Ben Youssef Nour
Chedi Khawla

TP Algorithmique

Exercice 1 :

- Objectif :

Étant donné une séquence d'entiers A = a [1.n] (avec $n \leq 5000$ et $-10000 \leq a[i] \leq 10000$), il faut identifier la plus longue sous-séquence strictement croissante. Une sous-séquence est une suite formée à partir d'éléments de A, tout en respectant leur ordre d'origine.

Par exemple, si A = (1, 2, 3, 4, 9, 10, 5, 6, 7), la plus longue sous-séquence strictement croissante est (1, 2, 3, 4, 5, 6, 7).

Le but est de trouver cette sous-séquence croissante la plus longue.

- Méthode de travail :

Le programme lit une séquence d'entiers à partir d'un fichier INPMONOSEQ.TXT et trouve la plus longue sous-séquence croissante (LIS) à l'aide d'un algorithme de programmation dynamique.

Tout d'abord, on ouvre le fichier et vérifie si celui-ci existe. Si le fichier est introuvable, le programme affiche un message d'erreur et s'arrête. Il lit en premier la taille de la séquence d'entiers ensuite la séquence d'entiers, puis calcule la plus longue sous-séquence croissante.

Pour cela, il utilise deux tableaux : « LIS » pour stocker la longueur de la LIS à chaque position, et « prev » pour garder la trace des prédécesseurs d'un élément dans cette sous-séquence. À la fin, le programme reconstruit le résultat en suivant les prédécesseurs et la stocke dans un tableau.

Finalement, il écrit dans un fichier de sortie OUTMONOSEQ.TXT la longueur de la LIS ainsi que chaque élément de cette sous-séquence, avec ses indices d'origine.

Voici un exemple pour mieux comprendre la méthode qu'on a utilisé pour résoudre ce problème :

1. Initialisation : n = 7

- Tableau A : (la plus longue séquence prévue : 0,1,4,9,10)

0	**1**	**4**	**9**	**2**	**10**	**5**

- Tableau LIS : (stocker la longueur)

1	**1**	**1**	**1**	**1**	**1**	**1**

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**

2. <u>Première itération :</u>

   Tableau A :

j	i					
0	**1**	**4**	**9**	**2**	**10**	**5**

```
// Remplissage de LIS avec une approche dynamique
for (int i = 1; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        if (A[i] > A[j] && LIS[i] < LIS[j] + 1) {
            LIS[i] = LIS[j] + 1;
            prev[i] = j;
        }
    }
}
```

- Tableau LIS : (stocker la longueur)

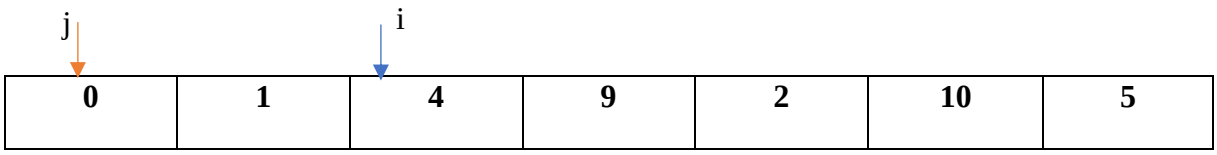
1	**2**	**1**	**1**	**1**	**1**	**1**

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	**0**	**-1**	**-1**	**-1**	**-1**	**-1**

3. Deuxième itération

Tableau A :



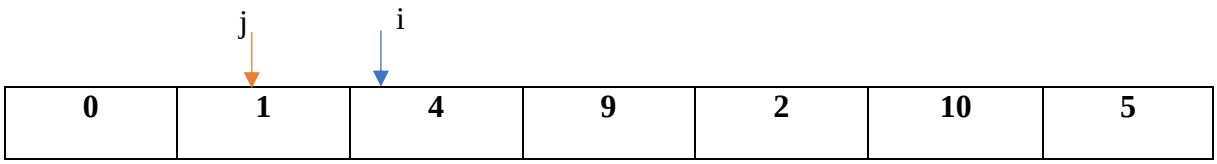
- Tableau LIS : (stocker la longueur)

1	**2**	**2**	**1**	**1**	**1**	**1**

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	**0**	**0**	**-1**	**-1**	**-1**	**-1**

Tableau A :



- Tableau LIS : (stocker la longueur)

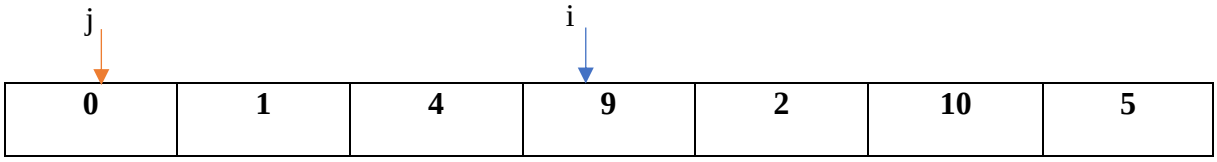
1	**2**	**3**	**1**	**1**	**1**	**1**

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	**0**	**1**	**-1**	**-1**	**-1**	**-1**

4. <u>Troisième itération</u>

Tableau A :



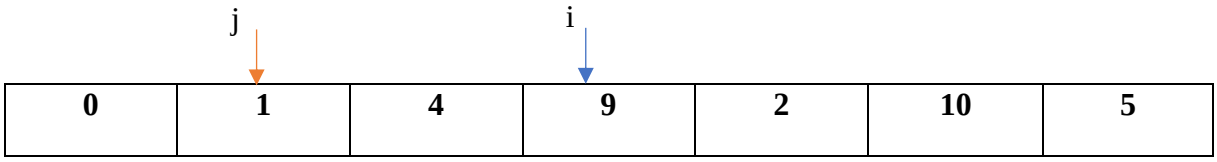
- Tableau LIS : (stocker la longueur)

1	2	3	2	1	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	0	-1	-1	-1

Tableau A :



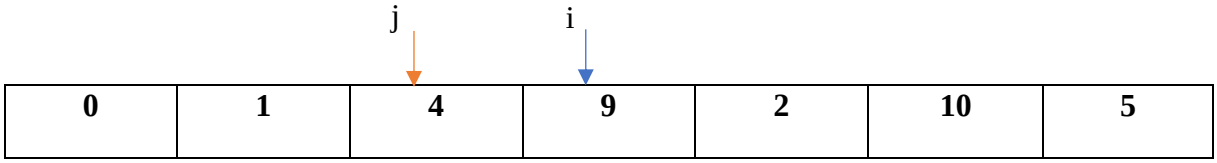
- Tableau LIS : (stocker la longueur)

1	2	3	3	1	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	1	-1	-1	-1

Tableau A :



- Tableau LIS : (stocker la longueur)

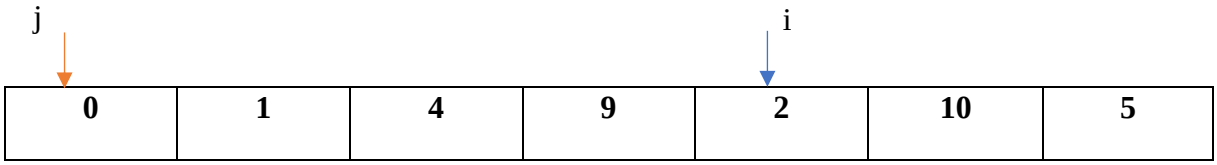
1	2	3	4	1	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	-1	-1	-1

5. Quatrième itération

Tableau A :



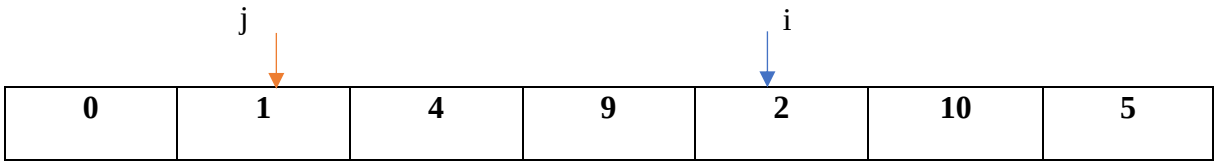
- Tableau LIS : (stocker la longueur)

1	2	3	4	2	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	0	-1	-1

Tableau A :



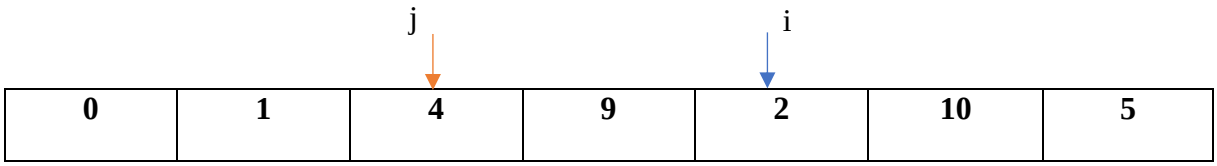
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	-1	-1

