Mini-projet 1

Lorsqu'on évoque la suite de Fibonacci, on entre dans le monde des mathématiques. La suite de Fibonacci est une séquence de nombres qui commence souvent par 0 et 1, et chaque nombre suivant est la somme des deux précédents. Cette séquence, qui semble simple à première vue, recèle en réalité une profonde complexité mathématique et se retrouve dans de nombreux aspects de la nature, de l'art et même de la technologie moderne.

Ces suites ont une importance particulière dans le domaine de la complexité algorithmique. Elles sont souvent utilisées comme des exemples de séquences qui illustrent les performances et les limites des algorithmes. Par exemple, la suite de Fibonacci est couramment utilisée pour analyser la complexité des algorithmes récursifs.

Dans cette partie, nous avons exploré trois approches différentes pour calculer les nombres de Fibonacci : la version récursive, la version itérative et la version basée sur l'exponentiation de matrice.

Version de base récursive :

La méthode récursive est la plus simple à comprendre, mais elle peut être inefficace pour de grandes valeurs de n en raison de la redondance des calculs. Cette version a été implémentée en utilisant une fonction récursive qui fait appel à elle-même pour calculer les nombres de Fibonacci. L'algorithme en pseudo-code suivant décrit l'approche implémentée :

Algorithm 1 Calcul des nombres de Fibonacci - Version Récursive

```
function FIBONACCI_RECURSIVE(n)
    if n ≤ 1 then
        return n
    else
        return FIBONACCI_RECURSIVE(n − 1) + FIBONACCI_RECURSIVE(n − 2)
    end if
end function
```

— **Analyse de la complexité :**

La méthode récursive est la plus simple à comprendre, mais elle peut être inefficace pour de grandes valeurs de n en raison de la redondance des calculs. Cette version a été implémentée en utilisant une fonction récursive qui fait appel à elle-même pour calculer les nombres de Fibonacci.

L'algorithme commence par le calcul de $F(n)$, qui nécessite le calcul de $F(n-1)$ et $F(n-2)$, puisque $F(n) = F(n-1) + F(n-2)$. Chaque appel récursif divise le problème en deux sous-problèmes plus petits, et cela se répète jusqu'à ce que nous atteignions les cas de base, $F(0)$ et $F(1)$, pour lesquels nous retournons simplement les valeurs. Ainsi, pour calculer $F(n)$, l'algorithme effectue une série d'appels récursifs pour les valeurs $F(n-1)$ et $F(n-2)$, et chaque niveau de la récursion divise le problème en deux.

La complexité en Θ de cette approche est de $\Theta(2^n)$ dans le pire des cas. Cela signifie que le nombre d'appels récursifs double à chaque niveau de la récursion, ce qui entraîne une croissance exponentielle du temps d'exécution en fonction de n. Cette complexité en $\Theta(2^n)$ est une borne inférieure et supérieure pour le temps d'exécution, ce qui signifie que l'algorithme peut être plus rapide pour certains cas particuliers, mais il ne sera jamais meilleur que $\Theta(2^n)$ dans le pire des cas.

Version de base itérative

La version itérative est plus efficace car elle évite la redondance des calculs. Elle utilise une boucle pour calculer progressivement les nombres de Fibonacci, en stockant les résultats intermédiaires dans une liste.

L'algorithme en pseudo-code suivant décrit l'implémentation de cette approche :

Algorithm 2 Calcul des nombres de Fibonacci - Version Itérative

```
function FIBONACCI_ITERATIVE(n)
    if n ≤ 0 then
        return 0
    else if n == 1 then
        return 1
    else
        fib ← [0, 1]
        for i from 2 to n do
            fib.append(fib[i − 1] + fib[i − 2])
        end for
        return fib[n]
    end if
end function
```

— **Analyse de la complexité :**

La version itérative utilise une boucle pour calculer les nombres de Fibonacci. Elle commence par initialiser une liste avec les deux premiers nombres de Fibonacci, $F(0)$ et $F(1)$. Ensuite, elle itère à travers la liste pour calculer progressivement les nombres de Fibonacci suivants en utilisant la relation $F(n) = F(n-1) + F(n-2)$.

La complexité en Θ de cette approche est de $\Theta(n)$. Chaque itération de la boucle calcule un nombre de Fibonacci supplémentaire, et il y a n itérations au total pour calculer $F(n)$. Étant donné que l'algorithme effectue une itération sur chaque valeur de 2 à n , le nombre total d'opérations est proportionnel à n . Par conséquent, sa complexité

est linéaire en $\Theta(n)$, ce qui signifie que le temps de calcul augmente linéairement avec la valeur de n .

