Complexité - Compte-rendu du TP1

AMODEO Julien, CHARPY Raphaël, PAREL Gabriel, SAADI Akim

15 septembre 2021

1 Mini-Projet 1 (Calcul de la suite de Fibonacci)

Il est proposé ici d'implémenter trois versions du calcul de la suite de Fibonacci (récursive, itérative, et méthode basée sur l'exponentiation de matrice), afin d'en comparer l'efficacité en terme de complexité par rapport à la taille de l'entrée, et temps de calcul. Nous avons pour cela choisi d'utiliser le langage C.

On commence par présenter la méthode récursive, qui consiste à réaliser un appel récursif de la fonction $fibonacci - recursif()$ à chaque étape :

```
#En entree : un entier n
#En sortie : le nombre final de fibonacci

fibonacci_recursif(n) :
si n < 2 alors
          retourne n
retourne fibonacci_recursif(n-1) + fibonacci_recursif(n-2)
```

Ce programme appelle $fibonacci - recursif()$ deux fois à chaque étape, ce qui lui octroit une complexité en $O(2^n)$ (pas exactement, car on utilise $n-1$ et $n-2$ et par conséquent on ne multiplie pas vraiment par 2 la complexité à chaque étape, mais 2^n nous semble être une bonne approximation), c'est à dire exponentielle par rapport à la taille de l'entrée. Le calcul des termes de la suite à partir du rang 50 prend un temps considérable sur nos machines, mais reste rapide (entre 0.5 et 3s. environ) pour les termes précédents.

Nous présentons ensuite la seconde méthode, qui calcule les termes de la suite de Fibonacci de façon itérative à l'aide de l'initialisation d'un compteur :

```
#En entree : un long entier n
#En sortie : le nombre final de fibonacci

 fibo_iteratives(n) :
 i = 1
 j = 1
 cpt = 0
 k = 1
 Tant que cpt < n #les instructions de ce bloc sont realisees n fois
          si cpt < 2 alors
                    k = cpt
          sinon
        i = j
        j = k
        k = i + j
          cpt = cpt +1
retourne k
```

Ce programme-ci est bien plus efficace en terme de temps de calcul, car sa complexité est en $O(n)$ dans le pire des cas, soit linéaire par rapport à la taille de l'entrée. Le nombre utilisé dans le *main* est la limite pour laquelle le temps de calcul dépasse un temps raisonnable sur nos machines. On constate facilement que cette méthode nous permet de calculer la suite de Fibonacci plus efficacement, car il nous est possible de retourner f_{10^9}, tandis que la première méthode peine à retourner f_{50}.

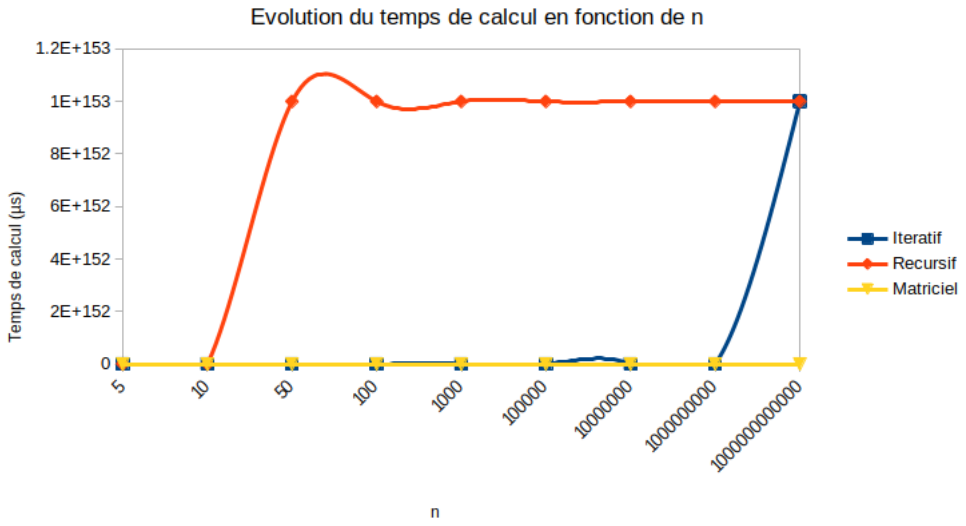
On termine par présenter la méthode basée sur l'exponentiation de matrices. Cette méthode, un peu plus astucieuse, utilise une expression matricielle de la formule de récurrence de la suite de Fibonacci qui nous permet de calculer n'importe quel terme de la suite à partir des termes initiaux f_0 et f_1 et d'une matrice mise à l'exposant recherché. Cela permet de s'affranchir de l'utilisation des termes précédents, et de ne réaliser que le calcul d'exponentation d'une matrice en dimension 2 :

```
#En entree : un tableau d'entier de dimension 4, A et un entier power
#En sortie : le resultat de l'exponention de la matrice A a la puissance power

powerMat(power,  A[4]):
entier temp[4]
```

```
entier result[4]
entier mat
entier powMat
entier i

pour i allant de 0    4
          temp[i] = A[i]
si power == 1 alors
          retourne temp
si power est pair alors
          tant que power != 1 #ce bloc est realise log_2(n) fois: l'exposant est divise par 2 a
              chaque etape
                    mat = matrix(temp,temp)
                    pour i allant de 0    4
                              result[i] = mat[i]
                              temp[i] = result[i]
                    power = power / 2
          retourne result
sinon
          powMat = powerMat(power-1, temp);
          mat = matrix(powMat, A);
          pour i allant de 0    4
                    result[i] = mat[i]
retourne result


#En entree : deux entier power et f0
#En sortie : le nombre de fibonacci

fibonacci(power,  f0):
matrix[4] = {0,1  1,1}
entier temp
entier result

temp = powerMat(power, matrix)
result = temp[0]*f0 + temp[1]
retourne result
```

