TP n°4 : Structures de données : Union-Find et sommes préfixes

1 Union-Find

La structure de données **Union-Find**, aussi appelée **ensemble disjoint** (DSU, *Disjoint Set Union*), permet de gérer efficacement un ensemble d'éléments partitionnés en sous-ensembles. On souhaite réaliser deux types d'opérations :

— **Find**(x) : déterminer à quel sous-ensemble appartient l'élément x,

— **Union**(x, y) : fusionner les sous-ensembles contenant x et y.

Cette structure est utilisée dans de nombreux algorithmes, comme **Kruskal** (pour les arbres couvrants minimaux), la détection de cycles, ou encore des problèmes de connexité dynamique.

Pour optimiser ces opérations, on utilise deux techniques :

1. **Compression de chemin (path compression)** lors de `find`, qui aplatit l'arbre des représentants et accélère les recherches ultérieures.
2. **Union par taille (union by size)**, qui attache l'arbre le plus petit au plus grand, pour limiter la hauteur des arbres.

Avec ces deux optimisations, la complexité amortie est presque constante, notée $O(\alpha(n))$, où α est la fonction inverse d'Ackermann (extrêmement lente).

Pseudo-code détaillé des procédures

```
1  Procédure MAKE-SET(n)
2      for i ← 1 to n do
3          parent[i] ← i
4          size[i] ← 1
5  Procédure FIND(x)
6      if parent[x] = x then
7          return x
8      parent[x] ← FIND(parent[x])                    // compression de chemin
9      return parent[x]
10 Procédure UNION(x, y)
11     racineX ← FIND(x)
12     racineY ← FIND(y)
13     if racineX = racineY then
14         return                                     // déjà dans le même ensemble
15     if size[racineX] < size[racineY] then
16         parent[racineX] ← racineY
17         size[racineY] ← size[racineY] + size[racineX]
18     else
19         parent[racineY] ← racineX
20         size[racineX] ← size[racineX] + size[racineY]
```

Squelette en C++

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

class DisjointSet {
public:
    vector<int> parent, size;

    DisjointSet(int n);

    int findUPar(int node);

    void unionBySize(int u, int v);

    int getSize(int u);
};

int main() {
    DisjointSet dsu(5);
    dsu.unionBySize(1, 2);
    dsu.unionBySize(3, 4);
    dsu.unionBySize(2, 3);
    cout << (dsu.findUPar(1) == dsu.findUPar(4)) << "\n";
    return 0 ;
}
```

Squelette en Python