Version basée sur l'exponentiation de matrice

La version basée sur l'exponentiation de matrice repose sur des propriétés mathématiques avancées. L'exponentiation de matrice est utilisée pour calculer les nombres de Fibonacci de manière efficace.

Lorsqu'il s'agit de calculer une puissance non nécessairement une puissance de 2, comme x^p, une approche efficace consiste à utiliser la méthode de l'exponentiation rapide. Cette méthode divise le problème en sous-problèmes plus petits, réduisant ainsi le nombre total de multiplications.

1. Si l'exposant p est égal à 0, le résultat est 1, car toute valeur élevée à la puissance 0 est égale à 1.
2. Si l'exposant p est pair, on divise p par 2 pour obtenir $p/2$, puis on calcule $x^{p/2}$ en utilisant la même méthode de manière récursive. Ensuite, on élève ce résultat au carré en le multipliant par lui-même.
3. Si l'exposant p est impair, on calcule x^{p-1} en utilisant la même méthode de manière récursive, ce qui donne un résultat pair. Ensuite, on multiplie ce résultat par x pour obtenir x^p.

L'algorithme en pseudo-code suivant décrit l'implémentation de cette approche :

Algorithm 3 Calcul des nombres de Fibonacci - Version basée sur l'exponentation de matrice

```
1 : function expoMatrix(n)
2 :     if n = 0 then
3 :         return 0
4 :     end if
5 :     if n = 1 then
6 :         return 1
7 :     end if
8 :     Initialize result as matrix [[0], [1]]
9 :     Initialize base as matrix [[0, 1], [1, 1]]
10 :    function expoMatrixRec(n, matrix)
11 :        if n = 1 then
12 :            return matrix
13 :        end if
14 :        if n is even then
15 :            half_power ← expoMatrixRec(n//2, matrix)
16 :            return matrixProduct(half_power, half_power)
17 :        else
18 :            previous_power ← expoMatrixRec(n − 1, matrix)
19 :            return matrixProduct(matrix, previous_power)
20 :        end if
21 :    end function
22 :    base ← expoMatrixRec(n, base)
23 :    result ← matrixProduct(base, result)
24 :    return result[0][0]
25 : end function
```

— **Analyse de la complexité :**

L'idée clé de l'algorithme d'exponentiation rapide pour les matrices est d'élever une matrice particulière à la puissance (n-1) pour obtenir le résultat.

La complexité en Θ de cette approche est de $\Theta(\log(n))$. Cette complexité logarithmique est due à la manière dont l'exponentiation de matrice fonctionne. L'algorithme divise le problème en utilisant des exposants binaires, ce qui réduit considérablement le nombre d'opérations nécessaires pour élever la matrice à une puissance donnée. Ainsi, à mesure que n augmente, le temps d'exécution augmente beaucoup moins rapidement que de manière linéaire ou exponentielle.

Cette approche est la plus efficace en termes de complexité algorithmique, car elle divise le problème de manière exponentielle grâce à l'exponentiation de matrice.

Comparaison des performances

Nous avons opté pour l'environnement *Python* afin d'effectuer nos tests. Pour comparer l'efficacité relative de ces trois approches, nous avons mesuré le temps d'exécution pour différentes valeurs de n, comme le montre la figure suivante :

