**Brute force solution :**

1. Présentation de l’algorithme : présenter son objectif, son contexte d’utilisation et son importance dans le domaine concerné. Parler de son application et de ses domaines d’utilisation

L’objectif de cet algorithme est de choisir la sélection la plus rentable aux bout de deux ans parmis une liste d’actions. Les critères sont les suivants :

- chaque action ne peut être achetée qu’une seule fois

- on ne peut pas acheter une fraction d’action

- le budget maximum est de 500€

Il faut donc écrire un algorithme qui détermine la taille de échantillon d’action et retourne la solution qui générera le plus de plus value.

1. Déscription : expliquer comment fonctionne l’algorithme de façon générale il faut utiliser des explications simples et des exemples.

La fonction calculGain prend en paramètre un tableau de tuple. Chaque tuple représente une action et contient en première colonne son prix et en deuxième colonne son pourcentage de bénéfice après 2 ans. La fonction calcul la valeur nette de l’action après deux ans et ajoute au tuple cette valeur nette.

La fonction testSampleSizeSolutions teste toutes les combinaisons d’action possible en partant du plus grand échantillon et en allant vers le plus petit. Pour chaque combinaison, on calcule le somme total du coup des actions. Lorsque l’on atteint une taille d’échantillons qui permet que la somme des actions est inférieure à 500€, on stocke toutes les solutions possibles dans un tableau.

La fonction findMaxGain boucle sur le tableau de solution retournées par la fonction précédente. Pour chaque solution, si le gain total est supérieur à bestSolution, alors c’est cette solution là qui prend la valeur de bestSolution et qui est retenue.

On a donc un algorithme qui :

- calcule la plus valeur pour chaque actions au bout de deux ans

- calcule la taille optimum d’un échantillon afin d’acheter le plus d’actions possibles avec un budget maximum de 500€

- retourne la solution générant le plus de gain possible parmis toutes les solutions retenues

1. Analyse de la complexité : détaillez l’analyse de la complexité temporelle et spatiale (meilleurs cas, pires cas et cas moyens). Utiliser la notation BigO.
2. Avantages et inconvenients : présenter les avantages et les inconvénients de l’algorithme. Parler des ses forces et de ses faiblesses par rapport à d’autres approches possibles --> rapidité, simplicité, fiabilité

L’avantage principale de cette approche est qu’elle teste toutes les solutions possibles et retourne la plus rentables parmi toutes. En l’utilisant, on est sûr d’indiquer au client la solution qui maximisera le plus son investissement pour un budget maximum de 500€.

L’inconvenient de cette méthode est qu’elle est gourmande en temps et en mémoire car toutes les possibilités sont testées et comparées une à une jusqu’à l’obtention de la réponse qui convient.

1. Montrer une étude de cas

Voir démonstration

1. Conclusion

En conclusion, cette méthode retourne la solution optimale car elle teste toutes les solutions possibles et les comparent pour obtenir la meilleur. L’inconvenient de cette approche est qu’elle est très gourmande en temps et en espace mémoire.

**Optimized solution :**

1. Présentation de l’algorithme : présenter son objectif, son contexte d’utilisation et son importance dans le domaine concerné. Parler de son application et de ses domaines d’utilisation

L’objectif de cet algorithme est de choisir la sélection la première solution rentable aux bout de deux ans parmi une liste d’actions. La contrainte de temps nous oblige à retourner la première solution qui remplie les critères. Toutes les options ne doivent pas être explorées. Les critères sont les suivants :

- chaque action ne peut être achetée qu’une seule fois

- on ne peut pas acheter une fraction d’action

- le budget maximum est de 500€

Il faut donc écrire un algorithme qui détermine la taille de échantillon d’action et retourne la première solution qui dont le budget passe sous le seuil des 500€.

1. Déscription : expliquer comment fonctionne l’algorithme de façon générale il faut utiliser des explications simples et des exemples.

La fonction calculGain prend en paramètre un tableau de tuple. Chaque tuple représente une action et contient en première colonne son prix et en deuxième colonne son pourcentage de bénéfice après 2 ans. La fonction calcul la valeur nette de l’action après deux ans et ajoute au tuple cette valeur nette.

La fonction testSampleSizeSolutions, qui prend en paramètre le tableau retourné par la fonction précédente, teste toutes les tailles échantillon possible jusqu’à trouver la première solution qui demande un investissement initial de moins de 500€.