Tableau A :



- Tableau LIS : (stocker la longueur)

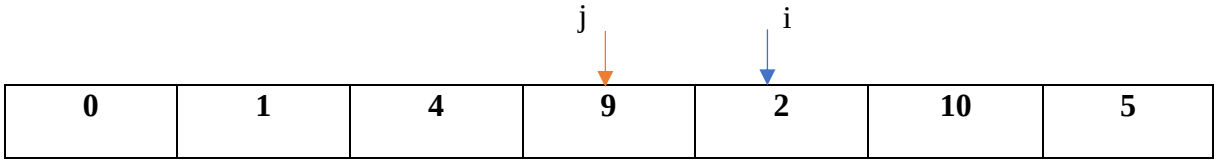
1	2	3	4	3	1	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	-1	-1

➔ Ne rien faire car A[j] > A[i] on passe au suivant

Tableau A :



- Tableau LIS : (stocker la longueur)

1	2	3	4	3	1	1

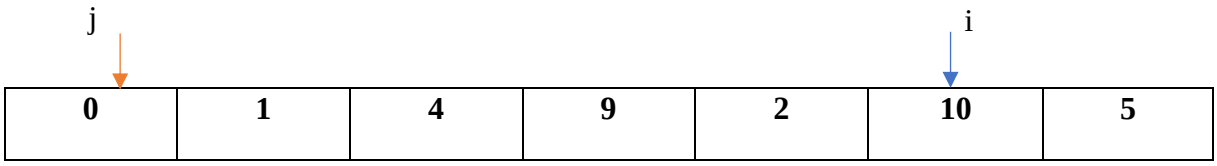
- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	-1	-1

➔ Ne rien faire car A[j] > A[i] on passe au suivant

6. <u>Cinquième itération :</u>

Tableau A :



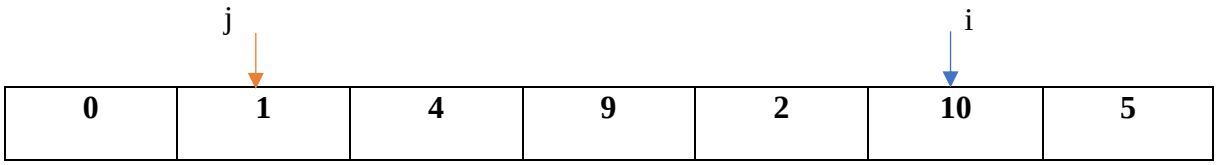
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	2	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	0	-1

Tableau A :



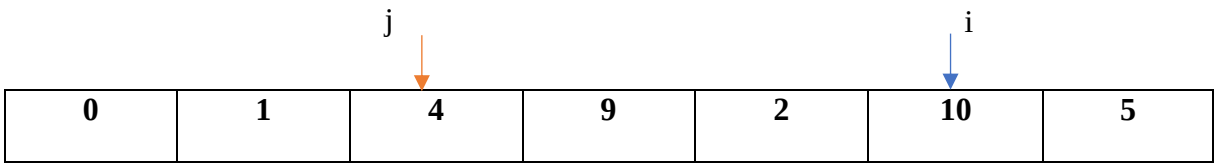
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	3	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	1	-1

Tableau A :



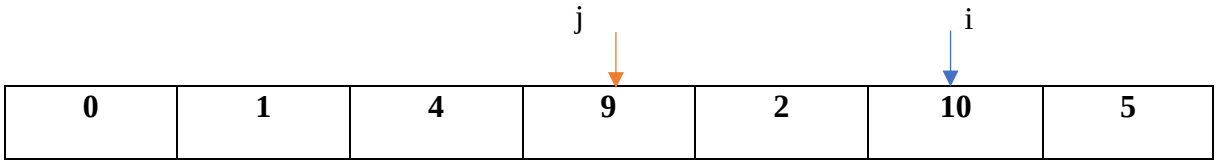
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	4	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	2	-1

Tableau A :



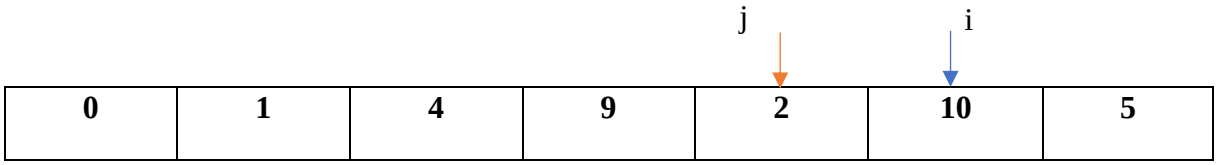
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	1

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	-1

Tableau A :



- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	1

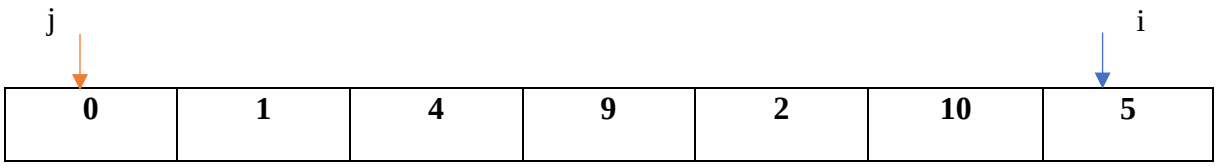
- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	-1

➔ Rien faire car c'est vrai que A[i] > A [j] mais LIS[i] > LIS [j] +1

7. Sixième itération

Tableau A :



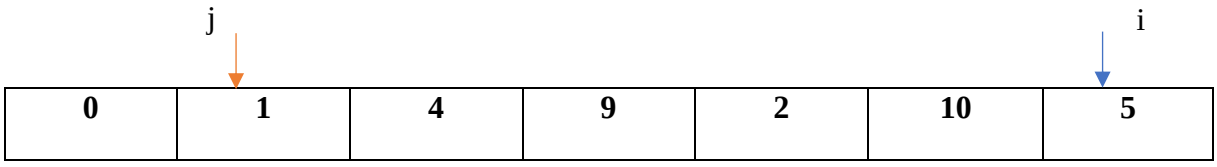
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	2

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	0

Tableau A :



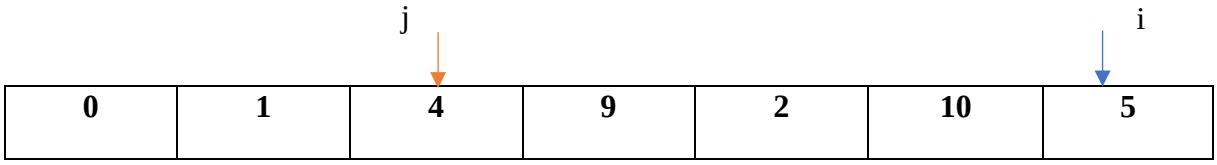
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	3

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	1

Tableau A :



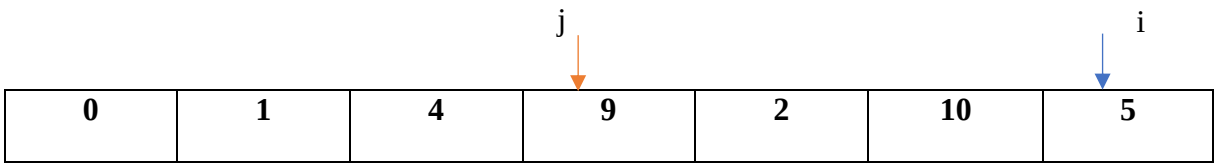
- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	4

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	2

Tableau A :



- Tableau LIS : (stocker la longueur)

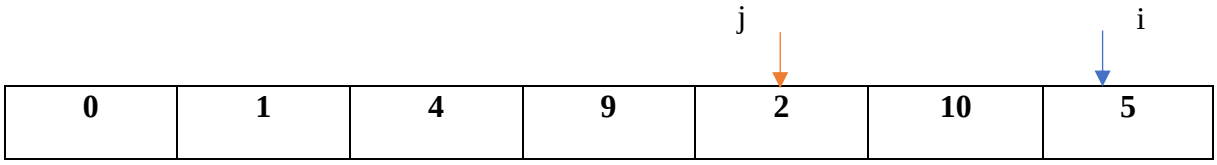
1	2	3	4	3	5	4

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	2

➔ Ne rien faire car A[j] > A[i] on passe au suivant

Tableau A :



- Tableau LIS : (stocker la longueur)