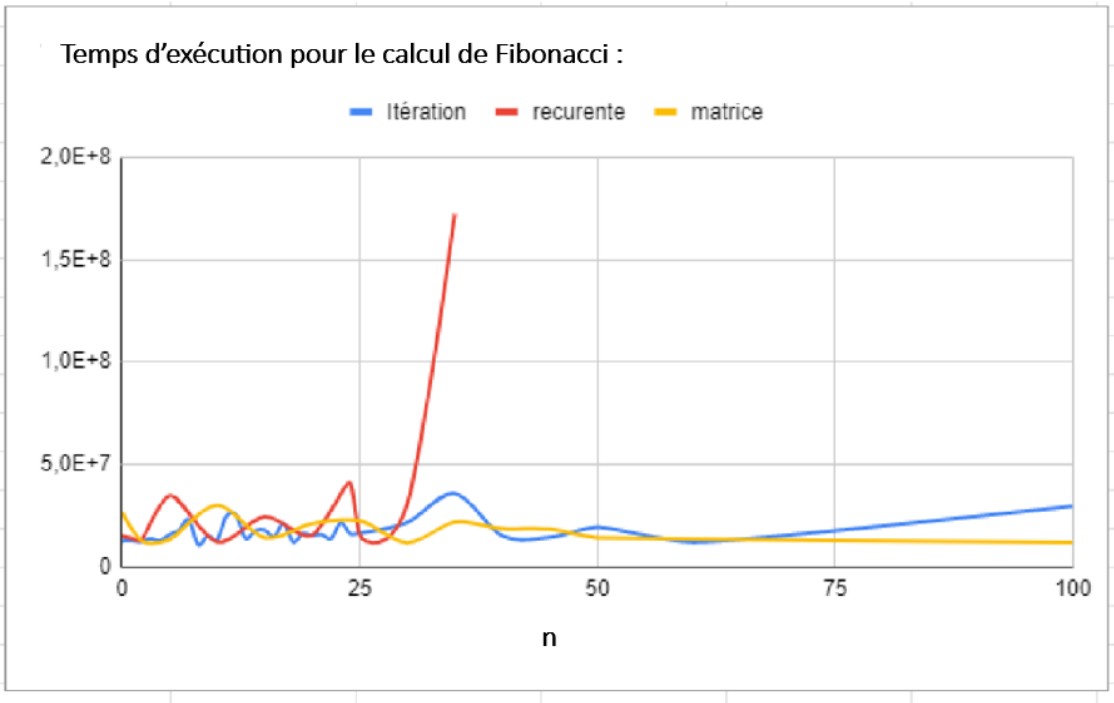

FIGURE 1 – Temps d'execution des trois approches.

Les figures suivantes indiquent le temps d'execution de chacune des méthodes :

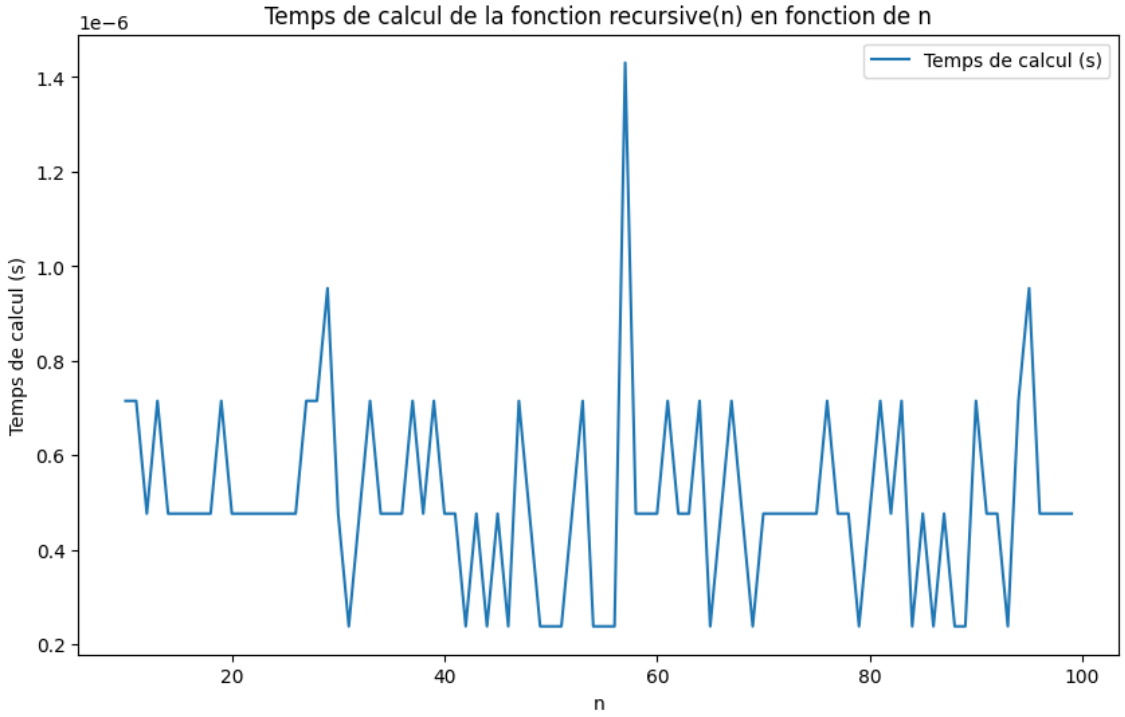

FIGURE 2 – Temps d'exécution de la version récursive.

