INFOB237 Rapport de projet

Aline Boulanger, Aurélie Genot, Yannis Van Achter

5 mai 2023

Table des matières

1 Introduction

Pour commencer, nous nous sommes répartis les différentes tâches. Nous avons décidé de prendre chacun un des trois problèmes en charge et de faire des réunions régulières afin de mettre en commun notre avancement. Les parties uniques de chaque point étaient réfléchies en groupe.

2 Calendrier de projet

Avant le pré-blocus (début avril), nous avons commencé à comprendre et analyser le projet ensemble. Afin de pouvoir avancer séparément durant le pré-blocus (car nos horaires n'étaient pas les mêmes durant celui-ci), nous nous sommes ensuite **réparti les tâches** en se concentrant **chacun sur un des trois problèmes** à résoudre.

Le calendrier original était le suivant :

1. Pour la fin du pré-blocus (pour le 16 avril) : avoir fini une partie des points décrits dans l'énoncé de chaque problème (analyse, spécifications JML, algorithme naïf...) Nous avions alors décidé de travailler sur les points communs à chaque énoncé pour par la suite réfléchir ensemble aux points propres à chaque exercice (donc les algorithmes basés sur "diviser pour régner", "la programmation dynamique" et aussi "l'approche gloutonne").
2. Semaine de la rentrée (17 avril) : 2ème réunion pour faire l'état d'avancement de nos points respectifs, résoudre ensemble les problèmes restants rencontrés sur ces derniers et réfléchir aux solutions finales. Nous avons ensuite retravaillé chacun sur notre algorithme pour le 24 avril.
3. Semaine du 24 au 30 avril : résolution ensemble des derniers problèmes restants, création du rapport ensemble.
4. Semaine du 1 mai au 6 mai : finalisation du rapport, fin des tests sur les algorithmes et remise du projet.

De manière générale, ce projet a été conduit avec une méthodologie semi-agile nous permettant d'avancer un maximum pendant le pré-blocus et ce de manière individuelle. Par la suite, nous sommes partis sur une méthodologie agile scrum pour garder un contact régulier avec le projet et être sûrs que ce dernier avance suffisamment pour la deadline du 6 mai.

3 Démarche générale et algorithmes

3.1 Méthodologie

Nous avons pris connaissance de l'énoncé et remarqué que nous devions réaliser un algorithme "Diviser pour régner", "Programmation Dynamique" et "Glouton". Notre méthodologie de travail a suivi le canva des étapes à suivre décrites dans chaque énoncé (le point "Analyse, Spécification et Implémentation".)

Tout au long du projet, nous nous sommes aidés de notre cours et de nos travaux pratiques, particulièrement pour les spécifications JML et l'implémentation des algorithmes.

3.2 L'invasion de plante carnivore

Le but de cet exercice est de voir si une plante est envahissante (c'est-à-dire si pour le nombre de plantes M d'une rangée du champ, une plante apparait strictement plus de M/2 fois). Ainsi, s'il n'y a qu'une seule plante, celle-ci est considérée comme étant envahissante.

Pour résoudre cet exercice, un fichier contenant l'ensemble des fleurs du champ est pris en paramètre.

3.2.1 Choix de l'algorithme

L'invasion de plante carnivore est un problème de type **"diviser pour régner"**. En effet, cette approche permet de diviser le problème en sous-problèmes plus simples et de résoudre ces derniers. Il faut ensuite combiner les solutions trouvées aux différents sous-problèmes pour obtenir la solution finale. L'idée est ici de comparer les envahisseurs trouvés pour finalement parvenir à trouver l'envahisseur final s'il y en a un.

3.2.2 Problèmes rencontrés

Si la solution naïve n'a pas tellement posé de problèmes, ce n'est pas le cas de la solution "diviser pour régner" qui a posé plusieurs difficultés au niveau de la récursivité.

En effet, la solution ne donnait pas toujours le bon envahisseur. Nous avons donc essayé de faire la partie comptant les occurrences des plantes grâce à un dictionnaire (où les clés représentaient chaque sorte de fleur et les valeurs l'occurrence de chaque fleur). Cela posait toujours des difficultés, ce pourquoi nous nous sommes finalement tournés vers une boucle parcourant les éléments restants du tableau pour compter les occurrences des éléments.