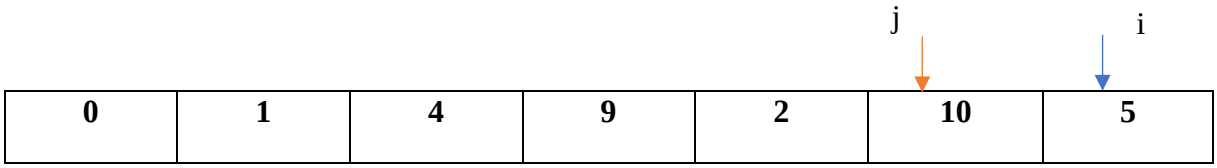
1	2	3	4	3	5	4

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	2

➔ Rien faire car c'est vrai que A[i] > A [j] mais LIS[i] > LIS [j] +1

Tableau A :



- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	4

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	2

➔ Ne rien faire car A[j] > A[i] on passe au suivant

- Final :

  Tableau A :

0	1	4	9	2	10	5

- Tableau LIS : (stocker la longueur)

1	2	3	4	3	5	4

- Tableau prev : (garder la trace des prédécesseurs d'un élément dans cette sous-séquence)

-1	0	1	2	1	3	2

La longueur de la plus longue séquence est le max du tableau LIS : 5 on doit garder son indice (position dans le tableau LIS) c'est la première valeur à ajouter dans le résultat souhaité A[5]

```
// Trouver la longueur maximale de la sous-séquence croissante
int maxLength = 0, maxIndex = 0;
for (int i = 0; i < n; ++i) {
    if (LIS[i] > maxLength) {
        maxLength = LIS[i];
        maxIndex = i;
    }
}
```

- Résultat

10	**0**	**0**	**0**	**0**	**0**	**0**

- RésultatIndice

6	**0**	**0**	**0**	**0**	**0**	**0**

Dans le tableau prev, prev [5] =3 dans le prochain élément à ajouter et A [3]

- Résultat

10	**9**	**0**	**0**	**0**	**0**	**0**

- RésultatIndice

6	**4**	**0**	**0**	**0**	**0**	**0**

Dans le tableau prev, prev [3] =2 dans le prochain élément à ajouter et A [2]

- Résultat

10	**9**	**4**	**0**	**0**	**0**	**0**

- RésultatIndice

6	**4**	**3**	**0**	**0**	**0**	**0**

Dans le tableau prev, prev [2] =1 dans le prochaine élément à ajouter et A[1]

- Résultat

10	**9**	**4**	**1**	**0**	**0**	**0**

- RésultatIndice

6	**4**	**3**	**2**	**0**	**0**	**0**

Dans le tableau prev, prev [1] = 0 dans le prochaine élément à ajouter et A[0]

- Résultat

10	**9**	**4**	**1**	**0**	**0**	**0**

- RésultatIndice

6	**4**	**3**	**2**	**1**	**0**	**0**

Dans le tableau prev, prev [0] = -1, On s'arrête

```
// Reconstruire la sous-séquence croissante en partant de l'indice maxIndex
int index = maxIndex;
while (index != -1) {
    result.push_back(A[index]);
    resultIndices.push_back(index + 1);
    index = prev[index];
}
```

A la fin on doit inverser la sous-séquence (Résultat et RésultatIndice) pour obtenir l'ordre croissant. Finalement, il écrit dans un fichier de sortie OUTMONOSEQ.TXT la longueur du résultat ainsi que chaque élément de cette sous-séquence (Résultat), avec ses indices d'origine (RésultatIndice)

Exercice 2 :

- Objectif :

L'objectif de cet exercice est de trouver la sous-séquence avec le plus grand nombre d'éléments dans une séquence d'entiers positifs, telle que la somme des éléments de cette sous-séquence soit divisible par un entier *k*. Le but est de maximiser la taille de la sous-séquence tout en respectant cette contrainte de divisibilité.

Exemple : Supposons que nous ayons une séquence de $n=10$ entiers :

1, 6, 11, 5, 10, 15, 20, 2, 4, 9 et $k=5$. Il faut trouver la plus longue sous-séquence dont la somme est divisible par 5.

➔ Solution : La sous-séquence suivante satisfait la condition : *6, 11, 5, 10, 15, 20, 4, 9*. La somme des éléments de cette sous-séquence est *80*, qui est divisible par *5*. La longueur de cette sous-séquence est *8*.

- Méthode de travail :

Ce programme a pour objectif de trouver la plus longue sous-séquence d'une séquence d'entiers positifs dont la somme est divisible par un entier donné k. La méthode de travail se divise en trois étapes principales : la lecture des données d'entrée, le calcul de la sous-séquence optimale, et l'écriture des résultats dans un fichier de sortie.

La première étape consiste à lire les données d'entrée via la méthode “readInput”. Le programme commence par ouvrir le fichier texte spécifié, qui contient les informations nécessaires à l'analyse. Le fichier comporte sur la première ligne un entier n représentant le nombre d'éléments dans la séquence. La deuxième ligne contient un entier k, qui est le diviseur cible. Enfin, la troisième ligne du fichier contient la séquence A composée de n entiers. Une fois les données lues, elles sont stockées dans un tableau pour être utilisées dans les calculs ultérieurs. Si le fichier ne peut pas être ouvert, le programme affiche un message d'erreur et s'arrête.

La seconde étape est réalisée par la méthode “findLongestDivisibleSubsequence”, qui met en œuvre un algorithme de programmation dynamique pour trouver la plus longue sous-séquence dont la somme est divisible par k. Pour cela, le programme utilise trois structures de données principales : un tableau **dp** pour stocker la longueur maximale de la sous-séquence pour chaque reste r (mod k), un tableau **sum** pour enregistrer la somme correspondante pour chaque valeur de dp, et un tableau de sous-séquences **subsequence** pour suivre les éléments sélectionnés.

L'algorithme parcourt chaque élément de la séquence A et met à jour les tables **dp**, **sum**, et **subsequence** en vérifiant pour chaque reste mod k si l'ajout de l'élément permet d'obtenir une sous-séquence plus longue.

À chaque itération, le programme copie les valeurs courantes dans de nouvelles structures (**newDp**, **newSum**, et **newSubsequence**) pour éviter d'écraser les données avant que toutes les mises à jour ne soient effectuées. Si un élément de la séquence peut être ajouté en tant que nouvelle sous-séquence divisible par k, il est directement intégré dans la solution.

La dernière étape est réalisée par la méthode “writeOutput”, qui écrit les résultats dans un fichier de sortie. Le programme commence par ouvrir le fichier de sortie et vérifie également son existence. Ensuite, il écrit la longueur de la plus longue sous-séquence trouvée, suivie des éléments de cette sous-séquence, avec leurs indices d'origine. Enfin, il calcule et écrit la somme totale de cette sous-séquence, qui doit être divisible par k. Si le fichier de sortie ne peut pas être créé ou ouvert, un message d'erreur est affiché.

Prenons un exemple encore plus simple avec k =7 et n = 4, où la séquence d'entiers A = [1, 2, 3, 4]. Nous allons détailler chaque itération de l'algorithme avec la mise à jour des tableaux à chaque étape.

- Initialisation :
  - **dp :** Un tableau de taille k = 7, initialisé à -1 sauf pour la case 0. Cela signifie que la sous-séquence vide a une somme divisible par 7.

0	-1	-1	-1	-1	-1	-1

- **sum** : Un tableau pour stocker les sommes correspondantes, initialisé à 0.

0	0	0	0	0	0	0

- **subsequence** : Un tableau pour stocker les sous-séquences correspondantes (tableau du tableau).

[]	[]	[]	[]	[]	[]	[]

```
// Table pour stocker la longueur maximale des sous-séque
std::vector<int> dp(k, -1);  // dp[i] représente la longu
std::vector<int> sum(k, 0);  // sum[i] représente la somm
std::vector<std::vector<int>> subsequence(k); // Pour gar

dp[0] = 0; // Pour la somme 0, la longueur est 0
```

- Itération 1 : num =1

Le reste de 1 mod 7 est 1.

Nous copions les tableaux dp, sum, et $subsequence$ avant les mises à jour.

Pour chaque reste r, nous vérifions s'il existe une sous-séquence pour r.

Pour l'instant, seule dp [0] est non-négative.

Nous ajoutons num =1 à la sous-séquence correspondant à r = 0, ce qui donne un nouveau reste $newR$ = (0 +1) mod 7 = 1.

