
            if p not in nodes:
                nodes[p] = TreeNode(p)
            nodes[p].children.append(nodes[i])
    return nodes[root]

def tree_edit_distance(t1, t2):
    def substitution_cost(a, b):
        if a is None and b is None:
            return 0
        if a is None or b is None:
            return 1
        return 0

    def compute(node1, node2):
        if node1 is None and node2 is None:
            return 0
        if node1 is None:
            return 1 + sum(compute(None, child) for child in node2.children)
        if node2 is None:
```

```
            return 1 + sum(compute(child, None) for child in node1.children)

        cost = substitution_cost(node1, node2)
        c1, c2 = node1.children, node2.children
        n, m = len(c1), len(c2)
        min_cost = float('inf')

        def align(i, j, acc):
            nonlocal min_cost
            if i == n and j == m:
                min_cost = min(min_cost, acc)
                return
            if i == n:
                extra = sum(compute(None, c2[k]) for k in range(j, m))
                min_cost = min(min_cost, acc + extra)
                return
            if j == m:
                extra = sum(compute(c1[k], None) for k in range(i, n))
                min_cost = min(min_cost, acc + extra)
                return

            # 1. Match
            align(i + 1, j + 1, acc + compute(c1[i], c2[j]))
            # 2. Delete c1[i]
            align(i + 1, j, acc + compute(c1[i], None))
            # 3. Insert c2[j]
            align(i, j + 1, acc + compute(None, c2[j]))

        align(0, 0, 0)
        return cost + min_cost

    return compute(t1, t2)

def read_parents(prompt):
    print(prompt)
    s = input("Nhap mang cha (root = -1): ")
    return list(map(int, s.strip().split()))

if __name__ == "__main__":
    parents1 = read_parents("Cay thu nhat:")
    parents2 = read_parents("Cay thu hai:")

    root1 = build_tree(parents1)
    root2 = build_tree(parents2)

    dist = tree_edit_distance(root1, root2)
    print("\nTree edit distance:", dist)
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

// Cau truc nut cay
struct TreeNode {
    int id;
    vector<TreeNode*> children;
    TreeNode(int _id) : id(_id) {}
};

// Xay dung cay tu mang cha
TreeNode* build_tree(const vector<int>& parents) {
    int n = parents.size();
    vector<TreeNode*> nodes(n, nullptr);
```

```
    TreeNode* root = nullptr;

    for (int i = 0; i < n; i++) {
        if (!nodes[i]) nodes[i] = new TreeNode(i);
        int p = parents[i];
        if (p == -1) {
            root = nodes[i];
        } else {
            if (!nodes[p]) nodes[p] = new TreeNode(p);
            nodes[p]->children.push_back(nodes[i]);
        }
    }
    return root;
}

// Ham tinh substitution cost
int substitution_cost(TreeNode* a, TreeNode* b) {
    if (a == nullptr && b == nullptr) return 0;
    if (a == nullptr || b == nullptr) return 1;
    return 0; // khong tinh doi ten nut
}

// De quy tinh tree edit distance
int compute(TreeNode* node1, TreeNode* node2) {
    if (!node1 && !node2) return 0;
    if (!node1) {
        // Them nut moi cho toan bo cay node2
        int sum = 1;
        for (auto child : node2->children)
            sum += compute(nullptr, child);
        return sum;
    }
    if (!node2) {
        // Xoa nut va tat ca con
        int sum = 1;
        for (auto child : node1->children)
            sum += compute(child, nullptr);
        return sum;
    }

    int cost = substitution_cost(node1, node2);
    auto& c1 = node1->children;
    auto& c2 = node2->children;
    int n = (int)c1.size();
    int m = (int)c2.size();

    int min_cost = INT_MAX;

    // Dung ham de quy long nhau giong Python align
    function<void(int,int,int)> align = [&](int i, int j, int acc) {
        if (i == n && j == m) {
            min_cost = min(min_cost, acc);
            return;
        }
        if (i == n) {
            int extra = 0;
            for (int k = j; k < m; k++) extra += compute(nullptr, c2[k]);
            min_cost = min(min_cost, acc + extra);
            return;
        }
        if (j == m) {
            int extra = 0;
            for (int k = i; k < n; k++) extra += compute(c1[k], nullptr);
            min_cost = min(min_cost, acc + extra);
```

```
            return;
        }

        // 1. Ghe p cap (match)
        align(i + 1, j + 1, acc + compute(c1[i], c2[j]));
        // 2. Xoa c1[i]
        align(i + 1, j, acc + compute(c1[i], nullptr));
        // 3. Chen c2[j]
        align(i, j + 1, acc + compute(nullptr, c2[j]));
    };

    align(0, 0, 0);
    return cost + min_cost;
}

int tree_edit_distance(TreeNode* t1, TreeNode* t2) {
    return compute(t1, t2);
}