3.2.3 Solution naïve

La classe **"solutionNaive"** comprend 3 méthodes permettant la résolution du problème. La méthode **"main"** crée un tableau pour chaque rangée du champ de fleurs sur base du fichier donné (ici *"champ.txt"*) et appelle la méthode **trouverEnvahisseur**. Cette dernière est la méthode de résolution de l'algorithme. Celle-ci regarde pour chaque fleur du tableau créé si celle-ci est envahissante. Pour ce faire, elle appelle la méthode **compterOccurence** afin de compter combien de fois apparait la fleur sur la rangée. Au sein de *trouverEnvahisseur*, une ArrayList "fleurEnvahissante" récupère le potentiel envahisseur. C'est une liste de String qui contiendra toujours 0 ou 1 seul élément, respectivement, s'il n'y a aucun envahisseur ou s'il y en a un.

Au niveau de **la complexité de cette solution**, c'est comme si on avait deux boucles imbriquées. En effet, on ne prend pas en compte la partie de lecture de fichier et on a que :

— *trouverEnvahisseur* appelle au sein d'une boucle for n fois compterOccurence (pour les n éléments du tableau)
— *compterOccurence* comprend aussi une boucle for qui itère n fois (sur les n éléments du tableau).

On obtient donc une complexité de $\mathcal{O}(n^2)$

Comme indiqué pour la complexité, cette solution comprend **deux boucles** for. La première, au sein de *trouverEnvahisseur* est une boucle for each qui parcourt l'ensemble des éléments appartenant à *fleurs*, le tableau de strings non nul contenant toutes les fleurs d'une rangée pris en paramètre. On a :
— Variant de boucle : passage à l'élément suivant
— Invariant de boucle : fleur $\in$ fleurs
— Terminaison : quand tous les éléments fleur $\in$ fleurs ont été parcouru
On retrouve une deuxième boucle for au sein de *compterOccurence*. Celle-ci parcourt l'entiéreté du tableau de strings non nul, *tab*, pris en paramètre. On a :

— Variant de boucle : i (i++)
— Invariant de boucle : $0 \leq$ i $<$ tab.length
— Terminaison : i $<$ tab.length

Ainsi, on boucle de i=0 lors du premier passage jusqu'à i $<$ tab.length à la terminaison. i est incrementé à chaque passage.

3.2.4 Solution finale

La solution finale pour ce problème est donc de type "Diviser pour régner". Elle suit le canva de l'exemple "Tri par Fusion".

La classe **"diviserPourRegner"** comprend 3 méthodes. La méthode **"main"** appelle la méthode **"lectureFichier"**. Cette dernière permet de lire le fichier contenant l'ensemble des fleurs du champ (*"champ.txt"*). Elle crée un tableau pour chaque rangée de fleurs du champ et appelle la méthode "**trouverEnvahisseur**" qui permet de trouver récursivement le potentiel envahisseur. *"lectureFichier"* renvoie une ArrayList de strings contenant les envahisseurs trouvés par *"trouverEnvahisseur"*.

trouverEnvahisseur est une méthode récursive. Comme cas de base, on a que s'il n'y a qu'un seul élément, alors c'est l'envahisseur (puisqu'il apparait alors strictement plus que M/2 fois). Sinon, on fait du diviser pour régner.

La complexité de *trouverEnvahisseur* est de $\mathcal{O}(nlog_2(n))$.

On a en effet :

— la division du tableau en deux, jusqu'à ce que cela ne soit plus possible $\mathcal{O}(log(n))$;
— la recherche récursive d'une plante envahissante ;
— la combinaison des éléments trouvés pour obtenir l'envahisseur final s'il y en a un grâce à une boucle for qui compare les éléments trouvés avec le reste des éléments du tableau $\mathcal{O}(n)$;

3.3 Le roi des bleus

Afin de résoudre le problème de Frank, il nous était demandé de faire un algorithme basé sur la programmation dynamique. Celui-ci est faisable sous sa forme récursive (solution naïve). Cependant, après plus d'une semaine sur la solution finale nous nous sommes concertés avec un autre groupe et en sommes arrivés à la conclusion suivante : **il n'y a pas de solution optimale utilisant la programmation dynamique**. En effet, si cela fonctionne pour trouver le minimum, cela n'est pas possible ici pour le maximum.