Ce programme est le plus efficace des trois, car il permet de calculer f_{10^16} quasi instantanément sur nos machines. En effet, sa complexité est en $O(log_2(n)))$, ce qui est sous-linéaire par rapport à la taille de l'entrée n (comme la taille de la matrice utilisée dans le calcul des termes de la suite est fixe, seul la taille de l'exposant importe, ie. partie entière de $(log_2(n) + 1)$).

Pour illustrer tout ce que qui a été évoqué plus haut, nous avons réalisé un graphique où l'évolution du temps de calcul en fonction de la taille de l'entrée est représentée pour chaque méthode :



Les valeurs sont très basses puis très élevées, si bien que même avec une échelle logarithmique il est difficile de voir les variations inhérentes à chaque méthode (l'exécution a été réalisée sur une machine dotée d'un processeur i3 avec 4Go de RAM). Cela étant, on constate bel et bien que la méthode itérative est meilleure (d'un point de vue du temps de calcul) que la méthode récursive, et que la méthode matricielle est plus efficace encore.

2 Mini-Projet 2 (Manipulation de graphes non-orientés)

Au cours de ce projet-ci, nous nous sommes intéressés à la notion de *zonevide* d'un graphe non-orienté. Une zone vide d'un graphe non-orienté $G = (S, A)$ est un sous-ensemble X de S tel que le sous-graphe de G induit par X ne possède aucune arête. Nous avons utilisé la représentation de graphes sous forme de matrices d'adjacence contenant des booléens ($G[i][j] = 0$ s'il n'existe pas d'arête entre les sommets i et j, sinon 1, avec $G[i][i] = 0$ pour éviter les boucles). Nous avons également choisi le langage C pour réaliser ce projet.

Nous avons pour commencer créé une fonction recevant un graphe G et un sous-ensemble X de G en entrées, qui vérifie si X est une zone vide de G :

```
#En entree : une matrice d'adjacence de dimension n et un sous ensemble de sommets de g
#En sortie : Un booleen indiquant si le sous ensemble de sommets est une zone vide ou non
entier i,j
pour i allant de 0    g.n
          si sommet_sous_graphe[i]=1 alors
                    pour j allant de 0    g.n
                              si sommet_sous_graphe[j]=1 alors
                                        si g.A[i][j]=1 alors #on teste si deux sommets du sous
                                            ensemble ont une arete entre eux
                                                  retourne 0
                                        fin_si
                              fin_si
                    fin_pour
          fin_si
fin_pour
retourne 1
```

Les deux boucles *for* imbriquées (allant de 0 à n) octroient, dans le pire des cas, une complexité en $O(n)$, ce qui est linéaire par rapport à la taille de l'entrée ($GRAPHEMAT$ est une matrice de taille $n*n$, et $SOMMET$ est de taille n, donc en tout $n+n$). Le temps de calcul nécessaire à l'execution de cet algorithme est donc moindre (ie. l'exécution est très rapide, comme nous avons pu le vérifier sur plusieurs graphes à l'aide du fichier $q1.c$ fourni dans le répertoire EX-2).