// Doc mang cha tu input
vector<int> read_parents(const string& prompt) {
    cout << prompt << "\n";
    cout << "Nhap mang cha (root = -1): ";
    string line;
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // xoa bo dem
    getline(cin, line);
    stringstream ss(line);
    vector<int> parents;
    int x;
    while (ss >> x) parents.push_back(x);
    return parents;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Doc mang cha cua cay 1
    cout << "Cay thu nhat:\n";
    string line1;
    getline(cin, line1);
    stringstream ss1(line1);
    vector<int> parents1;
    int tmp;
    while (ss1 >> tmp) parents1.push_back(tmp);

    // Doc mang cha cua cay 2
    cout << "Cay thu hai:\n";
    string line2;
    getline(cin, line2);
    stringstream ss2(line2);
    vector<int> parents2;
    while (ss2 >> tmp) parents2.push_back(tmp);

    TreeNode* root1 = build_tree(parents1);
    TreeNode* root2 = build_tree(parents2);

    int dist = tree_edit_distance(root1, root2);
    cout << "\nTree edit distance: " << dist << "\n";
    return 0;
}
```

Sample input:

```
Cay thu nhat:
Nhap mang cha (root = -1): -1 0 0 0 1 1 3 4 5
Cay thu hai:
Nhap mang cha (root = -1): -1 0 0 0 0 1 2 2 7
```

Sample output:

```
Tree edit distance: 4
```

Divide and Conquer - Dynamic Programming

Mục tiêu. Cho hai cây có thứ tự T_1 và T_2 (mỗi nút có danh sách các con theo thứ tự). Khoảng cách chỉnh sửa cây (Tree Edit Distance, TED) là chi phí tối thiểu để biến đổi T_1 thành T_2 bằng các phép:

- Xoá một nút (delete).
- Thêm một nút (insert).
- Thay thế nhãn nút (substitute).

Trong bài này, phép *substitute* giữa hai nút hiện tại không tính chi phí (bỏ qua nhãn).

Ý tưởng thuật toán. Phương pháp **chia để trị (divide and conquer)** kết hợp **lập trình động** được áp dụng như sau:

1. Nếu một nút trống (None):
   - Nếu cả hai nút rỗng: chi phí = 0.
   - Nếu một nút rỗng và nút kia không rỗng:
   $$1 + \sum \text{TED}(\lambda, \mathit{child})$$
   tức là phải thêm hoặc xoá toàn bộ cây con.
2. Nếu cả hai nút đều tồn tại:
   - Xét danh sách các con:
   $$C_1 = \text{children}(n_1), \quad C_2 = \text{children}(n_2)$$
   - Tạo bảng động dp$[i][j]$ với: dp$[i][j]$ là chi phí để biến đổi $\{C_1[0..i-1]\}$ thành $\{C_2[0..j-1]\}$.
   - Sử dụng công thức quy hoạch động tương tự *edit distance* cho chuỗi:
   $$\text{dp}[i][j] = \min \begin{cases} \text{dp}[i-1][j-1] + \text{TED}(C_1[i-1], C_2[j-1]) & \text{(so khớp cặp con)} \\ \text{dp}[i-1][j] + \text{TED}(C_1[i-1], \lambda) & \text{(xóa con)} \\ \text{dp}[i][j-1] + \text{TED}(\lambda, C_2[j-1]) & \text{(thêm con)} \end{cases}$$
   - Các trường hợp cơ sở:
   $$\text{dp}[0][j] = \sum_{k=0}^{j-1} \text{TED}(\lambda, C_2[k]), \quad \text{dp}[i][0] = \sum_{k=0}^{i-1} \text{TED}(C_1[k], \lambda)$$
   - Giá trị dp$[m][n]$ chính là chi phí tối thiểu cho việc chỉnh sửa các cây con.

Bước cuối. Khoảng cách giữa hai cây gốc là:

$$\text{TED}(n_1, n_2) = \text{dp}[|C_1|][|C_2|]$$

Trong đó chi phí thay thế tại gốc bằng 0.

Đặc điểm.

- Thuật toán hoạt động theo kiểu **chia nhỏ bài toán** thành bài toán con trên các cặp cây con, và **tổng hợp lời giải con** bằng bảng quy hoạch động.

- So với backtracking duyệt mọi hoán vị, cách này giảm số lần xét tổ hợp nhờ sử dụng bảng dp để tránh tính lại những cặp cây con giống nhau.

Mã giả.

```
function TED(node1, node2):
    if node1 == null and node2 == null:
        return 0
    if node1 == null:
        return 1 + sum(TED(null, c) for c in node2.children)
    if node2 == null:
        return 1 + sum(TED(c, null) for c in node1.children)

    C1 = node1.children
    C2 = node2.children
    m = len(C1), n = len(C2)

    dp = matrix (m+1) x (n+1)

    for i = 0..m:
        for j = 0..n:
            if i == 0 and j == 0:
                dp[i][j] = 0
            else if i == 0:
                dp[i][j] = dp[i][j-1] + TED(null, C2[j-1])
            else if j == 0:
                dp[i][j] = dp[i-1][j] + TED(C1[i-1], null)
            else:
                dp[i][j] = min(
                    dp[i-1][j-1] + TED(C1[i-1], C2[j-1]),
                    dp[i-1][j] + TED(C1[i-1], null),
                    dp[i][j-1] + TED(null, C2[j-1])
                )

    return dp[m][n]
```

Độ phức tạp. Trong trường hợp xấu, thuật toán vẫn tốn nhiều thời gian do lời gọi đệ quy lồng nhau, nhưng nhờ bảng dp trong từng bước so khớp cây con, số nhánh cần xét ít hơn so với backtracking thuần.

Cài đặt Python:

```python
class TreeNode:
    def __init__(self, id):
        self.id = id
        self.children = []

def build_tree_from_parents(parents):
    nodes = [TreeNode(i) for i in range(len(parents))]
    root = None
    for i, p in enumerate(parents):
        if p == -1:
            root = nodes[i]
        else:
            nodes[p].children.append(nodes[i])
    return root

def tree_edit_distance(t1, t2):
    # substitution cost = 0 if both nodes exist (ignore label)
    def compute(n1, n2):
        if n1 is None and n2 is None:
            return 0
        if n1 is None:
            return 1 + sum(compute(None, c) for c in n2.children)
```

```
        if n2 is None:
            return 1 + sum(compute(c, None) for c in n1.children)

        c1, c2 = n1.children, n2.children
        m, n = len(c1), len(c2)

        dp = [[0]*(n+1) for _ in range(m+1)]

        for i in range(m+1):
            for j in range(n+1):
                if i == 0 and j == 0:
                    dp[i][j] = 0
                elif i == 0:
                    dp[i][j] = dp[i][j-1] + compute(None, c2[j-1])
                elif j == 0:
                    dp[i][j] = dp[i-1][j] + compute(c1[i-1], None)
                else:
                    cost_sub = compute(c1[i-1], c2[j-1])
                    cost_del = compute(c1[i-1], None)
                    cost_ins = compute(None, c2[j-1])
                    dp[i][j] = min(
                        dp[i-1][j-1] + cost_sub,
                        dp[i-1][j] + cost_del,
                        dp[i][j-1] + cost_ins
                    )

        # substitution cost between n1 and n2 nodes is 0 (labels ignored)
        return dp[m][n]

    return compute(t1, t2)

def input_parents(prompt):
    print(prompt)
    arr = input("Nhap mang cha (root = -1): ")
    return list(map(int, arr.strip().split()))

if __name__ == "__main__":
    parents1 = input_parents("Cay thu nhat:")
    parents2 = input_parents("Cay thu hai:")

    root1 = build_tree_from_parents(parents1)
    root2 = build_tree_from_parents(parents2)

    dist = tree_edit_distance(root1, root2)
    print("\nTree Edit Distance:", dist)
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

// Cau truc nut cua cay
struct TreeNode {
    int id;
    vector<TreeNode*> children;
    TreeNode(int _id) : id(_id) {}
};

// Xay dung cay tu mang cha
TreeNode* build_tree_from_parents(const vector<int>& parents) {
    int n = parents.size();
    vector<TreeNode*> nodes(n);
    for (int i = 0; i < n; i++) {
        nodes[i] = new TreeNode(i);
```

```
    }
    TreeNode* root = nullptr;
    for (int i = 0; i < n; i++) {
        int p = parents[i];
        if (p == -1) {
            root = nodes[i];
        } else {
            nodes[p]->children.push_back(nodes[i]);
        }
    }
    return root;
}

// Ham de quy tinh tree edit distance
int compute(TreeNode* n1, TreeNode* n2) {
    // TH ca hai nut rong
    if (n1 == nullptr && n2 == nullptr) return 0;
    // TH nut n1 rong
    if (n1 == nullptr) {
        int sum = 1;
        for (auto c : n2->children)
            sum += compute(nullptr, c);
        return sum;
    }
    // TH nut n2 rong
    if (n2 == nullptr) {
        int sum = 1;
        for (auto c : n1->children)
            sum += compute(c, nullptr);
        return sum;
    }

    // Lay danh sach con cua 2 nut
    auto &c1 = n1->children;
    auto &c2 = n2->children;
    int m = (int)c1.size();
    int n = (int)c2.size();

    // Tao bang dp (m+1)x(n+1)
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 && j == 0) {
                dp[i][j] = 0;
            } else if (i == 0) {
                // Khi danh sach con cua n1 rong -> phai them tat ca con cua n2
                dp[i][j] = dp[i][j - 1] + compute(nullptr, c2[j - 1]);
            } else if (j == 0) {
                // Khi danh sach con cua n2 rong -> phai xoa tat ca con cua n1
                dp[i][j] = dp[i - 1][j] + compute(c1[i - 1], nullptr);
            } else {
                // Ba truong hop:
                // 1. Khop con c1[i-1] voi c2[j-1]
                int cost_sub = compute(c1[i - 1], c2[j - 1]);
                // 2. Xoa con c1[i-1]
                int cost_del = compute(c1[i - 1], nullptr);
                // 3. Chen con c2[j-1]
                int cost_ins = compute(nullptr, c2[j - 1]);

                dp[i][j] = min({
                    dp[i - 1][j - 1] + cost_sub,
                    dp[i - 1][j] + cost_del,
                    dp[i][j - 1] + cost_ins
```

