Université d'Aix-Marseille - Master Informatique 1ère année UE Complexité - TP1 - 3 Mini-projets

Mini-projet 1. Calcul des nombres de Fibonacci : le point de vue expérimental

1. fibonacci par itération

Complexité linéaire θ(n) qui ne dépend que de n composé de plusieurs affectation et une addition.

```
fibonacci_ite(entier n){                    // θ(n)
        si (n<2) renvoyer n                 // O(3)
        initialiser fn_1 = 0 , fn = 1       // O(2)
        Pour ( i à n ){                     // O(3n)
                initialiser temp = fn
                fn = fn + fn_1
                fn_1 = temp
        }
}
```

2. fibonacci récursive

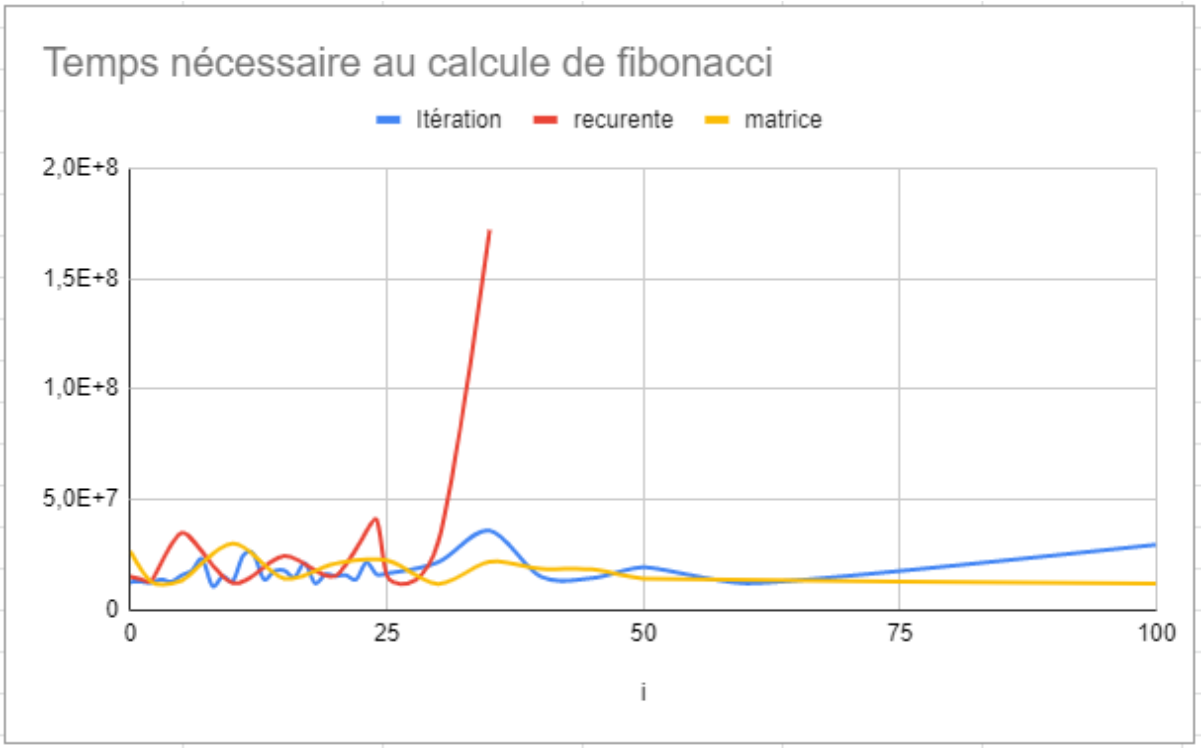
Complexité exponentiel θ(2^n), à chaque exécutions nous exécutons 2 fois la méthode simulant une structure d'arbre exponentielle.

```
fibonacci_recu(entier n){                                   // θ(2^n)
        si (n<2) renvoyer n                                 // O(3)
        renvoyer fibonacci_recu(n-1) + fibonacci_recu(n-2)  // O(2^n)
}
```

3. fibonacci par matrice

Afin de calculer la suite de fibonacci par matrice, on dispose de trois fonctions.
mult_matrix() qui permet de multiplier deux matrice de taille 2*2, sa complexité est θ(1) car on fait que 8 opération et ils sont fixe quelque soit l'entrée.
power_matrix() qui calcule la puissance n de la matrice donnée en paramètre. Cette dernière est récursive, à chaque itération elle appelle **power_matrix(m, n/2)** puis **mult_matrix(m, m)**, et si n est impaire elle appelle **mult_matrix(m, f)** f est la matrice de base. sa complexité est θ(log2(n))
fibonacci_matrix() qui fait juste appel à **power_matrix()**.

Donc la complexité de l'algorithme est θ(log2(n))



Mini-projet 2. Manipulation de graphes non-orientés

1. Test de vérification de zone vide

Le problème posé est de vérifier si un sous-ensemble X de sommets d'un graph G est une zone vide.

La solution dans la fonction **arrayIsVoid()** de la classe **Graph** consiste à parcourire les sommets de l'ensemble **X** donné en paramètre, dans la boucle on parcourt encore une fois **X** et pour chaque paire en vérifie dans la matrice **graph** s'il existe une arête entre les deux sommets, si oui on retourne faux, et si on atteint la fin des deux boucle sans trouver d'arête on retourne vrai.

La complexité de l'algorithme est θ(n^2)

2. Calcul de zone vide maximale

Le but ici est de calculer la zone vide maximale X du graphe G tels que X ne soit contenu dans aucune autre zone vide.