Remarque : dans cette section, nous ferons une généralisation entre les mots : matrice, tableau, carte afin de représenter le sol du damier où Frank tente la couronne de Roi des Bleus (et fait son parcours de recherche de bières).

Remarque' : dans le dossier *./Roi-des-bleu* se trouve le fichier *ANSWER.md* qui comprend toutes les réponses aux questions (Complexité, analyse de l'énoncé et explications des fichiers présents dans le dossier)

Remarque" : Quand nous parlerons de complexité, 'n' vaut ici le résultat du nombre de ligne $*$ nombre de colonne soit : n = N°Ligne $*$ N°Colonne
En effet, cela représente la taille du problème.

3.3.1 Choix de l'algorithme

Une fois, notre conclusion sur l'impossibilité de faire un algorithme dynamique établie, il a fallu partir sur une autre méthode, un autre paradigme algorithmique. Nous nous sommes alors tournés vers celui du **"générer et tester"** qui n'est pas utilisé explicitement pour le projet.

3.3.2 Problèmes rencontrés

Pas de gros problème rencontré mise à part sur le choix de l'algorithme comme expliqué précédemment.

3.3.3 Solution naïve

La solution naïve est donc basée sur de la programmation dynamique et récursive. Nous commençons donc par le début de la matrice en i=0 et j=0. A chaque itération, nous allons un cran plus loin dans la matrice et dans l'ordre suivant (en partant de i=0, j=0).

— récupération de la meilleure solution pour l'énoncé à l'horizontale
— récupération de la meilleure solution pour l'énoncé à la verticale
— récupération de la meilleure solution pour l'énoncé à la diagonale

Quand le tableau est totalement parcouru, la récursion s'arrête et donne la solution optimale aux coordonnées courantes pour le trajet de Frank.

L'algorithme étant basé sur l'exploration (récursive) des trois directions possibles, sa **complexité** est donc de $\mathcal{O}(3^n)$.

Etant donné qu'il n'y a aucune boucle dans cette solution, il n'y a donc aucune preuve à faire ici.

3.3.4 Solution finale

Notre solution finale est donc basée sur le **"générer et tester avec mémoïsation"**. Afin de bien comprendre l'algorithme, il faut d'abord comprendre le choix de la structure de données pour représenter la carte.

Nous avons donc un dictionnaire avec, pour clés, les coordonnées stockées en string sous la forme "(x, y)". A chaque clé est donc stockée une liste contenant à l'origine un élément unique : le nombre de bières à cette case. Vous pouvez alors vous poser la question, à juste titre, pourquoi ne pas associer un type Integer plutôt qu'un type ArrayList<Integer> ?

La réponse est simple, pour chaque liste, nous allons ajouter l'ensemble des possibilités basées sur la case suivante (ou précédente car nous démarrons de la fin). L'image suivante montre bien ce

principe :

FIGURE 1 – Chemin choisi par l'algorithme, partant de la fin du tableau et allant au début

Comme montré sur la figure 1, nous partons de la fin du tableau et explorons l'ensemble des possibilités.

Une fois que nous avons terminé l'exploration, nous nous reconcentrons sur la liste à l'endroit (0, 0) de la carte. En effet, c'est ici que se trouve la liste contenant les solutions potentielles, il nous reste alors à parcourir la liste de l'indice 1 (éviter de prendre la valeur originale de la case) jusqu'à la fin et sélectionner la valeur maximum restant sous la limite de bières imposée dans le fichier contenant les cartes.