```
                });
            }
        }
    }

    // Chi phi substitute o goc la 0 vi khong quan tam nhan nut
    return dp[m][n];
}

// Ham tinh tree edit distance
int tree_edit_distance(TreeNode* t1, TreeNode* t2) {
    return compute(t1, t2);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Doc du lieu dau vao
    cout << "Cay thu nhat (nhap mang cha, root = -1):\n";
    vector<int> parents1;
    string line;
    getline(cin, line);
    {
        stringstream ss(line);
        int x;
        while (ss >> x) parents1.push_back(x);
    }

    cout << "Cay thu hai (nhap mang cha, root = -1):\n";
    vector<int> parents2;
    getline(cin, line);
    {
        stringstream ss(line);
        int x;
        while (ss >> x) parents2.push_back(x);
    }

    // Xay dung cay
    TreeNode* root1 = build_tree_from_parents(parents1);
    TreeNode* root2 = build_tree_from_parents(parents2);

    // Tinh khoang cach TED
    int dist = tree_edit_distance(root1, root2);

    cout << "\nTree Edit Distance: " << dist << "\n";
    return 0;
}
```

Sample input:

```
Cay thu nhat:
Nhap mang cha (root = -1): -1 0 0 0 1 1 3 4 5
Cay thu hai:
Nhap mang cha (root = -1): -1 0 0 0 0 1 2 2 7
```

Sample output:

```
Tree edit distance: 4
```

7 Bài toán 7

Tree Traversal – Duyệt cây

Định nghĩa: Cho một cây $T = (V, E)$ có gốc. Duyệt cây (Tree Traversal) là quá trình thăm từng nút trong cây theo một thứ tự xác định. Tùy theo mục đích, ta có nhiều kiểu duyệt khác nhau.

Các phương pháp duyệt cây:

1. **Duyệt trước (Preorder Traversal):**
   - Ý tưởng:
     (a) Xử lý nút hiện tại.
     (b) Duyệt các nút con từ trái qua phải.
   - Mã giả:

```
preorder(node):
    print(node)
    for child in children(node):
        preorder(child)
```

2. **Duyệt sau (Postorder Traversal):**
   - Ý tưởng:
     (a) Duyệt tất cả nút con từ trái qua phải.
     (b) Xử lý nút hiện tại sau cùng.
   - Mã giả:

```
postorder(node):
    for child in children(node):
        postorder(child)
    print(node)
```

3. **Duyệt từ gốc xuống (Top-down Traversal / Level-order):**
   - Sử dụng hàng đợi (queue).
   - Các bước:
     (a) Đưa gốc vào hàng đợi.
     (b) Trong khi hàng đợi không rỗng:
       – Lấy ra một nút ở đầu hàng đợi.
       – In nút đó.
       – Đưa tất cả con của nút này vào cuối hàng đợi.
   - Mã giả:

```
queue = [root]
while queue not empty:
    node = queue.pop_front()
    print(node)
    for child in children(node):
        queue.push_back(child)
```

4. **Duyệt từ lá lên (Bottom-up Traversal):**
   - Ý tưởng:
     (a) Thực hiện BFS để lưu các nút theo từng mức (level).
     (b) In các nút theo thứ tự từ mức sâu nhất lên mức gốc.
   - Mã giả:

```
        levels = dictionary
        queue = [(root, 0)]
        while queue not empty:
            node, level = queue.pop_front()
            levels[level].append(node)
            for child in children(node):
                queue.push_back((child, level+1))

        for level from max_level down to 0:
            for node in levels[level]:
                print(node)
```

Độ phức tạp: Tất cả các phương pháp duyệt đều có độ phức tạp $O(n)$ với n là số nút trong cây, vì mỗi nút được thăm đúng một lần.

Cài đặt Python:

```python
from collections import defaultdict, deque

def build_tree_from_parents(parents):
    tree = defaultdict(list)
    root = -1
    for child, parent in enumerate(parents):
        if parent == -1:
            root = child
        else:
            tree[parent].append(child)
    return tree, root

def preorder(tree, node):
    print(node, end=' ')
    for child in tree[node]:
        preorder(tree, child)

def postorder(tree, node):
    for child in tree[node]:
        postorder(tree, child)
    print(node, end=' ')

def top_down(tree, root):
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node, end=' ')
        for child in tree[node]:
            queue.append(child)

def bottom_up(tree, root):
    levels = defaultdict(list)
    max_level = 0
    queue = deque([(root, 0)])

    while queue:
        node, level = queue.popleft()
        levels[level].append(node)
        max_level = max(max_level, level)
        for child in tree[node]:
            queue.append((child, level + 1))

    for level in reversed(range(max_level + 1)):
        for node in levels[level]:
            print(node, end=' ')
```