Nous mettons à jour dp [1], sum [1], et $subsequence$ [1] avec une sous-séquence de longueur 1 contenant l'élément 1.

```
// Parcourir tous les éléments du tableau
for (int num : A) {
    // On fait une copie de dp pour éviter d'écraser les valeurs pendan
    std::vector<int> newDp = dp;
    std::vector<int> newSum = sum;
    std::vector<std::vector<int>> newSubsequence = subsequence;

    // Pour chaque reste possible, on tente d'ajouter l'élément `num`
    for (int r = 0; r < k; ++r) {
        if (dp[r] != -1) {  // Si une sous-séquence existe pour ce rest
            int newR = (r + num % k + k) % k;

            // Mettre à jour dp[newR] si cela permet d'obtenir une sous
            if (newDp[newR] < dp[r] + 1) {
                newDp[newR] = dp[r] + 1;
                newSum[newR] = sum[r] + num;
                newSubsequence[newR] = subsequence[r];
                newSubsequence[newR].push_back(num);
            }
        }
    }

    // Ajout du nouvel élément en tant que nouvelle sous-séquence indép
    int newR = num % k;
    if (newDp[newR] < 1) {
        newDp[newR] = 1;
        newSum[newR] = num;
        newSubsequence[newR] = { num };
    }

    // Remplacer dp et sum par les nouveaux résultats
    dp = newDp;
    sum = newSum;
    subsequence = newSubsequence;
}
```

- **dp**

0	1	-1	-1	-1	-1	-1

- **sum**

0	1	0	0	0	0	0

- **subsequence**

[]	[1]	[]	[]	[]	[]	[]

- Itération 2 : num = 2

Le reste de 2 mod 7 est 2.

Nous copions les tableaux avant les mises à jour.

Pour r = 0, ajouter num = 2 donne $newR$ = (0 + 2) mod 7 = 2.

Nous mettons à jour dp [2], sum [2], et $subsequence$ [2] avec une nouvelle sous-séquence de longueur 1 contenant l'élément 2.

Pour r =1, ajouter num = 2 donne $newR$ = (1 + 2) mod 7 =3.

Nous mettons à jour dp [3], sum [3], et $subsequence$ [3] avec une nouvelle sous-séquence de longueur 2 contenant [1,2].

- **dp**

0	1	1	2	-1	-1	-1

- **sum**

0	1	2	3	0	0	0

- **subsequence**

[]	[1]	[2]	[1,2]	[]	[]	[]

- Itération 3 : num =3

Le reste de 3 mod 7 est 3.

Nous copions les tableaux avant les mises à jour.

Pour r = 0, ajouter num = 3 donne $newR$= (0+3) mod 7 = 3, mais cela ne permet pas d'obtenir une sous-séquence plus longue que dp [3].

Pour r=1, ajouter num = 3 donne $newR$ = (1+3) mod 7 = 4.

Nous mettons à jour dp [4], sum [4], et $subsequence$ [4] avec une nouvelle sous-séquence de longueur 2 contenant [1,3].

Pour r = 2, ajouter num = 3 donne $newR$ = (2+3) mod 7 = 5.

Nous mettons à jour dp [5], sum [5], et $subsequence$ [5] avec une nouvelle sous-séquence de longueur 2 contenant [2,3].

Pour r = 3, ajouter num = 3 donne $newR$ = (3+3) mod 7 = 6, créant une nouvelle sous-séquence de longueur 3 contenant [1,2,3].

- **dp**

0	1	1	2	2	2	3

- **sum**

0	1	2	3	4	5	6

- **subsequence**

[]	[1]	[2]	[1,2]	[1,3]	[2,3]	[1,2,3]

- Itération 4 : num = 4

Le reste de 4 mod 7 est 4.

Nous copions les tableaux avant les mises à jour.

Pour r=0, ajouter num=4 donne $newR$= (0+4) mod 7 = 4, mais cela ne permet pas d'obtenir une sous-séquence plus longue que dp [4].

Pour r=1, ajouter num = 4 donne $newR$ = (1+4) mod 7 = 5, mais cela ne permet pas d'obtenir une sous-séquence plus longue que dp [5].

Pour r=2, ajouter num = 4 donne $newR$ = (2+4) mod 7 = 6, mais cela ne permet pas d'obtenir une sous-séquence plus longue que dp [6].

Pour r=3, ajouter num = 4 donne $newR$ = (3+4) mod 7 = 0, créant une sous-séquence de longueur 3 contenant [1,2,4].

Pour r=4, ajouter num = 4 donne $newR$ = (4+4) mod 7 =1, mais cela ne permet pas d'obtenir une sous-séquence plus longue.

Pour r=5, ajouter num = 4 donne $newR$ = (5+4) mod 7 = 2, mais cela ne permet pas d'obtenir une sous-séquence plus longue.

Pour r=6, ajouter num = 4 donne $newR$ = (6+4) mod 7 = 3, mais cela ne permet pas d'obtenir une sous-séquence plus longue.

- **dp**

3	1	1	2	2	2	3

- **sum**

7	1	2	3	4	5	6

- **subsequence**

[1,2,4]	[1]	[2]	[1,2]	[1,3]	[2,3]	[1,2,3]

La plus longue sous-séquence dont la somme est divisible par 7 est de longueur 3 avec la somme 7, et la sous-séquence est [1,2,4].

Exercice 3 :

- Objectif :

L'objectif de cet exercice est de développer un algorithme permettant de trouver un arbre couvrant pour un graphe non orienté et connexe $G = (V,E)$. Un arbre couvrant est un sous-ensemble d'arêtes du graphe qui relie tous les sommets sans former de cycles, tout en étant un arbre. Cet exercice implique deux représentations différentes du graphe :

- **Matrice d'adjacence** : Une matrice de dimension $n \times n$ où l'élément a_{ij} représente le nombre d'arêtes reliant les sommets i et j. Les éléments diagonaux a_{ii} sont tous égaux à 0.
- **Liste d'adjacence** : Un tableau de listes où chaque liste i contient les sommets adjacents au sommet i.

L'algorithme doit être capable de construire l'arbre couvrant à partir des deux structures de données mentionnées, démontrant ainsi la flexibilité et l'efficacité de l'algorithme selon la représentation choisie. La sortie de l'algorithme sera une liste des arêtes qui composent l'arbre couvrant trouvé, permettant ainsi d'analyser et de vérifier la validité de l'arbre.

- Méthode de travail pour la matrice d'adjacence :

Le code présenté implémente l'algorithme de Kruskal pour trouver un arbre couvrant d'un graphe non orienté à l'aide d'une représentation par matrice d'adjacence. Il commence par inclure les en-têtes nécessaires et définit deux classes, **UnionFind** et **GraphMatrix**. La classe **UnionFind** est utilisée pour gérer la structure de données d'union-find, permettant de détecter les cycles lors de l'ajout d'arêtes. Le constructeur de UnionFind initialise les parents et les rangs pour chaque sommet. La méthode find est utilisée pour trouver le représentant d'un sommet, et la méthode unite fusionne deux ensembles.

La classe **GraphMatrix** est conçue pour représenter un graphe à l'aide d'une matrice d'adjacence. Son constructeur prend en entrée le nombre de sommets n et initialise une matrice adj de taille (n+1) x (n+1) à zéro. La méthode addEdge ajoute une arête entre deux sommets u et v en mettant à jour la matrice d'adjacence, ce qui permet de garder une trace des connexions entre les sommets. Les arêtes sont représentées par des éléments non nuls dans la matrice, indiquant la présence d'une connexion entre les sommets.

L'algorithme de Kruskal est implémenté dans la méthode **kruskalAlgorithm**. Cette méthode commence par créer une instance de UnionFind pour gérer les ensembles de sommets. Ensuite, elle récupère toutes les arêtes à partir de la matrice d'adjacence, en itérant sur les indices de la matrice et en ajoutant les paires de sommets connectés à un vecteur d'arêtes. Les arêtes sont ensuite triées pour déterminer leur ordre d'ajout, bien que dans ce cas, il n'y ait pas de poids à prendre en compte.

L'algorithme procède à l'ajout des arêtes en vérifiant, pour chaque arête, si elle peut être ajoutée sans créer un cycle. Cela est réalisé en utilisant les méthodes find et unite de l'objet UnionFind. Si les deux sommets d'une arête appartiennent à des ensembles différents, l'arête est ajoutée à l'arbre couvrant, et les ensembles correspondants sont fusionnés. Ce processus continue jusqu'à ce que toutes les arêtes aient été examinées. À la fin, le vecteur treeEdges contient les arêtes de l'arbre couvrant trouvé.

La fonction **readGraphMatrix** lit les données d'un fichier d'entrée spécifié, construit le graphe en ajoutant les arêtes à l'aide de la méthode addEdge, et retourne une instance de GraphMatrix. Enfin, la fonction writeOutput prend le fichier de sortie et le vecteur d'arêtes de l'arbre couvrant pour écrire les résultats dans le fichier. Chaque arête est écrite dans une nouvelle ligne, ce qui fournit une représentation claire des connexions dans l'arbre couvrant.

Dans l'ensemble, le code met en œuvre de manière efficace l'algorithme de Kruskal en utilisant une matrice d'adjacence pour représenter le graphe et une structure d'union-find pour gérer la détection des cycles, permettant ainsi de trouver un arbre couvrant en temps efficace.

