

# Report :

## Partie A:

Le code fourni implémente avec succès une solution au problème

**Ferry Loading** utilisant une méthode de le retour-arrière (backtracking) et une "Big Table" pour la mémorisation. La solution a passé le test du Virtual judge en ligne avec les caractéristiques suivantes :

1. **Correspondance des Résultats** : Le code produit la sortie correcte pour tous les cas de test.
2. **Gestion des Cas Limites** : Le code gère correctement les cas spéciaux(edge cases), comme lorsque aucune voiture ne peut être placée ou lorsque la capacité du ferry est inférieure à la longueur de la première voiture.
3. **Gestion des États** : L'utilisation d'une table `visited` garantit que les états ne sont pas revisités, ce qui améliore l'efficacité en évitant les calculs redondants.


### Performances :

- **Temps d'Exécution** : La solution a été acceptée avec un temps d'exécution de **1730 ms**, ce qui est efficace compte tenu des contraintes du problème et de la nature récursive de l'algorithme.
- **Analyse de la Complexité** :
  - **Complexité Temporelle** : L'algorithme utilise le backtracking avec mémorisation, donc la complexité dans le pire des cas est  $O(n \times L)$  où  $n$  est le nombre de voitures et  $L$  est la longueur du ferry en centimètres.
  - **Complexité Spatiale** : La mémoire utilisée inclut la table `visited` ( $O(n \times L)$ ), les tableaux pour la meilleure solution et la pile de récursion.

### Virtual judge test:

- Statut : **Accepté**
- Temps d'Exécution : **1730 ms**

### Capture d'Écran:

 hibaahansal	UVA 10261	Accepted	1730	3522	Java	71 min ago
---	-----------	----------	------	------	------	------------

## Partie B:

Le code utilise une **table de hachage** pour mémoriser les états visités, réduisant considérablement la taille de la mémoire par rapport à l'implémentation avec la "big table". **Correspondance des Résultats** : La sortie est correcte et passe tous les tests du juge en ligne.

1. **Amélioration de l'Efficacité** : L'utilisation de la table de hachage permet une gestion plus rapide et efficace des états visités grâce à une fonction de hachage optimisée et à la gestion des collisions par **sondage linéaire**.

#### Virtual judge Test:

- **Statut : Accepté**
- **Temps d'Exécution : 350 ms** (comparé à l'implémentation précédente avec une "big table", qui prenait 1730 ms).
- **Réduction de la Taille de la Table** :
  - La taille de la table de hachage est déterminée par  $L+n$  (longueur du ferry en centimètres + nombre de voitures). Cela réduit significativement l'espace mémoire nécessaire par rapport à une "big table", qui nécessitait  $O(n \times L)$  d'espace.
  - **Table Initialisée avec (-1)** : Cette approche permet de vérifier facilement si une entrée est occupée.

#### Capture d'Écran:

	hibaaahansal	UVA 10261	Accepted	330	4884	Java	42 min ago
---	--------------	-----------	----------	-----	------	------	------------

L'approche avec une table de hachage repose sur des principes fondamentaux, notamment une fonction de hachage efficace et une gestion des collisions optimisée, détaillés dans la partie C.

## Partie C:

#### Fonction de Hachage :

- La fonction de hachage  $(key+value)\%hashTableSize$  utilise le module pour garantir que les indices restent dans la plage de la table.
- La fonction combine key (l'indice actuel currK de la voiture à embarquer) et value (l'espace restant dans la traversier) pour représenter un état unique ce qui conduit à une distribution relativement uniforme, ce qui minimise les **collisions** puisque dans notre cas avoir la même valeur pour la combinaison key +value est rare puisque notre qui est la position sera toujours différent et l'espace restant de le traversier varie .

### Hash Table Size :

hashTableSize = L + integers.size();

### Raison du choix :

- On combine l'espace disponible (**L**) et le nombre de voitures pour obtenir une estimation raisonnable du nombre total d'états uniques à stocker.
- Cette taille garantit que la table est suffisamment grande pour réduire le risque de collisions tout en restant optimisée pour la mémoire.

### Gestion des Collisions :

- Lorsqu'un conflit est détecté, un sondage linéaire est utilisé pour trouver la prochaine position libre sur la table ceci est implémenté par la méthode `find(int key,int value)` et aussi on optimise cela en initialisant notre **stateTableHash** avec **-1** . Cette approche permet de vérifier facilement si une entrée est occupée , si c'est le cas faire une gestion de collision.
- Le choix de gérer les collisions par sondage linéaire repose sur sa simplicité d'implémentation et son efficacité dans des scénarios où les collisions sont rares. Par ailleurs, notre fonction de hachage produit des résultats quasi-unique, ce qui minimise significativement le risque de collisions.

### Insertion et Recherche :

- **Insertion** : implémenter la méthode `find(int key,int key)` . On ajoute un état uniquement s'il n'est pas déjà présent dans la table.
- **Recherche** : Vérifie si un état a déjà été traité, évitant ainsi les calculs redondants.

### Avantages de l'Approche :

- **Rapidité** : Avec un temps d'exécution de **350 ms**, cette implémentation est nettement plus rapide.
- **Efficacité Mémoire** : La table de hachage occupe beaucoup moins de mémoire par rapport à une "big table".
- **Simplicité** : L'approche reste facile à comprendre et à maintenir grâce à des opérations standard de hachage et de gestion des collisions.