Tout cela prend du temps et dépend de la taille de la carte. On a donc :
— La génération des possibilités de compléxité $\mathcal{O}(n)$
— L'ajout des valeurs précédentes au sein de la liste est de complexité $\mathcal{O}(n)$ également. On le fait donc trois fois en considérant chacune des directions possibles

Nous avons donc finalement comme **complexité de l'algorithme** : $\mathcal{O}(n * (n + n + n)) = \mathcal{O}(3(n^2)) = \mathcal{O}(n^2)$.

3.4 Le marchand

3.4.1 Choix de l'algorithme

Afin de résoudre le problème du marchand, il nous était demandé de réaliser **un algorithme basé sur une approche gloutonne**. En effet, ce type d'algorithme est utilisé pour résoudre des problèmes d'optimisation. Choisir une approche gloutonne correspond à faire un choix qui semble être le meilleur pour l'instant (qui aurait donc le plus de chance d'offrir une solution optimale). Cependant, une approche gloutonne ne permet pas de garantir à chaque fois que la solution trouvée sera optimale. Si cela n'est pas le cas, alors elle offrira, tout de même, une approximation intéressante.

En ce qui concerne le problème du marchand, nous avons décidé de partir sur **la propriété gloutonne suivante** : "choisir l'objet ayant le meilleur rapport valeur/poids, sans dépasser le poids maximum autorisé".

Pour la décortication de l'énoncé du problème du marchand, nous avons : cet algorithme prendra en entrée un fichier contenant plusieurs cas de test. Pour chaque cas de test, il y un nombre de produits possibles à vendre (= N), le poids maximum à ne pas dépasser (= M), les valeurs de chaque produit (= I) et les poids de chaque produit (= J).

But de cet algorithme : trouver la solution optimale afin de permettre au marchand de faire un maximum de profit en choisissant les produits les plus rentables tout en veillant à ne pas dépasser le poids maximum que son sac puisse porter.

3.4.2 Problèmes rencontrés

Etrangement, nous avons eu plus de difficultés à réaliser la solution naïve que la solution gloutonne. En effet, pour la solution naïve, nous étions d'abord partis sur une création de sous-listes pour ensuite itérer sur celles-ci mais l'algorithme nous renvoyait la solution *"0 0"*,alors que la lecture du fichier se passait sans le moindre problème. Finalement, nous avons donc décidé de changer de méthode. C'est cette dernière qui est expliquée dans le point suivant (Solution naïve).

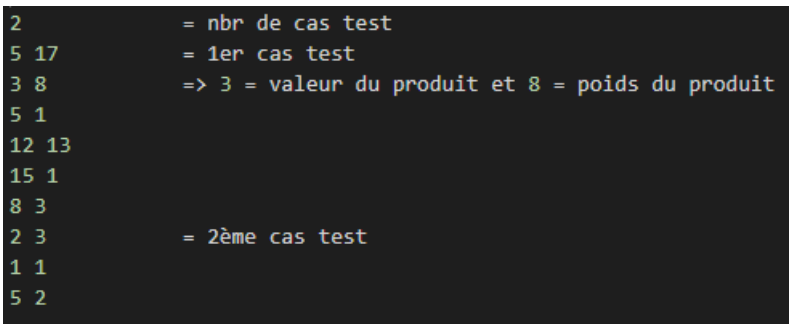

FIGURE 2 – Exemple de fichier d'entrée décortiqué

3.4.3 Solution naïve

Premièrement, à l'intérieur de **la classe "MarchandNaif"**, se trouve la méthode *"main"* qui est le point de départ de tout programme Java. En premier lieu, dans cette méthode, nous lisons le fichier (ici nommé *"Marchand.txt* (conforme à celui se trouvant dans l'énoncé du projet). Pour cela, **nous parcourons le fichier** afin de récolter diverses informations comme le nombre de cas tests se trouvant dans le fichier (ici représenté par l'entier n), le poids et la valeur de chaque produit disponible (respectivement représentés par les entiers i et j). Afin de lire ce fichier, nous utilisons la classe *"Scanner"* qui nous permet de lire les données de la console ou de toute autre source d'entrée. Ici, nous l'utilisons pour extraire les données en parcourant le fichier que nous recevons en entrée. Après avoir récolté ces informations importantes pour la suite, nous faisons simplement appel à la méthode *"findMaxValueNaif"*.