Après quoi on s'intéresse au calcul de zone vide maximale, c'est-à-dire d'un sous-ensemble X qui serait une zone vide de G, sans être inclus dans un sous-ensemble Y également zone vide de G. Cette méthode n'est pas idéale car elle dépend de l'ordre dans lequel les sommets sont "lus" par l'algorithme (il est donc attendu qu'un sommet maximal ne soit pas nécessairement maximum, pour le graphe considéré ; ie. il est possible qu'une autre zone vide plus grande existe dans G).

```
#En entree : Une matrice d adjacence de dimension n, une liste de sommets qui contiendra la
    zone vide maximal, une liste d'entiers representant l'ordre de parcours de sommet
#En sortie : un tableau de sommets representant zone vide maximal
          entier i
    entier j
          SOMMET : liste_sommet[g.n]
          pour i allant de 0 a g.n
                              zone_vide_maximal[i]:=0 #initialisation de la zone vide maximal
                    fin_pour
                    pour i allant de 0 a g.n
                              liste_sommet[i]:=1 #initialisation de la liste des sommets disponibles
                    fin_pour
                    pour i allant de 0    g.n
                              si liste_sommet[liste[i]]=1 alors
                                        zone_vide_maximal[liste[i]]:=1 #ajout    la zone vide maximal
                                        pour j allant de 0    g.n
                                                  si g.A[liste[i]][liste[j]]=1 alors
                                                            liste_sommet[liste[i]]:=0 #suppression
                                                                des voisins de la liste des sommets
                                                                disponible
                                                  fin_si
                                        fin_pour
                              fin_si
                    fin_pour
```

Les deux premières boucles *for* sont executées n fois, mais la complexité de notre algorithme dépend principalement des deux *for* imbriqués qui suivent. En effet, dans le pire des cas la complexité est en $O(n)$, ie. linéaire par rapport à la taille de l'entrée $n + 2n$. En pratique, l'algorithme fonctionne très rapidement, même pour des jeux de données où $n > 10$.

On s'intéresse ensuite à la notion de zone vide maximum, c'est-à-dire un sous-ensemble X qui est une zone vide de G tel qu'il n'existe aucune autre zone vide de G plus grande que X. Pour déterminer une telle zone vide, il est nécessaire de calculer toutes les zones vides possibles de G afin d'en comparer la taille (et déterminer laquelle est la plus grande). Ci-après l'algorithme en question, qui utilise une fonction *combinaison* récursive pour tester toutes les combinaisons possibles de sommets qui constituent des zones vides :

```
#En entree : une matrice d'adjacence de dimension n, un tableau de sommet representant un sous
    ensemble des sommets de g
#En sortie : un tableau de sommet representant une zone vide maximum
zone_vide_maximum (g, zone_vide_maximum[g.n]) :
          entier element[g.n]
          entier liste[g.n]
          entier i
          pour i allant de O    g.n
                    element[i] := 0 #initialisation de la liste de sommets disponibles
          combinaison(g,0,element,liste, zone_vide_maximum)


#En entree : une matrice d'adjacence de dimension n, un entier indiquant le rang actuel la liste d'
    ordre, une liste element disponible pour remplir la liste, la liste d'ordre de parcours de
    sommet, un tableau de sommet representant un sous ensemble X des sommets
#En sortie : un tableau de sommet representant une zone vide maximum
combinaison(g, rank, element, liste[], zone_vide_maximum) :
          entier i
          entier nombre_sommet
          entier nombre_sommet_temp
          SOMMET zone_vide_maximum_temp[g.n]
          si rank >= g.n alors #si la liste d'ordre de parcours de sommet est rempli
                    nombre_sommet:=0
                    nombre_sommet_temp:=0
                    zone_vide_maximal(g,zone_vide_maximum_temp,liste)
                    pour i allant de O    g.n
                              si zone_vide_maximum[i]!=0 alors
                                        nombre_sommet++ #Compte le nombre de sommet de la zone vide
                                            maximum
                              fin_si
                              si zone_vide_maximum_temp[i]!=0 alors
                                        nombre_sommet_temp++ #Compte le nombre de sommet de la zone
                                            vide maximal calcule
                              fin_si
                    si nombre_sommet_temp>nombre_sommet alors #si le nombre de sommet de la zone vide
                         maximal calcule est superieur au nombre de sommet de la zone vide maximum
                              pour i allant de O    g.n
                                        zone_vide_maximum[i]:=zone_vide_maximum_temp[i]
                    fin_si
                    retourne
          fin_si
          pour i allant de O    g.n #remplissage de la liste d'ordre de parcours de sommets
                    si element[i]=0
                              element[i]:=1 #enleve un sommet dans les elements disponible
                              liste[rank]:=i
           combinaison(g,rank+1,element,liste,zone_vide_maximum) #appel recursif
                              element[i]:=0 #remet un sommet dans les elements disponible
                    fin_si
```