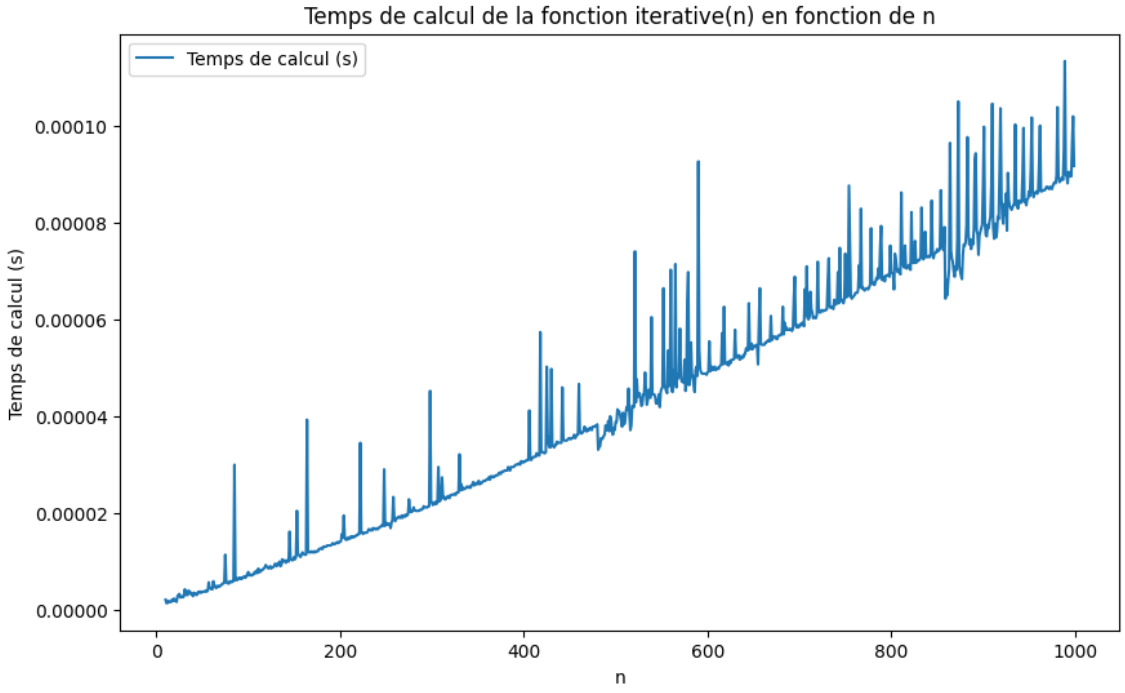

FIGURE 3 – Temps d'exécution de la version itérative.

Les résultats montrent clairement que la version récursive devient rapidement inefficace à mesure que n augmente en raison de la redondance des calculs. La version itérative est plus rapide et convient à un large éventail de valeurs de n. Cependant, la version basée sur l'exponentiation de matrice est la plus rapide de toutes, même pour des valeurs de n élevées, grâce à sa complexité logarithmique.

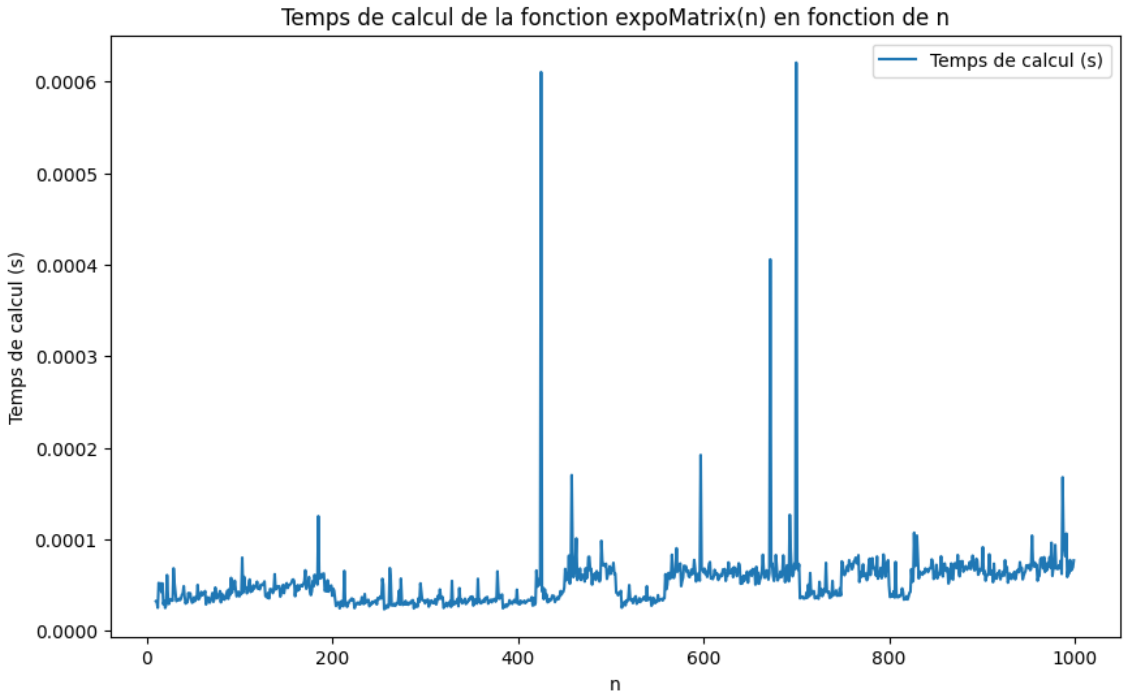

FIGURE 4 – Temps d'exécution de la version basée exponentiation de matrice.