```
print("Nhap mang cha (-1 cho root):")
parents = list(map(int, input().split()))

tree, root = build_tree_from_parents(parents)

print("Duyet preorder:")
preorder(tree, root)
print("\nDuyet postorder:")
postorder(tree, root)
print("\nDuyet top-down:")
top_down(tree, root)
print("\nDuyet bottom-up:")
bottom_up(tree, root)
```

Cài đặt C++:

```
#include <bits/stdc++.h>
using namespace std;

// Xay dung cay tu mang cha
pair<unordered_map<int, vector<int>>, int> build_tree_from_parents(const vector<int>& parents
    ) {
    unordered_map<int, vector<int>> tree;
    int root = -1;
    for (int child = 0; child < (int)parents.size(); child++) {
        int parent = parents[child];
        if (parent == -1) {
            root = child;
        } else {
            tree[parent].push_back(child);
        }
    }
    return {tree, root};
}

// Duyet tien thu tu (preorder)
void preorder(unordered_map<int, vector<int>>& tree, int node) {
    cout << node << " "; // Xu ly nut hien tai truoc
    for (int child : tree[node]) {
        preorder(tree, child);
    }
}

// Duyet hau thu tu (postorder)
void postorder(unordered_map<int, vector<int>>& tree, int node) {
    for (int child : tree[node]) {
        postorder(tree, child);
    }
    cout << node << " "; // Xu ly nut hien tai sau
}

// Duyet top-down (BFS)
void top_down(unordered_map<int, vector<int>>& tree, int root) {
    queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        for (int child : tree[node]) {
```

```
            q.push(child);
        }
    }
}

// Duyet bottom-up
void bottom_up(unordered_map<int, vector<int>>& tree, int root) {
    unordered_map<int, vector<int>> levels;
    queue<pair<int, int>> q;
    q.push({root, 0});
    int max_level = 0;

    // BFS de xac dinh cac muc (level)
    while (!q.empty()) {
        auto [node, level] = q.front();
        q.pop();
        levels[level].push_back(node);
        max_level = max(max_level, level);
        for (int child : tree[node]) {
            q.push({child, level + 1});
        }
    }

    // Duyet tu muc thap nhat len muc cao nhat
    for (int level = max_level; level >= 0; level--) {
        for (int node : levels[level]) {
            cout << node << " ";
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cout << "Nhap mang cha (-1 cho root):\n";
    vector<int> parents;
    int x;
    while (cin.peek() != '\n' && cin >> x) {
        parents.push_back(x);
    }

    auto [tree, root] = build_tree_from_parents(parents);

    cout << "Duyet preorder:\n";
    preorder(tree, root);

    cout << "\nDuyet postorder:\n";
    postorder(tree, root);

    cout << "\nDuyet top-down:\n";
    top_down(tree, root);

    cout << "\nDuyet bottom-up:\n";
    bottom_up(tree, root);

    return 0;
}
```

Sample input:

```
Nhap mang cha (-1 cho root):
-1 0 0 1 1 2 2
```

Sample output:

```
Duyet preorder:
0 1 3 4 2 5 6
Duyet postorder:
3 4 1 5 6 2 0
Duyet top-down:
0 1 2 3 4 5 6
Duyet bottom-up:
3 4 5 6 1 2 0
```

8 Bài toán 8 - 10

Thuật toán BFS

Thuật toán BFS (*Breadth-First Search*) được sử dụng để **duyệt hoặc tìm kiếm trên đồ thị** theo từng lớp (tầng).

Ý tưởng thuật toán

1. Bắt đầu từ một đỉnh nguồn s, đánh dấu đỉnh này là đã thăm (visited) và đưa vào hàng đợi.
2. Lặp lại cho đến khi hàng đợi rỗng:
   (a) Lấy một đỉnh u từ đầu hàng đợi.
   (b) Xử lý u (thêm vào thứ tự duyệt).
   (c) Với mỗi đỉnh kề v của u, nếu v chưa được thăm:
   - Đánh dấu v là đã thăm.
   - Đưa v vào cuối hàng đợi.

Đặc điểm của BFS

- BFS duyệt theo **từng lớp**: thăm tất cả các đỉnh có khoảng cách 1 cạnh đến s, sau đó đến các đỉnh cách 2 cạnh, ...
- Dữ liệu quan trọng: **hàng đợi (queue FIFO)** để quản lý thứ tự duyệt.

Kết quả

- Xây dựng được cây BFS (*breadth-first tree*) với gốc tại đỉnh s.
- Với đồ thị không trọng số, BFS đảm bảo tìm được đường đi ngắn nhất (ít cạnh nhất) từ s đến các đỉnh còn lại.

Độ phức tạp

$$O(|V| + |E|)$$

Mỗi đỉnh và mỗi cạnh được duyệt tối đa một lần.

Ứng dụng

- Tìm thành phần liên thông của đồ thị.
- Xác định đường đi ngắn nhất trong đồ thị không trọng số.
- Kiểm tra đồ thị hai phía (bipartite graph).

Vì sao BFS áp dụng cho nhiều loại đồ thị

- **Finite simple graph**: BFS lần lượt thăm từng đỉnh và đảm bảo dừng sau hữu hạn bước vì số đỉnh và số cạnh là hữu hạn.
- **Multigraph**: BFS xử lý được nhiều cạnh giữa cùng một cặp đỉnh nhờ việc *kiểm tra visited*, do đó không bị lặp vô hạn.
- **General graph**: BFS vẫn hoạt động chính xác nhờ cơ chế đánh dấu đỉnh đã thăm, tránh lặp lại trong chu trình.

8.1 Cài đặt Python