- **Exemple de graphe par la matrice d'adjacence :**
  - Sommets (n) : 4
  - Arêtes (m) : 5
  - Arêtes :

    (1, 2)

    (1, 3)

    (2, 3)

    (2, 4)

    (3, 4)

La matrice d'adjacence pour ce graphe sera :

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	1
3	0	1	1	0	1
4	0	1	1	1	0

- **Étapes d'exécution de l'algorithme de Kruskal**

1. Initialisation :
   - Créer une instance de UnionFind pour 4 sommets.
   - Créer un vecteur edges pour stocker toutes les arêtes.
2. Récupération des arêtes :
   - Parcourir la matrice d'adjacence et ajouter les arêtes dans le vecteur edges :
     - (1, 2)
     - (1, 3)
     - (2, 3)
     - (2, 4)
     - (3, 4)
3. Tri des arêtes :
   - Comme les arêtes n'ont pas de poids, nous les gardons dans l'ordre où elles sont trouvées.
4. Construction de l'arbre couvrant :
   - Pour chaque arête, vérifier si elle peut être ajoutée sans créer de cycle :
     - Ajouter (1, 2) : Trouver leurs racines : 1 et 2 (pas de cycle, ajoute).
     - Ajouter (1, 3) : Racines : 1 et 3 (pas de cycle, ajoute).
     - Ajouter (2, 3) : Racines : 1 et 3 (cycle détecté, ignore).
     - Ajouter (2, 4) : Racines : 1 et 4 (pas de cycle, ajoute).

- Ajouter (3, 4) : Racines : 1 et 4 (cycle détecté, ignore).

Résultat final

Les arêtes de l'arbre couvrant sont :

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

Ces arêtes sont ensuite écrites dans le fichier de sortie.

- Méthode de travail pour la liste d'adjacence :

Le code implémente un algorithme pour trouver un arbre couvrant d'un graphe non orienté en utilisant une représentation par liste d'adjacence. La classe **GraphList** est définie pour gérer cette représentation. Dans le constructeur de la classe, on initialise une liste d'adjacence, qui est un vecteur de vecteurs d'entiers. Chaque élément de cette liste représente un sommet et contient les sommets adjacents. L'indexation commence à 1 pour correspondre à la manière dont les sommets sont généralement référencés dans les graphes.

La méthode **addEdge** permet d'ajouter une arête entre deux sommets, u et v. Cette méthode modifie les listes d'adjacence en ajoutant v à la liste de u et u à la liste de v, établissant ainsi une connexion bidirectionnelle entre les deux sommets. Cela permet de conserver une structure de données qui reflète la nature non orientée du graphe.

L'algorithme de Prim est implémenté dans la méthode **primAlgorithm**, qui commence par initialiser un vecteur visited pour suivre les sommets déjà explorés et un vecteur treeEdges pour stocker les arêtes de l'arbre couvrant. Une queue est utilisée pour gérer les sommets à explorer, et le sommet 1 est initialement ajouté à cette queue. L'algorithme fonctionne en extrayant un sommet de la queue, puis en parcourant tous ses voisins. Pour chaque voisin non visité, l'arête entre le sommet courant et le voisin est ajoutée à treeEdges, et le voisin est marqué comme visité. Cette opération se répète jusqu'à ce que tous les sommets accessibles soient explorés, garantissant ainsi que l'arbre couvrant contient tous les sommets du graphe tout en minimisant le nombre d'arêtes.

Enfin, deux fonctions sont définies en dehors de la classe. La première, **readGraphList**, lit un fichier d'entrée pour construire le graphe en ajoutant des arêtes basées sur les données du fichier. La seconde, **writeOutput**, écrit les arêtes de l'arbre couvrant dans un fichier de sortie, permettant de visualiser le résultat de l'algorithme. Cette approche permet de résoudre efficacement le problème de l'arbre couvrant en utilisant une structure de données appropriée pour les graphes peu denses.

- **Exemple de graphe par la liste d'adjacence :**
  - **Étape 1 : Lecture du fichier et création du graphe**

Le fichier indique qu'il y a 5 sommets et 6 arêtes. Les arêtes sont les suivantes :

(1, 2)
(1, 3)
(2, 3)
(2, 4)
(3, 4)
(4, 5)

Après avoir lu ce fichier, le programme construit le graphe en ajoutant les arêtes dans la structure de données de liste d'adjacence.

  - **Étape 2 : Représentation du graphe**

Après l'ajout des arêtes, la liste d'adjacence du graphe sera :

1 : [2, 3]
2 : [1, 3, 4]
3 : [1, 2, 4]
4 : [2, 3, 5]
5 : [4]

  - **Étape 3 : Application de l'algorithme de Prim**

Nous allons maintenant appliquer l'algorithme de Prim pour trouver un arbre couvrant. Voici les étapes détaillées :

Initialisation :

Un vecteur **visited** de taille 6 (0 à 5) pour suivre les sommets visités : [false, false, false, false, false, false]

Un vecteur **treeEdges** pour stocker les arêtes de l'arbre couvrant : []

Une queue contenant le sommet de départ (1) : [1]

Marquer le sommet 1 comme visité : [false, true, false, false, false, false]

Itération 1 :

- Déqueue le sommet 1.
- Parcourir ses voisins (2 et 3).
    - Pour 2 :
        - Ajouter l'arête (1, 2) à treeEdges.
        - Marquer 2 comme visité : [false, true, true, false, false, false]
        - Enfiler 2 dans la queue : [2]
    - Pour 3 :
        - Ajouter l'arête (1, 3) à treeEdges.
        - Marquer 3 comme visité : [false, true, true, true, false, false]
        - Enfiler 3 dans la queue : [2, 3]

Itération 2 :

- Déqueue le sommet 2.
- Parcourir ses voisins (1, 3, 4).
    - Pour 1 : déjà visité.
    - Pour 3 : déjà visité.
    - Pour 4 :
        - Ajouter l'arête (2, 4) à treeEdges.
        - Marquer 4 comme visité : [false, true, true, true, true, false]
        - Enfiler 4 dans la queue : [3, 4]

Itération 3 :

- Déqueue le sommet 3.
- Parcourir ses voisins (1, 2, 4).

- Pour 1 : déjà visité.
- Pour 2 : déjà visité.
- Pour 4 : déjà visité.

Itération 4 :

- Déqueue le sommet 4.
- Parcourir ses voisins (2, 3, 5).
  - Pour 2 : déjà visité.
  - Pour 3 : déjà visité.
  - Pour 5 :
    - Ajouter l'arête (4, 5) à treeEdges.
    - Marquer 5 comme visité : [false, true, true, true, true, true]
    - Enfiler 5 dans la queue : [5]

Itération 5 :

- Déqueue le sommet 5.
- Parcourir son voisin (4).
  - Pour 4 : déjà visité.

➔ **Résultat Final**

À la fin de l'exécution de l'algorithme, les arêtes de l'arbre couvrant trouvées dans treeEdges sont :

- (1, 2)
- (1, 3)
- (2, 4)
- (4, 5)

Exercice 4 :

- Objectif :

L'exercice consiste à analyser un graphe non orienté pour déterminer sa connexité, c'est-à-dire la possibilité d'atteindre un sommet à partir d'un autre par une chaîne d'arêtes.

Contexte : Un graphe non orienté est une collection de sommets et d'arêtes où les arêtes n'ont pas de direction. Un graphe est dit connexe si, pour chaque paire de sommets, il existe un chemin qui les relie. Un sous-graphe connexe maximal, ou composante connexe, est un sous-ensemble de sommets qui sont tous reliés entre eux par des chemins, mais qui ne peuvent pas être étendus en ajoutant d'autres sommets du graphe.

L'objectif est de :

- Déterminer si le graphe est connexe : Cela signifie qu'il n'existe qu'une seule composante connexe.
- Si le graphe n'est pas connexe, identifier et lister toutes les composantes connexes du graphe. Cela implique de trouver tous les sous-ensembles de sommets qui sont interconnectés sans connexions aux autres sommets du graphe.

Entrée : Le programme doit lire les informations d'un fichier texte au format suivant :

- La première ligne contient deux entiers n (le nombre de sommets) et m (le nombre d'arêtes).
- Les lignes suivantes contiennent des paires d'entiers représentant les arêtes du graphe, où chaque paire indique une connexion entre deux sommets.

Sortie : Le programme doit écrire les résultats dans un fichier texte, incluant :

- Le nombre de composantes connexes k.
- Pour chaque composante connexe, il doit indiquer le numéro de la composante et les indices des sommets qui la composent.

Pour atteindre cet objectif, on doit :

1. Lire le graphe à partir du fichier d'entrée et le représenter sous deux formes :
    - **Matrice d'adjacence** : Une matrice carrée où l'élément (i, j) est 1 si une arête relie les sommets i et j, sinon 0.