Nous en concluons que les performances des trois approches pour le calcul des nombres de Fibonacci dépendent fortement de la valeur de n. La version récursive, bien qu'elle soit simple à comprendre, devient rapidement inefficace pour des valeurs élevées de n. La version itérative offre une performance linéaire acceptable pour des valeurs modérées de n, mais elle peut devenir lente pour de grandes valeurs. En revanche, la version basée sur l'exponentiation de matrice brille dans tous les cas, en offrant une complexité logarithmique qui la rend idéale pour des calculs de Fibonacci rapides et précis, même avec des valeurs très élevées de n. Elle est plus efficace que les autres approches, ce qui en fait le choix privilégié pour les calculs de Fibonacci.

Conclusion

Ce premier mini-projet nous a permis de comprendre les avantages et les inconvénients des différentes approches pour calculer les nombres de Fibonacci. Nous avons constaté que la version basée sur l'exponentiation de matrice est la plus efficace en termes de complexité algorithmique, ce qui en fait un choix judicieux pour les calculs de Fibonacci, en particulier lorsque des valeurs de n importantes sont impliquées.

Mini-projet 2

Lors de la mise en œuvre de ces solutions, nous avons choisi d'utiliser la représentation par **liste d'adjacence** pour les graphes non-orientés. Cette décision est basée sur plusieurs avantages de cette méthode :

1. **Efficacité en mémoire :** La représentation par liste d'adjacence est généralement plus efficace en termes de mémoire que la représentation matricielle, car elle ne stocke que les arêtes réellement présentes dans le graphe. Cela est particulièrement bénéfique pour les graphes de grande taille.
2. **Efficacité en temps :** La représentation par liste d'adjacence est souvent plus efficace pour effectuer des opérations de parcours et de recherche, ce qui est essentiel pour résoudre les problèmes liés aux **zones denses**.

1 Question 1 : Test de Vérification

Nous avons conçu un algorithme qui vérifie si un sous-ensemble donné de sommets forme une zone dense dans un graphe non-orienté.

Algorithm 4 Test de Vérification

```
1 : procedure EST_ZONE_DENSE(G, X)
2 :     for u in X do
3 :         for v in X do
4 :             if u ≠ v and v ∉ G[u] then
5 :                 return False
6 :             end if
7 :         end for
8 :     end for
9 :     return True
10 : end procedure
```

1 Explication et Complexité

Cet algorithme vérifie si chaque paire de sommets dans le sous-ensemble X est connectée dans le graphe G.

Sa complexité est $O(|X|^2)$, où $|X|$ est la taille de X.

Pour calculer sa complexité, examinons les étapes de l'algorithme.

- L'algorithme parcourt tous les sommets u dans X.
- Pour chaque sommet u, il parcourt à nouveau tous les sommets v dans X distincts de u.
- Dans la boucle interne, il vérifie si v n'est pas connecté à u dans le graphe G, ce qui implique une opération de recherche dans la liste d'adjacence de u

.

La complexité de chaque étape est la suivante :

1. Parcours des sommets de X : $O(|X|)$.
2. Pour chaque sommet u de X, parcours des sommets distincts v de X : $O(|X|)$ dans le pire cas.
3. À l'intérieur de la boucle interne, la recherche de l'existence de l'arête entre u et v a une complexité constante dans une liste d'adjacence : $O(1)$.

Pour obtenir la complexité totale de l'algorithme, multiplions ces étapes ensemble :

$$O(|X|) \cdot O(|X|) \cdot O(1) = O(|X|^2)$$

Ainsi, ça complexité $O(|X|^2)$ dans le pire cas. Cela signifie que le temps d'exécution de cet algorithme augmente quadratiquement avec la taille de l'ensemble de sommets X.

2 Question 2 : Calcul de Zone Dense Maximale

Cet algorithme calcule un sous-ensemble X de sommets qui forme une zone dense maximale dans un graphe non-orienté.