```python
from collections import deque, defaultdict

# n: so dinh, m: so cap dinh noi voi nhau
n, m = map(int, input().split())

# key: int (dinh), value: list[int] (danh sach dinh ke)
adj = defaultdict(list)

for _ in range(m):
    u, v, k = map(int, input().split())
    # Them k lan canh tu u den v
    for _ in range(k):
        adj[u].append(v)
        adj[v].append(u) # Do thi vo huong
def bfs(start):
    visited = [False] * n
    queue = deque()
    order = []

    visited[start] = True
    queue.append(start)

    while queue:
        u = queue.popleft()
        order.append(u)
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                queue.append(v)
    return order

# Thuc hien BFS tu dinh 0
start_node = 0
result = bfs(start_node)
print(*result)
```

Cài đặt C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m;
vector<vector<int>> adj; // adj[u] la danh sach ke cua dinh u

vector<int> bfs(int start) {
    vector<bool> visited(n, false); // danh dau cac dinh da tham
    queue<int> q;                   // hang doi de luu cac dinh se duyet
    vector<int> order;              // luu thu tu duyet cac dinh

    visited[start] = true; // danh dau dinh bat dau
    q.push(start);         // dua dinh bat dau vao hang doi

    while (!q.empty()) {
        int u = q.front(); // lay dinh o dau hang doi
        q.pop();
        order.push_back(u); // ghi nhan thu tu duyet

        // duyet tat ca cac dinh ke voi u
        for (int v : adj[u]) {
            if (!visited[v]) {        // neu v chua duoc tham
```

```
                visited[v] = true;     // danh dau v
                q.push(v);             // dua v vao hang doi
            }
        }
    }
    return order;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n >> m; // nhap so dinh va so cap dinh noi voi nhau
    adj.assign(n, {}); // khoi tao danh sach ke

    for (int i = 0; i < m; i++) {
        int u, v, k;
        cin >> u >> v >> k;
        // them k lan canh tu u den v
        for (int j = 0; j < k; j++) {
            adj[u].push_back(v);
            adj[v].push_back(u); // do thi vo huong
        }
    }

    int start_node = 0; // bat dau BFS tu dinh 0
    vector<int> result = bfs(start_node);

    // in ra thu tu duyet
    for (int u : result) {
        cout << u << " ";
    }
    cout << "\n";

    return 0;
}
```

Sample input:

```
6 10
0 1 2
0 2 3
1 2 1
2 3 5
3 3 1
3 4 2
2 4 1
1 4 6
1 5 3
0 5 2
```

Sample output:

```
0 1 2 5 4 3
```

9 Bài toán 11 - 13

Thuật toán DFS (Depth-First Search)

Thuật toán DFS được dùng để duyệt hoặc tìm kiếm trên đồ thị/cây bằng cách đi sâu nhất có thể theo một nhánh trước, sau đó quay lui (backtrack).

Mô tả thuật toán
Gọi hàm **DFS(u)** với đỉnh bắt đầu u, trong đó:

- u là đỉnh hiện tại đang xét.
- `visited` là tập các đỉnh đã được thăm.
- G là đồ thị được biểu diễn dưới dạng danh sách kề hoặc ma trận kề.

Các bước thực hiện

1. Đánh dấu đỉnh u là đã thăm.
2. Với mỗi đỉnh kề v của u:
   - Nếu v chưa được thăm, gọi đệ quy **DFS(v)** để đi sâu tiếp.

Điều kiện dừng Khi không còn đỉnh kề nào chưa được thăm từ u, thuật toán quay lui về đỉnh trước đó.

Mã giả thuật toán:

```
DFS(u):
    visited[u] <- true
    for v in Adj[u]:
        if not visited[v]:
            DFS(v)
```

Đặc điểm

- Duyệt theo chiều sâu, có thể cài đặt bằng **đệ quy** hoặc **ngăn xếp (stack)**.
- Độ phức tạp thời gian: $O(V + E)$ với V là số đỉnh và E là số cạnh.

Tính tổng quát của DFS: DFS hoạt động được trên nhiều loại đồ thị khác nhau vì:

1. **Finite simple graph:** DFS chỉ cần danh sách các đỉnh kề và trạng thái đã thăm. Với số đỉnh và số cạnh hữu hạn, việc duyệt sẽ luôn kết thúc.
2. **Multigraph:** DFS không quan tâm đến số lượng cạnh giữa hai đỉnh. Nếu có nhiều cạnh song song nối cùng một cặp đỉnh, việc đánh dấu `visited` bảo đảm mỗi đỉnh chỉ được thăm một lần, do đó các cạnh dư sẽ bị bỏ qua tự nhiên.
3. **General graph:** Nguyên tắc của DFS không phụ thuộc vào dạng biểu diễn cụ thể. Chỉ cần xác định được tập kề `Adj[u]` thì DFS có thể áp dụng, kể cả đồ thị vô hướng, có hướng, có chu trình, hoặc không liên thông (trong trường hợp không liên thông, cần chạy DFS từ nhiều đỉnh).