Par le cours, on sait que le problème ENSEMBLE DOMINANT (parfois appelé ZONE VIDE) est un problème NP-complet ; il n'existe donc pas à notre connaissance d'algorithme pouvant calculer la zone vide maximum d'un graphe de complexité polynomiale. Notre algorithme est en effet de complexité $O(n^3 n!)$ à cause de l'appel récursif dans la boucle *for* exécutée n fois, et appelant à chaque étape la fonction $zonevidemaximale()$ de complexité en $O(n^2)$ ($n!$ provient de l'appel récursif lui-même, car on modifie la valeur de *rank* à chaque étape). On peut le constater facilement : l'algorithme fonctionne sur de petits jeux de données (voir les fichiers $graph1-4$ dans le dossier EX-2), mais dès que nous essayons d'utiliser des graphes de plus d'une douzaine de sommets ($graph5-6$), l'execution n'est plus réalisée en temps raisonnable (la machine finit par freezer).

Nous avons écrit le fichier $comparaison-zvMaximal-zvMaximum$ pour comparer l'efficacité de l'algorithme implémenté en 2/ et celui en 3/ sur plusieurs graphes de taille allant croissant. Cela nous permet d'illustrer les points abordés plus hauts, c'est-à-dire mettre en évidence le fait que pour $n > 10$, l'algorithme de la question 3/ ne s'execute plus en temps raisonnable, tandis que celui de la question 2/ continue de retourner une solution, et ce très rapidement.

Pour remédier à cela, nous avons tenté de concevoir un algorithme heuristique pour ne pas calculer la totalité des zones vides possibles du graphe (et ainsi déterminer une solution réalisable en temps polynomial, bien que le problème soit NP-difficile). Pour ce faire, nous avons tenté d'adapter l'algorithme du sac à dos (avec élagage de l'arbre de type branch and bound) à notre situation. L'astuce repose sur le tri croissant des sommets du graphe en fonction de leur arité (nombre de sommets voisins reliés par une arète). En effet, l'algorithme du sac à dos avec branch and bound est très efficace si on l'utilise sur des jeux de données triés (par ordre décroissant, classiquement), et permet de retourner une approximation de la solution optimale. Nous n'avons, cependant, pas réussi à l'implémenter.

Cela dit, en théorie, cet algorithme est capable de retourner de meilleurs résultats que celui de la question 2/ car le fait de trier les sommets par arité croissante permet de commencer à créer des sous-ensembles à partir de sommets qui font partie de toute zone vide (arité 0). De plus, comme on ne calcule pas toutes les zones vides

(ce qui fait qu'on peut manquer la solution optimale), il est attendu que cet algorithme soit plus rapide que celui de l'algorithme implémenté en question 3/.

3 Mini-Projet 3 (Simulation d'une Machine de Turing Déterministe)

Au cours du dernier projet, nous avons cherché à implémenter un programme simulant une Machine de Turing Déterministe (MTD). Notre programme reçoit en entrée le programme M devant être simulé et un mot devant être traité par M. Pour commencer, on rappelle qu'une MTD est caractérisée par un quintuplet : un ensemble fin d'états, un alphabet de travail, un état initial, l'ensemble des états finaux (ici accepté ou rejeté), ainsi que la fonction de transition qui spécifie le passage d'un état à un autre.

Nous avons choisi de coder ce programme en java car l'approche objet nous semblait plus abordable pour simuler les différents composants de la machine. La machine est constituée d'un ruban de travail et d'une tête de lecture se déplaçant sur ce ruban (nous avons rassemblé ces deux objets en un seul, appelé *bandeDeLecture*). Les méthodes *avancer*() et *reculer*() de la classe *bandeDeLecture* permettent de représenter le déplacement de la tête de lecture sur la bande (via l'entier *casecourante*, modifié dans les dites-méthodes).