Algorithm 5 Calcul de Zone Dense Maximale

```
1 : procedure QUESTION2(G)
2 :     X_max ← {}                          ▷ Initialize maximum dense zone as empty set
3 :     for u in G do                       ▷ Iterate through all nodes
4 :         X_current ← {u}                 ▷ Initialize current zone with single node
5 :         R ← G.keys() − {u}              ▷ Remaining nodes
6 :         while R ≠ {} do                 ▷ While there are remaining nodes
7 :             v_best ← None
8 :             for v in X_current do
9 :                 for w in R do
10 :                    X_new ← X_current ∪ {w}
11 :                    if est_zone_dense(G, X_new) then
12 :                        v_best ← w
13 :                        break
14 :                    end if
15 :                end for
16 :                if v_best then
17 :                    X_current ← X_current ∪ {v_best}
18 :                    R ← R − {v_best}
19 :                else
20 :                    break
21 :                end if
22 :            end for
23 :        end while
24 :        if |X_current| > |X_max| then
25 :            X_max ← X_current
26 :        end if
27 :    end for
28 :    return X_max
29 : end procedure
```

Explication

L'algorithme utilisé pour résoudre cette question suit les étapes suivantes :

- intialiser une zone dense maximale vide (*max_dense_zone*).
- Ensuite, parcours de tous les nœuds du graphe (*graph*) en tant que nœud de départ potentiel pour une zone dense. Pour chaque nœud de départ, on initialise une zone courante (*current_zone*) contenant uniquement ce nœud, et nous créons un ensemble des nœuds restants (*remaining_nodes*) en incluant tous les nœuds du graphe G. Le nœud courant est retiré de la liste des nœuds restants.
- Tant qu'il reste des nœuds à explorer dans la liste des nœuds restants, l'algorithme tente de trouver le meilleur nœud voisin à ajouter à la zone courante. Pour cela, il parcourt les nœuds de la zone courante et les nœuds restants, et crée une nouvelle zone en ajoutant un nœud voisin. Si la nouvelle zone est dense, le nœud voisin est ajouté à la zone courante.
- Ce processus se répète jusqu'à ce qu'il n'y ait plus de nœuds à ajouter ou que la densité de la zone ne puisse plus être améliorée.
- Si la taille de la zone courante est supérieure à la taille de la zone dense maximale (*max_dense_zone*) connue jusqu'à présent, nous mettons à jour la zone dense maximale pour refléter la nouvelle zone courante.

Complexité de l'Algorithme

Pour évaluer la complexité de cet algorithme, examinons les étapes précédentes :

1. Parcours de tous les noeuds du graphe G pour déterminer un noeud de départ potentiel pour une zone dense : $O(|V|)$ où $|V|$ est le nombre de noeuds dans le graphe.
2. Dans le pire cas, chaque nœud est ajouté une fois à la zone courante, ce qui signifie qu'elle peut s'exécuter jusqu'à $O(|V|)$ fois.
3. 2 boucles imbriquées : la première boucle parcourt tous les noeuds de la zone courante, la seconde parcourt les noeuds restants. Dans le pire cas, chaque nœud est ajouté une fois à la zone courante, ce qui signifie qu'elle peut s'exécuter jusqu'à $O(|V|)$ fois. Dans le pire cas, nous devons comparer chaque nœud dans X avec chaque nœud dans R, ce qui donne une complexité de $O(|V||V|)$ pour cette partie.
4. Après avoir exploré toutes les options pour la zone courante, nous comparons sa taille à la taille de la zone dense maximale connue jusqu'à présent. Cette opération prend un temps constant, soit $O(1)$.

Pour obtenir la complexité totale de l'algorithme, multiplions ces étapes ensemble :

$$O(|V|) \cdot O(|V|) \cdot O(|V|) \cdot O(1) = O(|V|^3)$$

Donc, dans le pire cas, la complexité de cet algorithme est cubique par rapport au nombre de nœuds dans le graphe $|V|$.

Cet algorithme soit efficace pour les petits graphes, il est inefficace pour les graphes de grande taille en raison de sa complexité cubique.

3 Question 3 : Calcul de Zone Dense Maximum (Méthode Complète)

Cet algorithme prend en entrée un graphe non-orienté $G = (S, A)$ et qui calcule un sous-ensemble X de S qui est une zone dense maximum de G. Cela signifie qu'il n'existe pas de zone dense Y de G telle que $|Y| > |X|$.

Détails de l'algorithme

- Initialiser `max_dense_zone` comme un ensemble vide.
- Obtenir la liste de tous les sommets du graphe et la stocker dans `all_nodes`.
- Pour chaque taille de sous-ensemble possible (en commençant par le plus grand et en diminuant jusqu'à 1), faire les étapes suivantes :
  1. Générer tous les sous-ensembles de `all_nodes` de cette taille à l'aide de `combinations(all_node subset_size)`.
  2. Pour chaque sous-ensemble, convertir le sous-ensemble en un ensemble et vérifier si c'est une zone dense en utilisant `est_zone_dense(graph, subset_as_set)`.
  3. Si le sous-ensemble est une zone dense, retourner immédiatement ce sous-ensemble.
- Si aucun sous-ensemble n'est une zone dense, retourner l'ensemble vide.