Chính nhờ cơ chế `visited` mà DFS tránh việc lặp vô hạn và đảm bảo duyệt được toàn bộ đồ thị một cách hệ thống.

Cài đặt Python

```python
from collections import deque, defaultdict

# n: so dinh, m: so cap dinh noi voi nhau
n, m = map(int, input().split())

# key: int (dinh), value: list[int] (danh sach dinh ke)
adj = defaultdict(list)

for _ in range(m):
    u, v, k = map(int, input().split())
    # Them k lan canh tu u den v
    for _ in range(k):
        adj[u].append(v)
        adj[v].append(u) # Do thi vo huong
```

```python
def dfs(u, visited, order):
    visited[u] = True
    order.append(u)
    for v in adj[u]:
        if not visited[v]:
            dfs(v, visited, order)

# Thuc hien DFS tu dinh 0
visited = [False] * n
order = []
dfs(0, visited, order)
print(*order)
```

9.1 Cài đặt C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m;
vector<vector<int>> adj; // adj[u] la danh sach ke cua dinh u
vector<bool> visited;    // danh dau dinh da tham chua
vector<int> order;       // luu thu tu cac dinh duoc duyet

// Ham DFS de quy
void dfs(int u) {
    visited[u] = true;    // danh dau da tham dinh u
    order.push_back(u);   // luu dinh u vao thu tu duyet
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v);       // goi de quy voi dinh v
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n >> m; // nhap so dinh n va so cap dinh noi voi nhau m
    adj.assign(n, {}); // khoi tao danh sach ke

    // Nhap danh sach canh
    for (int i = 0; i < m; i++) {
        int u, v, k;
        cin >> u >> v >> k;
        // them k lan canh tu u den v
        for (int j = 0; j < k; j++) {
            adj[u].push_back(v);
            adj[v].push_back(u); // do thi vo huong
        }
    }

    visited.assign(n, false); // ban dau tat ca cac dinh chua tham
    order.clear();

    dfs(0); // thuc hien DFS bat dau tu dinh 0

    // In ra thu tu cac dinh duoc duyet
    for (int u : order) {
        cout << u << " ";
    }
    cout << "\n";
```

```
    return 0;
}
```

Sample input:

```
6 10
0 1 2
0 2 3
1 2 1
2 3 5
3 3 1
3 4 2
2 4 1
1 4 6
1 5 3
0 5 2
```

Sample output:

```
0 1 2 3 4 5
```

10 Bài toán 14 - 16

Thuật toán Dijkstra

Thuật toán Dijkstra được sử dụng để tìm **đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại** trong đồ thị có trọng số không âm.

Ý tưởng

1. Khởi tạo khoảng cách:
   - Gán khoảng cách từ đỉnh nguồn đến chính nó bằng 0.
   - Gán khoảng cách từ đỉnh nguồn đến tất cả các đỉnh khác là $+\infty$.
2. Tập S rỗng ban đầu sẽ lưu các đỉnh đã được chọn (đã biết khoảng cách ngắn nhất cố định).
3. Lặp lại cho đến khi tất cả các đỉnh đã được xử lý:
   (a) Chọn từ các đỉnh chưa nằm trong S một đỉnh u có **khoảng cách tạm thời nhỏ nhất**.
   (b) Thêm u vào S.
   (c) Với mỗi đỉnh kề v của u, thực hiện cập nhật (relaxation):

$$\text{nếu } dist[v] > dist[u] + w(u,v) \text{ thì gán } dist[v] = dist[u] + w(u,v).$$

Kết thúc: Khi thuật toán dừng, mảng `dist` chứa độ dài đường đi ngắn nhất từ đỉnh nguồn đến mọi đỉnh.

Độ phức tạp

- Dùng hàng đợi ưu tiên (min-heap): $O((V+E)\log V)$.
- Áp dụng với đồ thị có trọng số không âm.

Ứng dụng

- Tìm đường đi tối ưu trong bản đồ (GPS).
- Mạng máy tính (giao thức OSPF, IS-IS).

- Là một bước trong các thuật toán phức tạp hơn như Johnson.

Tính tổng quát của thuật toán Dijkstra: Thuật toán Dijkstra có thể áp dụng cho nhiều loại đồ thị khác nhau nhờ các lý do sau:

1. **Finite simple graph:**
   Đồ thị đơn có số đỉnh và số cạnh hữu hạn. Dijkstra chỉ cần thông tin trọng số các cạnh và danh sách kề, do đó luôn thực hiện được và dừng sau hữu hạn bước.
2. **Multigraph:**
   Nếu giữa hai đỉnh có nhiều cạnh song song với các trọng số khác nhau, thuật toán vẫn hoạt động bình thường vì quá trình cập nhật (relaxation) luôn chọn được cạnh có tổng chi phí nhỏ nhất. Cạnh kém tối ưu sẽ bị bỏ qua.
3. **General graph:**
   Dijkstra không yêu cầu đồ thị phải đơn hay không đa cạnh, chỉ cần:
   - Trọng số các cạnh không âm.
   - Xác định được danh sách các đỉnh kề và trọng số cạnh.

   Thuật toán sẽ tìm được đường đi ngắn nhất ngay cả khi đồ thị là vô hướng, có hướng, hoặc nhiều thành phần liên thông.

Cài đặt Python