En entrée, la machine reçoit un ruban d'entrée (contenant le mot à traiter) et une fonction de transition, et délivre en sortie un booléen, *true* si le mot est accepté, $false$ s'il ne l'est pas (par le biais de la méthode $fonctionDeTransition.programmePourMTD()$). Ci-après le code de la classe MTD correspondante :

```
class MTD {
    private bandeDeLecture bande;
    private bandeDeLecture bandeDeTravail;
    private fonctionDeTransition fonction;

    public MTD(bandeDeLecture bande, fonctionDeTransition fonction) {
        this.bande = bande;
        this.fonction = fonction;
    }

    void lireRuban() {
        #on copie d'abord le contenu du ruban d'entree sur le ruban de travail
        bandeDeTravail = new bandeDeLecture(bande.ruban.size());
        for (int j = 0; j < bandeDeTravail.ruban.size(); j++){
            bandeDeTravail.ruban.set(j, bandeDeTravail.beta);
        }
        for (int i =0; i<bande.ruban.size(); i++){
            bandeDeTravail.ruban.set(i, bande.ruban.get(i));
        }

        #la machine r alise ensuite le traitement du mot depuis son ruban de travail
        fonction.programmePourMTD(bandeDeTravail);
    }
}
```

Durant le traitement du mot, la MTD copie le contenu du ruban d'entrée sur son ruban de travail, puis effectue les opérations que lui indique la fonction de transition, représentée par la classe $fonctionDeTransition$. Cet objet centralise les différentes fonctions de transition que nous avons crée via la méthode $programmePourMTD$, ce qui permet de mettre en place les conditions d'acceptation ou de refus du mot reçu en entrée.

Nous avons implémenté deux programmes pour MTD : l'un reconnait les mots palindromiques, et le second reconnait les mots de type $x^n y^n$, où n est un entier. Leur pseudo-code est présenté ci-après :

```
mot_palindrome(m) :
    n := longueur m
    tant que n != 0 faire
    case_courante := 0
    lettre <-- m[case_courante]
        tant que case_courante < n faire
            case_courante = case_courante + 1 #on avance sur le ruban
        si m[case_courante] = lettre alors #on efface la derniere et la premiere case du ruban
            effacer m[case_courante]
            effacer m[0]
        sinon rejeter
    accepter


mot_xy(m) :
    n := longueur m
    cx := 0
    cy := 0
    i := 0
    tant que mot[i] = x et i < n faire
        cx := cx + 1
        i := i + 1 #on avance sur le ruban
    tant que mot[i] = y et i < n faire
```

```
        cy := cy + 1
        i := i + 1
    si n = 0 ou mot[0] = B alors
        rejeter
    sinon si (i < n et mot[i] = B) ou i = n alors
        si cx = cy alors #si le nb de x est egal au nb de y lus
            accepter
        sinon
            rejeter
    sinon
        rejeter
```

Ces deux programmes sont executables par le biais d'une entrée utilisateur lors de l'execution du programme du simulateur. Pour tester leur efficacité il est possible d'entrer les mots "tenet" et "tente" (pour le programme *motpalindrome*) ainsi que "xxxyyyBB" et "xxyyyB" (pour *motxy*).

D'un point de vue complexité, le premier algorithme est, dans le pire des cas (ie. lors que le mot est effectivement un palindrome) de complexité en $O(n)$, où n est la longueur du mot. Quant au second algorithme, dans le pire des cas (ie. il y a $n/2$ x et $n/2$ y), la complexité est également en $O(n)$. De ce fait, la simulation de MTD par notre programme est de complexité en $O(3n)$ (remplissage du ruban de travail et copie du ruban d'entrée sur celui-ci). Nous avons par exemple testé *motxy* sur des mots de grande taille et l'execution reste quasi instantanée, ce qui semble aller dans ce sens également.

Pour finir, bien que nous ayons réussi à simuler le fonctionnement d'une MTD, nous n'avons cependant pas réussi à concevoir des attributs représentant l'alphabet ou l'ensemble des états de la machine (nous avons uniquement une association de l'état d'acceptation à $retournervrai$ et celui de l'état de refus à $retournerfaux$). Une possibilité pourrait être de fournir une liste de caractères en attribut des fonctions de transition, dont la première opération consisterait à vérifier que le ruban d'entrée ne contient pas de caractères ne faisant pas partie de l'alphabet considéré).