1. Analyse de la complexité : détaillez l’analyse de la complexité temporelle et spatiale (meilleurs cas, pires cas et cas moyens). Utiliser la notation BigO.
2. Avantages et inconvenients : présenter les avantages et les inconvénients de l’algorithme. Parler des ses forces et de ses faiblesses par rapport à d’autres approches possibles --> rapidité, simplicité, fiabilité

L’avantage de cette méthode est qu’elle est très rapide car elle ne teste pas toutes les solutions possibles et retourne la première réponse qui correspond aux critères d’acceptations.

Inconvénient est qu’elle ne retourne pas la solution la plus rentable.

1. Montrer une étude de cas

Voir démonstration

1. Conclusion

En conclusion, cette méthode permet de retourner la sélection d’actions qui remplis les critères exigés par l’entreprise. Elle retourne la solution qui convient mais ce n’est pas nécessairement la solution optimale.

**Brute force :**

calculGain :

1. Le fait de parcourir la liste d’actions prend linéairement du temps --> O(n) où «n» est le nombre d’éléments dans la liste d’actions
2. Pour chaque tuple on effectue un des calculs constants qui ne contribuent pas à accroître la complexité globale
3. L’ajout du tuple à la liste globale prend également du temps donc on a O(n) pour chaque tuple
4. Le faite de renvoyer la liste complète prend également O(n)

Toutes ces actions sont dominées par l’étape de parcours de la liste d’actions --> O(n) où «n» est le nombre d’éléments dans la liste d’actions. Les étapes pour chaque action contribuent à accroître la complexité mais elles sont constantes par rapport au nombre d’action

Donc on a O(n) où «n» est le nombre d’actions dans la liste

testSampleSizeSolutions :

1. On parcours la plage de valeurs en ordre décroissant de 20 à 1 : cela prends un temps constant donc O(1)
2. Générer toutes les combinaisons possibles : cela dépend du nombre d’élément dans le tableau array et de la valeur actuelle de la plage. La complexité de combinations est exponentielle --> O(2^n), où «n» est le nombre d’éléments dans la liste array
3. On doit additionne la somme de tout les premiers éléments de chaque tuple --> linéaire par rapport à la taille de la combinaison. --> O(n) où «n» est la taille de la combinaison
4. L’ajout d’une combinaision à la liste possibleSolutions --> cela prend un temps lineaire donc O(1)
5. Renvoie de la liste --> temps constant --> O(1)

L’étape qui a la complexité la plus élevée est la génération de toutes les combinaisons possibles par rapport aux autres donc --> O(2^n) où «n» est le nombre d’éléments dans la liste «array»

findMaxGain :

1. Parcours la liste de «solutions» : dépend du nombre de solution trouvées par l’algorithme précédent. Le nombre de solutions trouvées est «n» donc --> O(n)
2. Parcours des actions dans chaque solution donc cela dépend du nombre d’actions dans une solution. On suppose qu’il y a en moyenne «m» actions dans chaque solution --> O(m)

En combinant ces deux étapes, on obtient O(n\*m). Plus m est grand, plus cela augmente la complexité globale.

L’équation globale qui résume la complexité de ces trois fonctions est :

O(calculGain(n) + testSampleSizeSolutions(n) + findMaxGain(n x m))

**Optimized :**

calculGain :

Idem que dans brute force

testSampleSizeSolution :

Cette fonction renvoie un résultat dès qu’une solution satisfaisante est trouvée.

La complexité de cette fonction dépend fortement des valeurs dans la liste «array» et des tailles des combinaisons qui satisfont la condition (somme<=500). Si une solution satisfaisante est trouvée rapidement, l’algorithme est efficace mais si aucune solution n’est trouvée jusqu’à échantillon le plus petit i = 1, alors la complexité augmentera.

Dans le pire des cas, cette version révisée pourrait avoir une complexité exponentielle, car elle parcourt toutes le combinaisons possibles pour chaque taille jusqu’à 20. Par conséquent, sa complexité pourrait être approximée par O(2^n), où «n» dépend du nombre d’actions dans la liste array.

L’équation globale qui résume la complexité de ces deux fonctions est :

O(calculateGain(n) + textSampleSizeSolutions(2^n)