Devoir 1

# Approche

Pour résoudre ce problème, il sera nécessaire de tester un grand nombre de possibilités. L’approche à ce problème que nous verrons dans ce document est d’essayer de réduire le plus possible le nombre de tests nécessaires, en éliminant les combinaisons à l’avance, avant de les tester.

Le processus pour optimiser le nombre de tests, et d’ignorer les combinaisons invalides, est la suivante.

Tout d’abord, nous assumons que nous aurons besoin de tester toutes les combinaisons possibles de pièces. Nous allons donc vérifier le total de chaque combinaison :

Où i est l’indice de chaque pièce, ni est le nombre de fois que cette pièce se répète, et valeuri est la valeur de i.

Évidemment, l’ordre dans lequel les pièces sont placés dans la combinaison n’a pas d’importance, donc nous pouvons ainsi éliminer toutes les combinaisons équivalentes.

Nous allons donc approcher ce problème en commençant par évaluer toutes les combinaisons qui contiennent seulement qu’une pièce d’abord, et ensuite évaluer les combinaisons avec deux pièces, et trois pièces, et ainsi de suite.

Si nous prenons par exemple une situation avec deux pieces P1=1 et P2=2 :

Les combinaisons à 1 pièce sont {{1}, {2}}

Les combinaisons à 2 pièces sont {{1,1},{1,2},{2,1},{2,2}}, or, les combinaisons {1,2} et {2,1}

Les combinaisons à 3 pièces sont {{1,1,1},{1,1,2},{1,2,1},{1,2,2},{2,1,1},{2,1,2},{2,2,1},{2,2,2}}, or, les combinaisons {1,1,2},{1,2,1} et {2,1,1} sont identiques, et les combinaisons {1,2,2},{2,1,2} et {2,2,1} sont également identiques.

Pour éliminer toutes les combinaisons identiques, nous allons donc ajouter une règle. Toutes les combinaisons doivent être en ordre, c’est-à-dire qu’il est impossible d’avoir une combinaison avec une pièce plus petite qui suit une pièce plus grande (comme {2,1} par exemple).

Maintenant que nous n’avons que des combinaisons uniques, nous allons donc comparer le total obtenu de chaque combinaison, et nous le comparons avec la valeur désirée. C’est ici qu’on peut éliminer toutes les combinaisons qui dépassent la valeur désirée. L’algorithme fonctionne avec des « Niveaux » N qui représentent le nombre de pièces dans les combinaisons. Donc, les combinaisons de niveau 1 sont des combinaisons contenant seulement qu’une pièce. Le but de l’algorithme est de trouver la combinaison avec le plus petit nombre de pièces possible qui arrivent à un total désiré. Nous cherchons donc la combinaison avec le plus bas niveau possible, et vù que nous commençons au premier niveau, et que nous augmentons de niveau progressivement, la première combinaison que nous allons trouver sera automatiquement la plus optimale.

Nous évaluons donc le total de chacune des combinaisons N=1, et nous gardons toutes les combinaisons avec un total inférieur à la valeur désirée, pour ensuite prendre ces combinaisons et ajouter une pièce de chaque type. Par contre, nous avons vu précédemment que pour éviter les combinaisons identiques, nous ne devons jamais ajouter une pièce inférieure à la plus grosse pièce de la combinaison.

Chaque combinaison sera un objet contenant l’indice de chaque pièce de la combinaison, la somme de cette combinaison, ainsi que l’indice de la plus grande pièce de cette combinaison.

Par exemple, si nous avons une situation avec 6 pièces, P1=1, P2=5, P3=10, P4=25, P5=100 et P6=200, et une combinaison comme suit : {1,1,5,25}, cette combinaison sera représentée par un objet contenant:

* Un tableau contenant l’index de chaque pièce dans la combinaison : array = {1,1,2,4}
* Le total de la combinaison : sum = 32
* L’indice de la plus grande pièce contenue dans cette combinaison : maxIndex = 4

Notez que la raison pour laquelle nous incluons l’indice de la plus grande pièce contenue est simplement pour réduire le temps d’exécution au plus possible.