Pour répondre à cela, la fonction **getMaximalVoid()** de la classe **Graph** initialise une liste **result** qui contiendra les élément de la zone vide, puis parcourt tous les sommets du graphe donné en paramètre, et vérifie si on peut les rajouter à la liste **result** avec la fonction

checkVoid() et les ajoute si oui.la fonction **checkVoid()** prend en paramètre la matrice d'un graphe, une liste de sommet qui est une zone vide, et un sommet à vérifier, puis parcourt la liste et vérifie si il existe une arête entre chaque sommet de la liste et le sommet à vérifier et retourne faux si oui, et vrai sinon.

La fonction **checkVoid()** est θ(n) et dans **getMaximalVoid()** on fait appel à **checkVoid()** n fois. Donc la complexité de l'algorithme est θ(n^2).

3. Calcul de zone vide maximum

Dans cette partie on veut calculer la zone vide X maximum d'un graphe G tels qu'il n'existe aucune autre zone vide dans G avec une taille supérieure à celle de X.

Afin de répondre à cette question on dispose de deux fonctions **getAllMaximalVoids()** et **getMaxVoid()**, ainsi qu'une variable globale **size** qui représente le nombre de sommet dans le graphe, une liste **voidMax** pour on stocke la zone vide maximum, une liste **voidTemp** pour la zone vide maximale de chaque itération et enfin un vecteur de boolean **testedElements**.
getAllMaximalVoids() est une fonction récursive qui permet de parcourir toutes les permutation possible de l'ensemble des sommets du graphe, elle prend en paramètre un entier **level** qui permet de savoir dans quel case de la permutation on est, pour chaque itération, on met **testedElements[level]** à vrai pour ne pas le prendre en compte dans les prochaines cases. Puis on vérifie si on peut ajouter le sommet qu'on calcule à **voidTemp** et appelle **getAllMaximalVoids(level + 1)**, à la fin de l'itération on retire le sommet de **voidTemp** et on remet **testedElements[level]** à faux pour les prochaines permutation.
La condition d'arrêt de la fonction est quand **level** = **size**, ce qui veut dire qu'on atteint la fin de la permutation. On met à jour **voidMax** si la taille de **voidTemp** est supérieure à sa taille.
Dans la fonction **getMaxVoid()** on initialise juste la valeur de **testedElements** et donne la taille du graphe à la variable **size**, puis on lance **getAllMaximalVoids(0)**.

Donc la complexité de **getMaxVoid()** = la complexité de **getAllMaximalVoids()** + c.
Pour la complexité de **getAllMaximalVoids()** : T[n] = n * T[n-1]
Sachant que T[0] est la complexité de la condition d'arrêt donc dans le pire des cas quand on clone **voidTemp** dans **voidMax** qui est on θ(n).
Donc T[n] = n * n!
La complexité de l'algorithme est θ(n * n!).

4. Calcul de zone maximum méthode incomplète

Il est demandé de faire un algorithme efficace de calcul de zone maximum sans avoir la certitude d'avoir la solution optimale.

On part de l'idée que les sommets qui ont le plus de voisins bloquent plus l'agrandissement de la zone vide, donc si on essayait de les ajouter en dernier, on aura un résultat proche ou égal de l'optimal plus rapidement.
Dans la fonction **getIncompleteMaxVoid()** on calcul d'abord le nombre de voisin de chaque sommet, puis on trie les sommets dans un ordre décroissant, on crée une liste **result** pour stocker la zone vide, on parcourt la liste triée, on ajoute l'élément si **checkVoid()** retourne vrai. Et enfin on retourne le résultat.

La complexité de l'algorithme est la complexité de calcul de nombre de voisins θ(n^2)., plus la complexité du tri θ(n^2), plus la complexité de calcul de la zone vide dans le pire des cas quand le graph en entièrement vide θ((n*(n+1))/2) = θ(n ^2).
Donc la complexité de **getIncompleteMaxVoid()** est θ(3 * n^2) = θ(n ^2)

Mini-projet 3. Simulation d'une Machine de Turing Déterministe:

La complexité de ce programme:

1) Le fonctionnement de la simulation:

- Le simulateur contient le entrés suivant:

- Les etats : etate1, etate2….
- les symboles : symbole1, symbole2 …
- le symbole blanc.
- les états finaux : finalEtat1,finalEtat2…
- …

Rapidement :
- La premiere ligne lis les **etats** de la machine, la seconde ligne c'est les **symbole** de l'alphabet qui inclut le **symbole blanc**.
- Il y a aussi les **états finals** et l'**état initia**l de notre machine de turing.

2) Détails de la création de la machine de turing:

Pour créer le ruban, la classe **Case** est utilisée, c'est une liste **doublement chaînée** qui peut ajouter des valeurs vers la gauche et vers la droite ou chaque État connaît son prédécesseur et son successeur.

3) La simulation de la machine de turing :

- lecture des états.

  Créer les objets et les ajouter à la liste de la machine de turing.

- Lecture des Symboles.
- Lecture du symbole Blanc.
- Lecture de l'état initial.
- Lecture des états finaux.

- Lecture des transitions.

- La complexité de la création est de **O(3n)**. vue la lecture des transaction qui a les plus des calcul (**nombre des etats + nombre de symboles + nombre de transitions**).

- Et pour la **complexité** de la lecture du mot est **dépend** de la machine de turing. On ne peut pas déterminer une même complexité pour chaque machine de turing parce que **ça change** à chaque **changement de la machine**.

4) Exception:

Notre programme est **déterministe**. Si on passe une machine de turing qui n'est pas déterministe une **exception** sera levée et une erreur sera retournée lors de la création de la machine de turing.