```
class DisjointSet:
    def __init__(self, n):
        pass

    def findUPar(self, node):
        pass

    def unionBySize(self, u, v):
        pass

     def getSize(self, u):
      pass

dsu = DisjointSet(5)
dsu.union(1, 2)
dsu.union(3, 4)
dsu.union(2, 3)
print(dsu.find(1) == dsu.find(4))
```

Exercice 1: Paires de sommets dans un arbre pondéré

On vous donne un arbre pondéré à n sommets. Rappelons qu'un arbre est un graphe connexe sans cycles. Les sommets u_i et v_i sont reliés par une arête de poids w_i.

Vous recevez ensuite m requêtes. La i-ème requête est donnée sous la forme d'un entier q_i. Dans cette requête, il faut calculer le nombre de paires de sommets (u, v) avec $u < v$ telles que le poids maximum d'une arête sur le chemin simple entre u et v ne dépasse pas q_i.

Entrée :

- La première ligne contient deux entiers n, m ($1 \leq n, m \leq 2 \cdot 10^5$).
- Les $n-1$ lignes suivantes décrivent chacune une arête sous la forme de trois entiers u_i, v_i, w_i ($1 \leq u_i, v_i \leq n$, $u_i \neq v_i$, $1 \leq w_i \leq 2 \cdot 10^5$).
- La dernière ligne contient m entiers $q_1, q_2, \ldots, q_m$ ($1 \leq q_i \leq 2 \cdot 10^5$).

Sortie : Affichez m entiers – les réponses aux requêtes. La i-ème valeur doit être égale au nombre de paires de sommets (u, v) telles que le poids maximum sur le chemin de u à v est au plus q_i.

2 Sommes préfixes

Les **sommes préfixes** sont une technique fondamentale en algorithmique permettant de calculer efficacement des sommes de sous-tableaux ou de sous-séquences. Elles consistent à construire un tableau auxiliaire P tel que :

$$P[i] = \sum_{j=1}^{i} a_j$$

où $a_1, \ldots, a_n$ sont les éléments du tableau initial.

Une fois ce tableau construit, on peut obtenir la somme de n'importe quel segment $[l, r]$ en temps constant :

$$\text{sum}(l, r) = P[r] - P[l-1].$$

La construction de P se fait en temps linéaire $O(n)$, et chaque requête de somme devient $O(1)$, ce qui est particulièrement utile pour les problèmes nécessitant de nombreuses requêtes de ce type.

Exemple 1 : Trouver un indice équilibré On cherche un indice i tel que la somme des éléments à gauche soit égale à la somme des éléments à droite :

$$\sum_{j=1}^{i-1} a_j = \sum_{j=i+1}^{n} a_j.$$

Grâce aux sommes préfixes, cette condition se réécrit simplement :

$$P[i-1] = P[n] - P[i].$$

Exemple 2 : Trouver la clique maximale dans un graphe d'intervalles Un graphe d'intervalles est un graphe dont les sommets correspondent à des intervalles sur la droite réelle, et où deux sommets sont adjacents si leurs intervalles se chevauchent.

Le **nombre maximal d'intervalles se chevauchant** en un point correspond exactement à la **taille de la clique maximale** du graphe d'intervalles.

On peut calculer cela efficacement en utilisant un tableau de *différences*, qui repose sur le principe des sommes préfixes. L'idée consiste à, pour chaque intervalle, ajouter $+1$ à l'indice de son début et -1 à sa fin dans le tableau de différences. La somme préfixe des différences donne,

pour chaque point, le nombre d'intervalles actifs. Par exemple, pour les intervalles $[1, 3]$, $[2, 5]$, $[4, 6]$, le tableau des différences est :

$$\text{diff} = [0, +1, +1, -1, +1, -1, -1].$$

Les sommes préfixes donnent $[0, 1, 2, 1, 2, 1, 0]$. Le maximum vaut 2, donc la clique maximale a taille 2.

Exercice 2: Coffee break

Alice, grande amatrice de café, souhaite déterminer la température optimale pour préparer la tasse parfaite. Elle connaît n recettes de café. La i-ème recette recommande une température comprise entre l_i et r_i degrés inclus.
Alice pense qu'une température est *admissible* si elle est recommandée par au moins k recettes.
Elle pose q questions : pour chaque question, on lui donne un intervalle $[a, b]$ et elle veut savoir combien de températures entières admissibles se trouvent dans cet intervalle.

Entrée :

- La première ligne contient trois entiers n, k, q ($1 \leq k \leq n \leq 2 \cdot 10^5$, $1 \leq q \leq 2 \cdot 10^5$).
- Les n lignes suivantes contiennent deux entiers l_i, r_i ($1 \leq l_i \leq r_i \leq 200000$), l'intervalle de températures recommandé par la i-ème recette.
- Les q lignes suivantes contiennent deux entiers a, b ($1 \leq a \leq b \leq 200000$), décrivant l'intervalle de la question.

Sortie :
Pour chaque question, afficher le nombre de températures entières admissibles dans $[a, b]$.

Exercice 3: Combien de triangles ?

Comme tout mathématicien inconnu, Yuri a des nombres favoris : A, B, C et D, avec $A \leq B \leq C \leq D$. Yuri aime également les triangles et s'est un jour demandé : combien de triangles non dégénérés à côtés entiers x, y et z existent, tels que

$$A \leq x \leq B \leq y \leq C \leq z \leq D\,?$$

Yuri prépare actuellement des problèmes pour un nouveau concours et est donc très occupé. C'est pourquoi il vous demande de calculer le nombre de triangles correspondant à cette propriété.

Un triangle est dit *non dégénéré* si et seulement si ses sommets ne sont pas alignés, c'est-à-dire si la condition $x + y > z$ est vérifiée.

Entrée : La première ligne contient quatre entiers A, B, C et D tels que $1 \leq A \leq B \leq C \leq D \leq 5 \cdot 10^5$.

Sortie : Affichez le nombre de triangles non dégénérés à côtés entiers x, y et z tels que l'inégalité $A \leq x \leq B \leq y \leq C \leq z \leq D$ soit respectée.