```python
import heapq
from collections import defaultdict

# n: so dinh, m: so cap dinh co canh noi
n, m = map(int, input().split())

# adj[u] = danh sach cac cap (v, w) nghia la co canh tu u den v voi trong so w
adj = defaultdict(list)

# Nhap m dong, moi dong co: u, v, k (co k canh giua u va v), sau do k dong trong so
for _ in range(m):
    u, v, k = map(int, input().split())
    for _ in range(k):
        w = int(input())  # Nhap trong so w cua 1 canh tu u den v
        adj[u].append((v, w))
        adj[v].append((u, w))  # Do thi vo huong


# Thuat toan Dijkstra tim duong di ngan nhat tu dinh start
def dijkstra(start):
    dist = [float('inf')] * n  # dist[u]: khoang cach ngan nhat tu start den u
    dist[start] = 0

    heap = [(0, start)]  # Hang doi uu tien (khoang cach, dinh)

    while heap:
        d, u = heapq.heappop(heap)

        if d > dist[u]:
            continue  # Bo qua neu da co duong di ngan hon

        for v, w in adj[u]:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                heapq.heappush(heap, (dist[v], v))

    return dist
```

```python
# Dijkstra tu dinh 0
start_node = 0
shortest_distances = dijkstra(start_node)

# In ra khoang cach ngan nhat tu dinh 0 den cac dinh con lai
print("Khoang cach ngan nhat tu dinh", start_node, ":")
for i, d in enumerate(shortest_distances):
    if d == float('inf'):
        print(f"Dinh {i}: khong the den")
    else:
        print(f"Dinh {i}: {d}")
```

Cài đặt C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m;
vector<vector<pair<int, int>>> adj;
// adj[u] la danh sach ke cua dinh u
// moi phan tu la cap (v, w) nghia la co canh tu u den v co trong so w

// Ham thuc hien thuat toan Dijkstra tu dinh bat dau
vector<int> dijkstra(int start) {
    const int INF = 1e9; // gia tri vo cung (lon)
    vector<int> dist(n, INF); // mang dist luu khoang cach ngan nhat tu start den moi dinh
    dist[start] = 0; // khoang cach tu start den chinh no la 0

    // hang doi uu tien (min-heap) luu cap (khoang cach tam thoi, dinh)
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start}); // dua dinh bat dau vao hang doi

    while (!pq.empty()) {
        auto [d, u] = pq.top(); // lay ra phan tu co khoang cach nho nhat
        pq.pop();

        // neu da tim duoc duong ngan hon truoc do thi bo qua
        if (d > dist[u]) continue;

        // duyet tat ca canh ke cua dinh u
        for (auto [v, w] : adj[u]) {
            // neu tim duoc duong ngan hon den v thi cap nhat
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v}); // dua vao hang doi
            }
        }
    }

    return dist; // tra ve ket qua la mang khoang cach ngan nhat
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n >> m; // nhap so dinh n va so cap dinh co canh noi m
    adj.assign(n, {}); // khoi tao danh sach ke

    // Nhap m cap dinh
    for (int i = 0; i < m; i++) {
```

```
        int u, v, k;
        cin >> u >> v >> k; // co k canh giua u va v
        for (int j = 0; j < k; j++) {
            int w;
            cin >> w; // trong so cua canh
            // do thi vo huong nen luu ca 2 chieu
            adj[u].push_back({v, w});
            adj[v].push_back({u, w});
        }
    }

    int start = 0; // chon dinh bat dau la 0
    vector<int> dist = dijkstra(start); // goi ham Dijkstra

    cout << "Khoang cach ngan nhat tu dinh " << start << ":\n";
    for (int i = 0; i < n; i++) {
        if (dist[i] == 1000000000)
            cout << "Dinh " << i << ": khong the den\n";
        else
            cout << "Dinh " << i << ": " << dist[i] << "\n";
    }

    return 0;
}
```

Sample input:

```
7 8
0 1 2
4
2
0 2 1
3
1 3 1
6
2 3 2
1
2
3 4 1
2
4 4 1
0
5 6 1
1
6 3 1
8
```

Sample output:

```
Khoang cach ngan nhat tu dinh 0 :
Dinh 0: 0
Dinh 1: 2
Dinh 2: 3
Dinh 3: 4
Dinh 4: 6
Dinh 5: 13
Dinh 6: 12
```