À chaque niveau, nous allons ajouter une pièce aux combinaisons qui avaient un total inférieur à la valeur désirée, mais nous n’allons jamais ajouter une pièce plus petite que celle indiquée par la variable « maxIndex ».

Lorsque le total équivaut à la valeur désirée, le programme se termine (sauf si nous désirons vérifier toutes les possibilités, ce que nous verrons plus tard)

Nous avons déjà réduit grandement le nombre de vérifications nécessaires, mais nous pouvons optimiser d’avantage cette solution.

Nous savons qu’il est inutile d’ajouter une pièce plus petite que la pièce d’indice maximale dans un objet Combinaison. Donc, pour savoir si une combinaison devrait passer au prochain niveau, au lieu de vérifier si le total de cette combinaison est inférieur au résultat désiré, nous allons vérifier si la somme du total de la combinaison avec la plus grande pièce de la combinaison est plus petit que la valeur désirée. Si non, nous savons que cette combinaison aurait été éliminée au prochain niveau de toute façon, donc nous pouvons l’éliminer immédiatement. À noter que cette vérification se fait après d’avoir vérifié si le total de la combinaison est égal au résultat désiré.

Évidemment, nous ne devons pas laisser le programme s’exécuter à l’infini, car au cas ou on aurait une solution impossible, nous aurions soit un problème de type « Out of memory » ou « Stack overflow ».

Or, si nous avons une combinaison avec un total supérieur à la valeur désirée, qui est composé uniquement de la plus petite pièce, nous savons par défaut qu’il n’y a aucune combinaison plus grande que ça qui peut résoudre le problème. C’est donc après d’avoir déterminé que le total d’une combinaison est supérieur à la valeur désirée que nous allons vérifier si le maxIndex de cette combinaison correspond à la plus petite pièce. Si c’est le cas, le programme termine l’exécution et indique qu’il est impossible de trouver une solution pour ce problème.

Ensuite, pour trouver le nombre de solutions possibles à un problème, nous pouvons tout simplement laisser le programme continuer après qu’il ait trouvé la solution initiale, et incrementer une variable à chaque fois qu’il en trouve une autre, jusqu’à ce qu’il atteint la combinaison finale, qui a un total supérieur à la valeur désirée et qui a un maxIndex égal à la plus petite pièce.

Cette méthode nous permet d’avoir le temps minimal d’exécution pour obtenir la première solution, et de pouvoir attendre d’avantage pour laisser le temps au programme de trouver les autres solutions.

Maintenant, nous devons trouver la complexité algorithmique du programme. Voici la partie principale de l’algorithme en question :



Ici, nous nous intéressons au « worst case scenario » : Dans le cas de cet algorithme, vù qu’il dépends beaucoup des combinaisons qui ont un total plus grand que la valeur désirée pour optimiser les calculs, le pire cas possible est un cas où cet avantage est réduit au néant par des valeurs de pièces très petites et une valeur désirée très grande.

Tout d’abord, nous pouvons facilement constater que la boucle extérieure va être exécutée n fois. (Où n est le nombre de niveaux nécessaire pour trouver la solution, ou encore le nombre minimal de pièces nécessaire pour arriver au montant désiré)

Parcontre, la boucle intérieure va évoluer différement :

Si nous analysons la progression du nombre de possibilités de former une combinaison à un niveau spécifié n (avec un nombre i de pièces) selon notre algorithme, nous obtenons ce qui suit :

Où n est le nombre d’éléments dans chaque combinaison, et i est le nombre de pièces.

Ceci représente le nombre de combinaison possible pour un niveau. La boucle intérieure de l’algorithme va parcourir chaque niveau jusqu’à n, et cycler selon l’équation ci-dessus.

Alors, cette boucle va s’éxecuter ainsi :

Ce problème est assez difficile à résoudre de manière mathématique. Parcontre, il est très facile à résoudre à l’aide de programmation. Ainsi, en implémentant cette simple fonction récursive, nous pouvons évaluer le nombre de cycle que cette fonction effectue pour un n final donné, et un i donné :



Donc, en analysant cette situation, on peut constater que… yeah cest compliqué as fuck