Rapport du projet de programmation #2

Part 1 :

Tout au long du processus de vérification du résultat, mon code a échoué quelques tests mais rapidement j'ai pu connaitre la source du problème , ce qui m'a permis alors d'améliorer mon code pour qu'il puisse être correct . Ainsi, la solution produite par mon code, pour la **mémorisation des états en utilisant la Big Table , a passé tous les tests donnés par le professeur mais également sur le Online Judge avec un temps moyen d'exécution qui est optimisé et qui est de 0.250 s .**

My Submissions

#	Problem	Verdict	Language	Run Time	Submission Date
25811729	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-05 03:55:47
25811705	10261 Ferry Loading	Accepted	JAVA	0.250	2020-12-05 03:54:30

Figure 1 : *Big Table memoization*

Part 2 :

Après quelques tentatives ratées et une amélioration du code , j'ai pu produire une bonne et efficace solution pour **la mémorisation des états en utilisant une table de hachage cette fois, et qui passe également tous les tests** donnés par le professeur mais également le Online Judge avec un temps moyen d'exécution moyen de **0.145 s** pour une économie de mémoire par un facteur de **50** et un temps moyen d'exécution moyen de **0.175 s** pour une économie de mémoire par un facteur de **100** , ce qui est très rapide.

My Submissions

#	Problem	Verdict	Language	Run Time	Submission Date
25793927	10261 Ferry Loading	Accepted	JAVA	0.150	2020-12-01 21:19:15
25793926	10261 Ferry Loading	Accepted	JAVA	0.140	2020-12-01 21:19:02

Figure 2 : *Hash Table memoization size/50*

My Submissions

#	Problem	Verdict	Language	Run Time	Submission Date
25793931	10261 Ferry Loading	Accepted	JAVA	0.170	2020-12-01 21:21:06
25793930	10261 Ferry Loading	Accepted	JAVA	0.180	2020-12-01 21:20:55

Figure 3 : *Hash Table memoization size/100*

Part 3 :

Tout d'abord , le but principale de l'utilisation de Hash Table dans cet exercice est de rendre plus efficace la vérification des états visités en faisant de bons choix de clés-valeurs , et de la dimension ainsi que de la fonction de hachage , ce qui permettra ainsi une réduction considérable des collisions et donc une exécution plus rapide .

En effet , j'ai décidé d'utiliser HashTable de la librairie java et de choisir une fonction de hachage qui prend comme arguments la valeur de K et de S qui vont ensuite être utilisées afin de faire un calcul pour produire un résultat unique . Le calcul consiste à multiplier K par S et diviser le tout par S+5 et le tout modulo la taille de la table de hachage , la raison de ce choix est la suivante : si on prend par exemple l'état (100,2500) et l'état (2500,100) , si on avait seulement fait K*S on aurait dans les deux cas le même résultat ainsi pour éviter cela on divise par S+5 afin d'obtenir une solution unique et pour éviter le cas où S=0 on ajoute 5 pour ne pas voir de runtime error . De plus , ce qui justifie l'utilisation du modulo c'est le fait qu'on veut des valeurs comprises dans la table de Hachage . Ainsi , le résultat obtenu par cette fonction qui va être utilisé comme clé dans ma table de hachage avec valeur 0 ou 1 signifiant si l'état (K,S) a été visité ou pas. En ce qui concerne la dimension de la table de hachage , j'ai opté au début pour une taille (N*L)/100 ensuite avec (N*L)/50 (N nombre de voitures et L la longueur du ferry) , la taille est calculée à l'aide de la méthode calculateSizeHashTable() qui donne ainsi une taille beaucoup plus petite que celle de la big table , ce qui veut dire une utilisation de mémoire plus faible .

De plus , pour l'implémentation la mémorisation des états avec la table de hachage à l'intérieur de la méthode BackTrackSolve , j'utilise des méthodes qui aident à la réalisation de cela telle que la méthode present(int K , int S) qui prend comme arguments K et S qui vérifient si la résultat donné par la combinaison (K,S) existe déjà dans la table , ainsi que la méthode setPresent(int K , int S) qui ajoute une combinaison (K,S) dans la table après avoir été visité.

En ce qui concerne l'implémentation générale , j'ai essayé d'éviter l'utilisation des boucles . Par exemple : pour connaitre l'espace restant dans le côté droit, je calculais à chaque fois la somme des tailles des voitures ce qui rendait l'exécution plus lente, mais une fois enlevé et remplacé par une variable qui calcule au fur et à mesure ceci , j'ai pu gagner un peu de temps. De plus, j'ai aussi éviter les opérations de complexité O(n^2) qui ralentissent considérablement le temps d'exécution et qui permettent d'éviter de dépasser le temps d'exécution maximale du Online Judge . Ainsi , j'ai fait plusieurs tentatives avant d'arriver à cette solution finale après avoir eu des erreurs de compilation et des erreurs de runtime , mais j'ai pu rapidement régler cela en quelques lignes de code .

On conclue donc que le choix d'une bonne table de hachage et d'une bonne fonction de hachage étaient des décisions de design importantes dans ma solution qui ont permis d'éviter trop de collisions et être alors utile aux performances et rassembler entre **rapidité d'exécution et utilisation de la mémoire .**