- **Liste d'adjacence** : Un tableau de listes où chaque liste contient les sommets adjacents à un sommet donné.

2. Parcourir le graphe à l'aide de méthodes de recherche :
   - Recherche en profondeur (DFS)
3. Identifier les composantes connexes : Commencer à partir d'un sommet non visité et marquer tous les sommets accessibles jusqu'à ce qu'il n'y ait plus de sommets à explorer. Répéter ce processus jusqu'à ce que tous les sommets aient été visités.

- Méthode de travail pour matrice d'adjacence :

Le code présenté implémente un algorithme pour déterminer les composantes connexes d'un graphe non orienté en utilisant une matrice d'adjacence. La classe **GraphMatrix** est initialisée avec un nombre de sommets n, et une matrice d'adjacence de dimensions (n + 1) x (n + 1) est créée, initialisée à zéro, ce qui signifie qu'aucune arête n'existe initialement. La méthode **addEdge** permet d'ajouter une arête entre deux sommets u et v en mettant à jour les valeurs correspondantes dans la matrice d'adjacence à 1, indiquant ainsi la présence d'une arête entre ces sommets.

L'exploration des sommets est effectuée à l'aide de la méthode de recherche en profondeur (DFS) dans la fonction dfs. Cette fonction prend comme paramètres le sommet actuel v, un vecteur visited qui garde la trace des sommets déjà visités, et un vecteur component qui stocke les sommets de la composante connexe en cours de formation. Lorsque le sommet est visité, il est ajouté au vecteur component, et la fonction explore récursivement tous ses voisins non visités. Ce processus continue jusqu'à ce que tous les sommets connectés à v aient été visités.

La méthode **findConnectedComponents** gère l'ensemble du processus de détection des composantes connexes. Elle initialise un vecteur visited pour suivre les sommets visités et un vecteur de vecteurs components pour stocker toutes les composantes trouvées. En itérant sur tous les sommets, si un sommet n'a pas encore été visité, la méthode appelle dfs pour explorer cette composante et ajouter les résultats au vecteur components. Une fois toutes les composantes détectées, le résultat est écrit dans un fichier de sortie spécifié, incluant le nombre total de composantes connexes et les sommets de chaque composante, formatés de manière lisible. Enfin, la fonction **readGraphMatrix** lit un fichier d'entrée contenant la description du

graphe, construit la matrice d'adjacence en ajoutant les arêtes spécifiées, et renvoie une instance de GraphMatrix prête à être utilisée pour trouver les composantes connexes.

Pour illustrer le fonctionnement de l'algorithme de détection des composantes connexes à l'aide d'une matrice d'adjacence, prenons un exemple simple avec un graphe ayant 5 sommets et 4 arêtes.

- **Exemple de graphe par la matrice d'adjacence :**

Considérons le fichier d'entrée INPCONGRAPH.TXT suivant :

5 4

1 2

3 4

3 5

La première ligne indique qu'il y a 5 sommets et 4 arêtes.

Les lignes suivantes listent les arêtes entre les sommets :

- Arête entre le sommet 1 et le sommet 2.
- Arête entre le sommet 3 et le sommet 4.
- Arête entre le sommet 3 et le sommet 5.

Initialisation :

Le graphe est initialisé avec 5 sommets, et une matrice d'adjacence de taille (6 x 6) (indices de 0 à 5) est créée, initialisée avec des zéros :

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Ajout des Arêtes : Pour chaque arête, la matrice d'adjacence est mise à jour :

- Après l'ajout de l'arête (1, 2) :

0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

- Après l'ajout de l'arête (3, 4) :

0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0
0	0	0	0	0	0

- Après l'ajout de l'arête (3, 5) :

0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	0	0	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

Détection des Composantes Connexes :

- Le vecteur visited est initialisé à false pour tous les sommets.
- On itère sur chaque sommet pour détecter les composantes connexes :
  - Pour le sommet 1 :
    - Non visité, appelle dfs(1), marque le sommet 1 comme visité.
    - Explore le voisin 2, marque 2 comme visité, et ajoute à la composante [1, 2].
  - Pour le sommet 2 :
    - Déjà visité, on passe au suivant.
  - Pour le sommet 3 :
    - Non visité, appelle dfs(3), marque 3 comme visité.
    - Explore le voisin 4, marque 4 comme visité, ajoute à la composante [3, 4].
    - Explore le voisin 5, marque 5 comme visité, ajoute à la composante [3, 4, 5].
  - Pour le sommet 4 et 5 :
    - Déjà visités, fin de l'itération.

Résultat Final

2

Composante connexe 1

1 2

Composante connexe 2

3 4 5

Interprétation du Résultat

- Nombre de Composantes Connexes : 2
- Composante Connexe 1 : Comprend les sommets 1 et 2, qui sont connectés entre eux.
- Composante Connexe 2 : Comprend les sommets 3, 4, et 5, qui sont tous interconnectés

- Méthode de travail pour liste d'adjacence :

Le code présenté est une implémentation d'un algorithme pour trouver les composantes connexes d'un graphe non orienté à l'aide d'une liste d'adjacence. La classe **GraphList** est conçue pour gérer les opérations sur le graphe. Le constructeur de la classe initialise la liste d'adjacence en créant un vecteur de vecteurs, où chaque index représente un sommet du graphe et contient les sommets adjacents. La méthode **addEdge** ajoute une arête entre deux sommets, en mettant à jour les listes d'adjacence correspondantes pour chaque sommet, assurant ainsi que le graphe reste non orienté.

La détection des composantes connexes est réalisée via la méthode **findConnectedComponents**. Cette méthode commence par initialiser un vecteur visited pour garder une trace des sommets déjà visités. Elle itère sur tous les sommets, et pour chaque sommet non visité, elle appelle la méthode dfs (depth-first search) pour explorer tous les sommets connectés. La méthode **dfs** marque le sommet actuel comme visité et ajoute ce sommet à la composante actuelle. Elle explore ensuite tous les voisins de ce sommet, appelant récursivement dfs sur chacun d'eux si ce voisin n'a pas déjà été visité.

Une fois toutes les composantes connexes identifiées, la méthode **writeOutput** est utilisée pour écrire le résultat dans un fichier de sortie. Elle enregistre le nombre total de composantes trouvées, suivies de chaque composante avec les sommets qui la composent. Si le fichier de sortie ne peut pas être ouvert, un message d'erreur est affiché.

Enfin, une fonction externe **readGraphList** est fournie pour lire un fichier d'entrée qui spécifie le nombre de sommets et d'arêtes du graphe, ajoutant les arêtes au graphe en conséquence. Cette organisation du code permet une gestion efficace des graphes et de leurs propriétés connexes, en utilisant une structure de données intuitive pour la représentation des arêtes.

Voici un exemple détaillé d'exécution du code de la classe GraphList pour trouver les composantes connexes d'un graphe non orienté à l'aide d'une liste d'adjacence.

- **Exemple de graphe par la liste d'adjacence :**

Considérons le fichier d'entrée INPCONGRAPH.TXT suivant :

5 4

1 2

3 4

3 5

4 5

Ce fichier indique qu'il y a 5 sommets et 4 arêtes, les arêtes étant les suivantes :

- 1 - 2
- 3 - 4
- 3 - 5
- 4 – 5

Étapes d'exécution

1. **Lecture du fichier d'entrée :**
   - La fonction readGraphList est appelée avec le nom du fichier d'entrée.
   - Elle lit le nombre de sommets (n = 5) et le nombre d'arêtes (m = 4).
   - Un objet GraphList est créé avec n sommets.
2. **Ajout des arêtes :**
   - Pour chaque paire de sommets lue, la méthode addEdge est appelée, qui met à jour la liste d'adjacence.
   - Après l'ajout des arêtes, la structure de la liste d'adjacence est la suivante :
     - 1 : [2]
     - 2 : [1]
     - 3 : [4, 5]
     - 4 : [3, 5]

- 5 : [3, 4]

3. **Recherche des composantes connexes :**
   - La méthode findConnectedComponents est appelée. Elle initialise un vecteur visited pour garder une trace des sommets visités.
   - Elle itère sur chaque sommet. Pour le sommet 1, comme il n'est pas visité, elle appelle dfs (1).
4. **Exploration avec DFS :**
   - Dans dfs (1) :
     - 1 est marqué comme visité, et ajouté à la composante courante : component = [1].
     - On parcourt les voisins de 1. Le voisin 2 est trouvé et comme il n'est pas visité, on appelle dfs(2).
   - Dans dfs (2) :
     - 2 est marqué comme visité, et ajouté à la composante : component = [1, 2].
     - Les voisins de 2 sont examinés, mais il n'y a pas de nouveaux sommets à visiter, donc dfs (2) se termine et retourne à dfs (1).
5. **Complétion de la première composante :**
   - La première composante connexe, qui contient les sommets 1 et 2, est ajoutée à components.
6. **Continuation de la recherche :**
   - La méthode continue avec les sommets 3, 4, et 5.
   - Pour 3, elle appelle dfs (3) :
     - 3 est marqué comme visité, component = [3].
     - Les voisins 4 et 5 sont explorés, appelant successivement dfs (4) et dfs (5), ajoutant ainsi 3, 4, et 5 à la composante.
   - La seconde composante connexe est alors identifiée : components = [[1, 2], [3, 4, 5]].