Pseudo-code

Voici le pseudo-code correspondant à l'algorithme :

---

procedure QUESTION3(G)
  $X_{\max} \leftarrow \{\}$ ▷ Initialiser la zone dense maximale comme un ensemble vide
  $N \leftarrow$ nombre de sommets de G ▷ Nombre de sommets dans le graphe
  for $k \leftarrow N$ to 1 **do** ▷ Pour chaque taille possible de zone
    for S in toutes les combinaisons de sommets de taille k **do** ▷ Générer toutes les combinaisons de sommets de taille k
      if est_zone_dense(G, S) **then** ▷ Si le sous-ensemble est une zone dense
        return S ▷ Retourner immédiatement ce sous-ensemble
      end if
    end for
  end for
  return $X_{\max}$ ▷ Retourner la zone dense maximale trouvée
end procedure

Complexité de l'algorithme

La complexité temporelle de cet algorithme est principalement déterminée par deux facteurs : le nombre total de sous-ensembles possibles et le coût de vérifier si un sous-ensemble est une zone dense.

Le nombre total de sous-ensembles d'un ensemble de taille n est 2^n. Par conséquent, la boucle qui génère tous les sous-ensembles possibles a une complexité temporelle de $O(2^n)$.

La fonction `est_zone_dense` vérifie si tous les sommets d'un sous-ensemble sont connectés deux à deux par des arêtes. Dans le pire des cas, cela nécessite de vérifier toutes les paires possibles de sommets, ce qui a une complexité temporelle de $O(n^2)$.

Par conséquent, la complexité temporelle totale de l'algorithme est $O(2^n * n^2)$.

4 Question 4 : Calcul de zone dense maximum (méthode "incomplète")

Description de l'algorithme

L'algorithme commence par trier les sommets par degré décroissant. Il initialise ensuite la zone dense maximale avec le sommet de plus haut degré.

Il parcourt ensuite les sommets restants dans l'ordre décroissant du degré. Pour chaque sommet, il vérifie si l'ajout du sommet à la zone dense maximale donne toujours une zone dense. Si c'est le cas, il ajoute le sommet à la zone dense maximale.

Détails de l'algorithme

- Trie les sommets par degré décroissant et les stocke dans `nodes_by_degree`.
- Initialise la zone dense maximale avec le sommet de plus haut degré.
- Parcourt les sommets restants dans l'ordre décroissant du degré :
  1. Pour chaque sommet, vérifie si l'ajout du sommet à la zone dense maximale donne toujours une zone dense.
  2. Si c'est le cas, ajoute le sommet à la zone dense maximale.

Pseudo-code

Voici le pseudo-code correspondant à l'algorithme :

```
procedure QUESTION4(G)
    nodes_by_degree ← trie_les_sommets_par_degré_décroissant(G)
    max_dense_zone ← ensemble_avec_le_sommet_de_plus_haut_degré(nodes_by_degree)
    for chaque node dans nodes_by_degree[1 :] do
        if est_zone_dense(G, max_dense_zone|ensemble_avec_node(node)) then
            ajoute_node_à_max_dense_zone(node)
        end if
    end for
    return max_dense_zone
end procedure
```

Complexité de l'algorithme

La complexité temporelle de cet algorithme est principalement déterminée par deux facteurs : le coût du tri des sommets par degré et le coût de vérifier si un sous-ensemble est une zone dense.

Le coût du tri des sommets par degré est $O(n \log n)$, où n est le nombre de sommets.

La fonction `est_zone_dense` vérifie si tous les sommets d'un sous-ensemble sont connectés deux à deux par des arêtes. Dans le pire des cas, cela nécessite de vérifier toutes les paires possibles de sommets, ce qui a une complexité temporelle de $O(n^2)$.

Par conséquent, la complexité temporelle totale de l'algorithme est $O(n \log n + n^2)$.