Dans cette méthode *"findMaxValueNaif"*, nous commençons par trier le sac qui contient les produits possibles en fonction de leur rapport valeur/poids grâce à *"Collections.sort"*. Ensuite, après avoir initialisé value à 0 (qui sera la valeur que le marchand pourra transporter dans son sac), nous itérons sur chaque item (désignant les produits) se trouvant dans le sac et nous ajoutons la valeur de ceux-ci dans value si et seulement si le poids max n'est pas dépassé. Pour finir, nous retournons juste value.

En ce qui concerne la complexité de l'algorithme naïf, elle est de $\mathcal{O}(n)$ car nous parcourons une fois une ArrayList. En effet, ne sachant comment est implémenté *"collection.sort"*, nous ne pouvons pas en connaitre la complexité.

Pour les invariants de boucles ainsi que leurs preuves :

for (Items item : bag)
— Invariant $\Longrightarrow$ item $\subset$ bag && item $\in$ Items
— Variant $\Longrightarrow$ Passage à l'item suivant
— Terminaison $\Longrightarrow$ lorsque tous les item $\in$ Items ont été parcouru

Symboles utilisés pour les preuves :
— $\in$ signifie "appartient"
— $\subset$ signifie "se trouvant"
— && signifie "et"

Afin de réaliser cette solution naïve, nous avons eu besoin d'implémenter **la classe "Items"** qui nous a permis d'implémenter un comparateur ainsi que quelques méthodes qui nous permettent d'obtenir le poids et la valeur d'un produit.

3.4.4 Solution finale

La solution finale du problème du marchand est donc sous forme "gloutonne".

Tout d'abord, à l'intérieur de **la classe "Glouton"** se trouve la méthode *"main"*. Dans celle-ci, nous lisons le fichier (toujours "Marchand.txt). La méthodologie utilisée est la même que celle expliquée dans le point précédent (Solution naïve). Après cela, nous faisons simplement appel à la méthode *"findMaxValue"* pour chaque cas test.

Ensuite, nous avons la méthode *"findMaxValue"* qui nous permet de remplir le sac du marchand avec des produits de telle sorte que le marchand fasse un maximum de profit tout en ne dépassant pas le poids maximum qu'il peut porter. Dans cette méthode, nous avons **déclaré une ArrayList nommée products** où nous allons mettre tous les produits se trouvant dans le cas test que l'on est en train de vérifier. Ensuite, nous avons décidé de **trier cette ArrayList** dans l'ordre décroissant en fonction du rapport valeur par poids de chaque produit. Nous avons **l'étape d'initialisation** qui nous permet d'initialiser toutes les variables qu'on va avoir besoin à 0. Pour finir, il y a **un if imbriqué dans une boucle afin de pouvoir aller chercher la valeur et le poids de chaque produit contenu dans l'ArrayList products et de l'ajouter dans le sac** si celui-ci ne fait pas passer le poids du sac du marchand au-dessus de la limite que celui-ci peut porter. A la fin, on **retourne la valeur maximale** que le marchand peut avoir.

Au niveau de la **complexité de cet algorithme glouton**, elle est de $\mathcal{O}(nbritems)$
— $\mathcal{O}(nbritems)$ = complexité de la boucle for
— ($\mathcal{O}(n)$= complexité de la boucle for each)
— $\mathcal{O}(nbritems)$ = complexité de la boucle for et du if imbriqué

Afin de réaliser cette solution gloutonne, nous avons de nouveau eu besoin de **la classe "Items"** dont l'explication se trouve dans le point précédent (Solution Naïve).

4 Conclusion

Ce projet nous a permis de mettre en pratique ce que nous avons vu au cours d'algorithmique mais également en "Conception et programmation orientée objet" pour la partie implémentation en Java. Nous avons pu nous familiariser avec les différents types d'algorithmes ainsi que découvrir Latex lors de la rédaction de ce rapport.

Nous avons globalement respecté notre calendrier de projet malgré quelques difficultés qui nous ont fait prendre un peu de retard et nous sommes satisfaits de la méthodologie adoptée.