7. **Écriture des résultats :**
    - Après avoir exploré tous les sommets, la méthode writeOutput est appelée.
    - Elle écrit le nombre de composantes (2) dans le fichier de sortie OUTCONGRAPH.TXT, suivi des composantes elles-mêmes :

      2

      Composante connexe 1

      1 2

      Composante connexe 2

      3 4 5

À la fin de l'exécution, le programme a trouvé deux composantes connexes dans le graphe :

- La première composante connexe est composée des sommets 1 et 2.
- La deuxième composante connexe est composée des sommets 3, 4 et 5.

Exercice 5 :

- Objectif :

Dans cet exercice, on vise à explorer et de comparer différentes méthodes pour résoudre le problème du plus court chemin dans un graphe non orienté et pondéré. On se concentre sur l'implémentation de l'algorithme de Dijkstra en utilisant trois représentations du graphe : matrice d'adjacence, liste d'adjacence et tas (heap). À travers ces différentes approches, l'objectif est de mieux comprendre les avantages et les inconvénients de chaque représentation et structure de données en termes de complexité temporelle et mémoire.

On vise à manipuler un graphe fourni via un fichier texte, à implémenter des algorithmes pour calculer la distance et le chemin les plus courts entre deux sommets, et à optimiser ces calculs en utilisant différentes structures de données.

- Méthode de travail pour matrice d'adjacence :

Le programme ci-dessus implémente un algorithme de calcul des plus courts chemins dans un **graphe non orienté et pondéré** en utilisant l'**algorithme de Dijkstra**. Voici un aperçu détaillé de la méthode de travail, organisée en plusieurs étapes.

1. **Lecture du graphe depuis un fichier** : La méthode *readGraph* lit les données du graphe à partir d'un fichier texte. Ce fichier contient le nombre de sommets, le nombre d'arêtes, le sommet de départ et celui d'arrivée. Ensuite, les arêtes du graphe sont lues, chaque arête étant définie par deux sommets connectés et le poids de l'arête. Ces informations sont stockées dans une **matrice d'adjacence** appelée *adjMatrix*, initialisée avec une valeur infinie (INF) pour indiquer l'absence de connexion entre les sommets au départ.
2. **Algorithme de Dijkstra** : La méthode *dijkstra* implémente l'algorithme de Dijkstra pour trouver le plus court chemin entre le sommet de départ et tous les autres sommets du graphe.
    - **Initialisation** : La distance du sommet de départ est définie comme 0 et toutes les autres distances sont initialisées à l'infini.
    - **Queue de priorité** : Une **file de priorité** (*min-heap*) est utilisée pour stocker les sommets à explorer, priorisés par leur distance actuelle.

- **Exploration des sommets** : À chaque étape, le sommet avec la plus petite distance (contenu dans la file de priorité) est exploré, et la distance vers ses voisins est mise à jour si un chemin plus court est trouvé. Le tableau *prev* permet de suivre le chemin emprunté pour revenir au sommet initial.
- Le résultat est une paire de vecteurs : le vecteur des distances les plus courtes pour chaque sommet (*dist*), et un vecteur contenant le prédécesseur de chaque sommet (*prev*), utilisé pour reconstruire le chemin.

3. **Écriture du résultat dans un fichier** : La méthode *writeOutput* écrit les résultats dans un fichier texte. Elle commence par écrire la distance la plus courte entre le sommet de départ et le sommet d'arrivée. Ensuite, elle reconstruit le chemin en suivant le vecteur des prédécesseurs *prev*, puis affiche ce chemin de manière ordonnée. Le chemin est affiché sous la forme de sommets séparés par une flèche "→".

- Méthode de travail pour Liste d'adjacence :

La première étape consiste à lire les données du graphe à partir d'un fichier d'entrée. Cette opération se déroule comme suit :

- **Ouverture du Fichier** : Le fichier est ouvert en lecture. Si l'ouverture échoue, un message d'erreur est affiché.
- **Extraction des Données** :
  - Le nombre de sommets (*numVertices*), le nombre d'arêtes (*numEdges*), ainsi que les sommets de départ (*startVertex*) et d'arrivée (*endVertex*) sont extraits de la première ligne du fichier.
  - La liste d'adjacence (*adjacencyList*) est ensuite initialisée avec une taille suffisante pour contenir tous les sommets.
- **Ajout des Arêtes** : Les arêtes du graphe sont lues dans le fichier, et pour chaque paire de sommets connectés par une arête, l'arête est ajoutée à la liste d'adjacence de chaque sommet (pour représenter le graphe non orienté). Chaque arête est associée à un poids, qui est également stocké.

Exécution de l'Algorithme de Dijkstra

L'algorithme de Dijkstra est ensuite exécuté pour calculer le chemin le plus court à partir du sommet de départ. Voici les principales étapes :

- **Initialisation des Structures de Données** :
  - Un tableau distances est créé pour stocker la distance minimale connue pour chaque sommet, initialisé à l'infini (INF), sauf pour le sommet de départ qui est initialisé à 0.
  - Un tableau *predecessors* est utilisé pour garder une trace du prédécesseur de chaque sommet dans le chemin le plus court, initialisé à -1.
  - Un tableau *visited* est créé pour suivre les sommets qui ont déjà été traités.
- **File de Priorité** :
  - Une file de priorité (*min-heap*) est utilisée pour sélectionner le sommet non visité avec la distance minimale. Le sommet de départ est ajouté à cette file.
- **Boucle Principale** :
  - Tant que la file de priorité n'est pas vide, le sommet *u* avec la distance minimale est extrait.
  - Si *u* a déjà été visité, il est ignoré. Sinon, il est marqué comme visité.
  - Pour chaque voisin *v* de *u*, si v n'a pas encore été visité et que la distance à *u* plus le poids de l'arête (*u, v*) est inférieur à la distance connue pour *v*, alors :
    - La distance pour *v* est mise à jour.
    - Le prédécesseur de *v* est défini sur *u*.
    - *v* est ajouté à la file de priorité.

Écriture des Résultats

Après l'exécution de l'algorithme, les résultats sont écrits dans un fichier de sortie :

- **Distance Minimale** : La distance la plus courte du sommet de départ au sommet d'arrivée est écrite dans le fichier.

- **Reconstruction du Chemin** :
    - Le chemin le plus court est reconstruit en remontant à partir du sommet d'arrivée à l'aide du tableau *predecessors*.
    - Les sommets du chemin sont ajoutés à un vecteur, qui est ensuite inversé pour obtenir l'ordre correct.
    - Le chemin est écrit dans le fichier, les sommets étant séparés par une flèche (→).

- Méthode de travail pour Tas :

La classe *GraphTas* représente le graphe à l'aide d'une matrice d'adjacence et d'une liste d'adjacence, tandis qu'un tas (priority queue) est utilisé pour optimiser la recherche du sommet avec la distance minimale.

La première étape dans la construction du graphe se fait via le constructeur de la classe *GraphTas* :

- **Initialisation** : Le constructeur prend en paramètre le nombre de sommets (*vertices*) et initialise la matrice d'adjacence (*adjMatrix*) avec des valeurs représentant des arêtes infinies (INF) pour indiquer qu'il n'y a pas encore d'arêtes entre les sommets. Une liste d'adjacence (*adjList*) est également initialisée pour stocker les voisins de chaque sommet.
- **Ajout des Arêtes** : La méthode *addEdge* permet d'ajouter une arête entre deux sommets avec un poids donné. Cette méthode met à jour à la fois la matrice d'adjacence et la liste d'adjacence pour refléter cette connexion. Pour un graphe non orienté, les arêtes sont ajoutées dans les deux directions.

Chargement du Graphe à partir d'un Fichier

La méthode *loadFromFile* est utilisée pour lire le graphe à partir d'un fichier d'entrée :

- **Ouverture et Lecture du Fichier** : Un fichier est ouvert pour lecture. Si l'ouverture échoue, un message d'erreur est affiché.
- **Extraction des Données** : Les valeurs des sommets, des arêtes, ainsi que des sommets de départ et d'arrivée sont extraites du fichier. Les indices des sommets sont ajustés pour

correspondre à une indexation de type 0 (c'est-à-dire que les sommets dans le fichier sont supposés être indexés à partir de 1).