5 Comparaison :

- **Question 2 : Calcul de zone dense maximale**
  - Approche : Cet algorithme utilise une approche gloutonne pour trouver une zone dense maximale dans le graphe.
  - Complexité temporelle : $O(n^3)$, où n est le nombre de sommets dans le graphe.
- **Question 3 : Calcul de zone dense maximum (méthode complète)**
  - Approche : Cet algorithme utilise une approche de force brute pour trouver la zone dense maximum. Il génère tous les sous-ensembles possibles de sommets et vérifie s'ils sont une zone dense.
  - Complexité temporelle : $O(2^n * n^2)$, où n est le nombre de sommets dans le graphe.
- **Question 4 : Calcul de zone dense maximum (méthode incomplète)**
  - Approche : Cet algorithme utilise une approche heuristique pour trouver une grande zone dense. Il trie les sommets par degré décroissant et tente d'ajouter des sommets à la zone dense maximale jusqu'à ce qu'il ne puisse plus ajouter de sommets sans violer la propriété de la zone dense.
  - Complexité temporelle : $O(n \log n + n^2)$, où n est le nombre de sommets dans le graphe.

En conclusion, l'algorithme pour la question 2 est le plus simple mais peut ne pas trouver

la zone dense maximum. L'algorithme pour la question 3 trouve la zone dense maximum mais a une complexité temporelle élevée. L'algorithme pour la question 4 est un compromis entre les deux : il trouve généralement une grande zone dense et a une complexité temporelle plus faible que l'algorithme pour la question 3.

Mini-projet 3

La Machine de Turing déterministe, ou MDT, est un concept clé en informatique théorique. Cette machine est composée d'un ruban, d'une tête de lecture/écriture et d'une matrice de transition. Les symboles sur la bande peuvent être lus, écrits et effacés par la tête, qui se déplace de gauche à droite. La MDT obéit à une règle simple : en fonction de son état actuel et du symbole sous sa tête, elle effectue une action déterminée.

— **Explication sur le codage choisi :**

Une matrice est définie pour représenter les transitions de la machine de Turing. Chaque ligne de cette matrice spécifie une transition sous la forme suivante : *{état actuel, symbole à vérifier sur le ruban, prochain état, symbole à écrire sur le ruban, direction de déplacement (Right, Left, S qui signifie le fait de rester sur la même position)}*. Cette matrice décrit comment la machine de Turing évolue d'un état à un autre en fonction du symbole sur le ruban. Les états de terminaison sur la machine sont représentés par **Y** ou **N** $Yltatfinalpouraccepterlemot, Npourrefuserlemot$.

Nous avons choisi d'implémenter notre code sous ***Java*** ; nous remplissons manuellement la matrice pour faire les tests, cependant, nous pouvons donner la main à l'utilisateur pour la remplir.

```
Character matrice[][] = new Character[][]{
    {'0', 'a', '1', 'a', 'R'},
    {'0', 'b', 'N', 'b', 'S'},
    {'1', 'a', 'N', 'a', 'S'},
    {'1', 'b', '2', 'b', 'R'},
    {'2', 'a', 'N', 'a', 'S'},
    {'2', 'b', 'N', 'b', 'S'},
    {'2', '#', 'Y', '#', 'S'}
};
```

Prenons cet exemple, nous définissons l'ensemble des transitions pour une MT qui reconnaît le mot **'ab'**.

- La fonction `Solution` est appelée avec les paramètres suivants :
  - `matrice` : la matrice de transitions de la MT.
  - "ab" : le mot à vérifier sur le ruban.
  - 0 : la position de départ sur le ruban (position initiale).

- La fonction commence par initialiser quelques variables, notamment `Pos` (position actuelle sur le ruban), `q` (état actuel de la MT), `j` (indice de la ligne dans la matrice), et `T` (variable de terminaison).
- Un `ArrayList` appelé `ruban` est utilisé pour représenter le ruban. Le mot "ab" est converti en un tableau de caractères, puis chaque caractère est ajouté à l'`ArrayList ruban`.
- Ensuite, une boucle `while` est utilisée pour exécuter la machine de Turing tant que `T` est égal à 0 (c'est-à-dire tant que la machine n'a pas terminé).
- À chaque itération de la boucle, la machine vérifie si elle peut effectuer une transition à partir de l'état actuel `q` et du symbole actuel sur le ruban `ruban.get(Pos)` en regardant la matrice de transitions `matrice`.

```
if (Character.getNumericValue(matrice[j][0]) == q)
```

- Si une transition est trouvée, la machine met à jour l'état, le symbole sur le ruban, et la position du ruban en fonction de la transition.
- La direction de déplacement (à gauche ou à droite) est indiquée par la cinquième colonne de la matrice de transitions.
- La boucle continue jusqu'à ce que la machine atteigne un état de terminaison (marqué par un 'Y' ou 'N' dans la matrice de transitions) ou qu'elle ne trouve plus de transition possible.
- Une fois la boucle terminée, le programme imprime "mot reconnu" si la machine a terminé dans un état de terminaison Y. Sinon, il imprime "mot refusé".
- Enfin, le contenu du ruban est affiché pour montrer le résultat.