- **Ajout d'Arêtes** : Pour chaque arête spécifiée dans le fichier, la méthode *addEdge* est appelée pour mettre à jour les structures de données du graphe.

Exécution de l'Algorithme de Dijkstra

La méthode *dijkstraWithHeap* implémente l'algorithme de Dijkstra :

- **Initialisation des Structures de Données** : Deux vecteurs, *dist* et *prev*, sont créés pour stocker respectivement les distances minimales à chaque sommet et les prédécesseurs pour reconstruire le chemin le plus court. La distance pour le sommet de départ est initialisée à 0, et toutes les autres distances sont initialisées à *INF*.
- **Utilisation d'un Tas** : Un tas de priorité (min-heap) est utilisé pour gérer les sommets à explorer. Le sommet de départ est ajouté au tas avec une distance de 0.
- **Boucle Principale de Dijkstra** :
    - Tant que le tas n'est pas vide, le sommet avec la distance minimale (*u*) est extrait.
    - Si le sommet extrait est le sommet de fin (end), l'algorithme se termine prématurément, car le chemin le plus court a été trouvé.
    - Pour chaque voisin de u, si la distance à travers u est inférieure à la distance connue pour le voisin (*v*), la distance est mise à jour, le prédécesseur est enregistré, et le voisin est ajouté au tas pour être exploré ultérieurement.

Sauvegarde des Résultats dans un Fichier

Enfin, la méthode *saveToFile* est utilisée pour écrire les résultats dans un fichier de sortie :

- **Ouverture du Fichier** : Un fichier est ouvert en écriture. En cas d'erreur, un message d'erreur est affiché.
- **Sauvegarde de la Distance Minimale** : La distance minimale entre le sommet de départ et le sommet de fin est écrite dans le fichier.

- **Reconnaissance et Sauvegarde du Chemin** : Le chemin le plus court est reconstruit à partir du vecteur des prédécesseurs (*prev*). Les sommets sont ajoutés à un vecteur, qui est ensuite inversé pour obtenir l'ordre correct, puis écrit dans le fichier avec des flèches (→) séparant les sommets.

➔ **Avantages de 5.2 (Liste d'Adjacence) par Rapport à 5.1 (Matrice d'Adjacence)**

1. **Utilisation de l'Espace Mémoire**
    - **Efficacité pour les Graphes Clairsemés** : Les listes d'adjacence consomment moins d'espace mémoire pour les graphes clairsemés. Dans une matrice d'adjacence, chaque paire de sommets nécessite un espace (même s'il n'y a pas d'arête entre eux), tandis qu'une liste d'adjacence ne stocke que les arêtes présentes, ce qui est plus efficace lorsque le nombre d'arêtes est inférieur au carré du nombre de sommets.
2. **Accès aux Voisins**
    - **Parcours Efficace des Voisins** : Avec une liste d'adjacence, il est possible de parcourir directement les voisins d'un sommet sans avoir à vérifier chaque sommet dans une ligne de la matrice, ce qui peut réduire considérablement le temps de calcul lors de l'exploration des arêtes.
3. **Dynamisme et Modifications**
    - **Ajout et Suppression d'Arêtes** : Les listes d'adjacence facilitent l'ajout et la suppression d'arêtes, ce qui est plus complexe avec une matrice d'adjacence. Cela rend la structure plus flexible pour les graphes dynamiques où la structure peut changer fréquemment.

➔ **Avantages de 5.3 (Tas) par Rapport à 5.1 (Matrice d'Adjacence)**

1. **Efficacité de l'Algorithme de Dijkstra**
    - **Optimisation des Performances** : L'utilisation d'un tas pour stocker les nœuds à explorer permet d'optimiser l'algorithme de Dijkstra. Dans une matrice d'adjacence, trouver le sommet avec la plus petite distance peut nécessiter un temps O(n) pour vérifier tous les sommets. Avec un tas, cette opération est

réduite à O(log n), ce qui est particulièrement bénéfique pour les graphes de grande taille.

2. **Réduction des Coûts de Temps d'Exécution**
    - **Complexité Temporelle Améliorée** : L'utilisation d'un tas permet de réduire la complexité globale de l'algorithme de Dijkstra à O((n + m) log n) (où m est le nombre d'arêtes), contrairement à une approche naïve avec une matrice d'adjacence, qui peut être moins efficace, en particulier pour les graphes clairsemés.
3. **Flexibilité pour les Graphes de Grande Taille**
    - **Scalabilité** : Les structures utilisant un tas sont plus adaptées pour traiter des graphes de grande taille, où la performance devient cruciale. La combinaison d'une liste d'adjacence et d'un tas permet une gestion plus efficace des distances minimales dans les scénarios complexes.

Conclusion

Les solutions utilisant une liste d'adjacence (5.2) et un tas (5.3) offrent des avantages significatifs en termes de mémoire, de performance et de flexibilité par rapport à l'utilisation d'une matrice d'adjacence (5.1). Ces améliorations sont particulièrement notables dans le contexte de graphes clairsemés ou de grande taille, rendant les approches 5.2 et 5.3 plus efficaces et adaptées à des applications pratiques dans le calcul des chemins les plus courts.

La Question 5.4

L'objectif de la question 5.4 est de déterminer le chemin le plus court entre un sommet de départ et tous les autres sommets d'un graphe non orienté, en utilisant une matrice d'adjacence pour sa représentation. Cette question vise à élargir notre compréhension de l'algorithme de Dijkstra, en se concentrant non seulement sur un chemin spécifique, mais sur la détermination des distances minimales à partir d'un sommet donné vers tous les sommets accessibles du graphe. Cela permet d'évaluer l'efficacité de l'algorithme dans un contexte plus général et d'observer comment les performances peuvent varier selon la structure du graphe.

➔ Méthode de Travail Utilisée dans le Code

Le code qu'on a proposé pour la question 5.4 a été conçu de manière systématique pour implémenter l'algorithme de Dijkstra, en utilisant une matrice d'adjacence comme structure de données principale pour représenter le graphe.

Voici un aperçu détaillé des étapes effectués :

1. **Initialisation de la Classe et des Attributs** :
    - La classe *GraphGeneral* est initialisée avec les paramètres *vertices*, *edges*, *start*, et *end*, permettant de définir le nombre de sommets, le nombre d'arêtes, et les sommets de départ et d'arrivée. Cela favorise une encapsulation claire des données relatives au graphe.
    - La matrice *graph* est initialisée avec une taille *vertices* x *vertices*, remplie de la valeur INF (représentant l'absence d'arêtes), pour chaque paire de sommets. Cette approche assure que toutes les distances sont initialement considérées comme infinies, sauf pour la distance de départ, qui est définie à zéro.
2. **Ajout d'Arêtes** :
    - La méthode *addEdge* permet d'ajouter des arêtes au graphe en définissant le poids correspondant dans la matrice d'adjacence. Le fait de mettre à jour la matrice pour les deux sommets *u* et *v* garantit que le graphe reste non orienté. Cette méthode offre une interface simple pour l'insertion d'arêtes, améliorant ainsi la lisibilité et la maintenance du code.

3. **Implémentation de l'Algorithme de Dijkstra** :
    - La méthode *dijkstra* commence par initialiser un tableau *visited* pour suivre les sommets déjà traités. Le tableau *dist* est rempli de INF, sauf pour le sommet de départ, dont la distance est fixée à zéro. Le tableau *prev* est initialisé à -1 pour chaque sommet, ce qui facilitera la reconstruction du chemin par la suite.
    - Une boucle principale s'exécute pour chaque sommet, où *minDistance* est appelée pour déterminer le sommet avec la distance minimale non visitée. Ce choix est crucial pour garantir que l'algorithme fonctionne de manière optimale.
    - Après avoir sélectionné le sommet *u*, le code met à jour les distances de ses voisins en vérifiant si un chemin plus court peut être établi. Si c'est le cas, la distance est mise à jour et le prédécesseur est enregistré. Cela constitue la relaxation des arêtes.
4. **Gestion des Résultats** :
    - Une fois l'algorithme terminé, la méthode *outputResults* est chargée d'écrire les résultats dans un fichier de sortie. Elle commence par vérifier si une distance vers le sommet d'arrivée existe. Si c'est le cas, elle enregistre la distance et construit le chemin en parcourant le tableau des prédécesseurs *prev*. Cela permet de reconstruire le chemin en remontant depuis le sommet d'arrivée jusqu'au sommet de départ.
    - Les résultats sont formatés pour être présentés de manière lisible, utilisant la notation 1-based pour correspondre aux attentes de l'utilisateur. Cela renforce l'expérience utilisateur en présentant les informations de manière claire et structurée.