TD15 - Programmation dynamique

Exercice 1

Dans un premier temps on construit l'exercice dans sa version récursive classique, puis on détermine ce que l'on peut améliorer avec une programmation dynamique.

Partie A : Version récursive classique

Les coefficients du binôme de Newton $C_{k,n}$ sont définis pour $0 \leq k \leq n$. Ils se calculent par une formule de récurrence :

- $C_{k,n} = 1$ si $k = 0$ ou $k = n$,
- $C_{k,n} = C_{k-1,n-1} + C_{k,n-1}$ sinon.

Un coefficient s'obtient donc comme la somme de deux autres. Ce coefficient $C_{k,n}$ est habituellement noté $\binom{n}{k}$ et lu « k parmi n ». Ce sont aussi les coefficients qui apparaissent dans le développement de $(a+b)^n$. Par exemple :

$$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$$

et on a $C_{0,4} = 1$, $C_{1,4} = 4$, $C_{2,4} = 6$, $C_{3,4} = 4$, $C_{4,4} = 1$.

Question : Programmer une fonction récursive `binome(k,n)` qui renvoie $C_{k,n}$

- Les cas terminaux sont pour k = 0 et k = n.
- Pour le cas général, il faut faire deux appels récursifs : un appel pour $C_{k-1,n-1}$ et un autre pour $C_{k,n-1}$.

Question : Tester votre programme avec les exemples précédents

Partie B : Coefficients du binôme, version récursive dynamique

Question 1 : arbre

Construisez l'arbre des appels de la fonction `binome(3,6)` dans sa version récursive.

Question 2 : mémoïsation

A partir de cet arbre, déterminer ce qu'il est utile de mémoriser dans la version dynamique de cette fonction.

Question 3 : programmation

Programmer une fonction `binome_mem(k,n,mem)` La fonction doit réussir à mémoriser les calculs qui vous paraissent réutilisables dans le dictionnaire et les utiliser si nécessaire. On appelle la fonction avec un dictionnaire vide appelé `mem`. Utilisez l'exemple de la suite de Fibonacci pour la construction de vos fonctions.

Question 4 : comparaison

Modifier `binome(k,n)` et `binome_mem(k,n,mem)` afin que le mot `calcul` s'affiche à chaque fois qu'un appel récursif est lancé. Comparer les affichages dans le cas du calcul de $C_{3,6}$.

Partie C : Triangle de Pascal

Question 1 : Affichage avec `binome(k ,n)`

Utilisez la fonction de la partie A pour afficher le triangle de Pascal: la ligne numéro n est composée des coefficients $C_{k,n}$ pour $k = 0, 1, \ldots, n$ (la numérotation n des lignes commence avec n = 0).

Rappel: Voici comment afficher une chaîne de caractères à l'écran sans passer à la ligne suivante, après un espace : `print (chaine, end=" ")`. Pour aller à la ligne, il faut alors utiliser `print()`.

Question 2 : Affichage avec `binome_mem (k, n, mem)`

Faire de même mais en utilisant la version dynamique du calcul des coefficients binomiaux.

Question 3

Pour mesurer l'optimisation obtenue, nous allons mesurer les temps de calculs nécessaires à la réalisation des deux calculs précédents. Il faudra utiliser le module time (`from time import perf_counter`) et modifier votre programme de la sorte:

```
debut=perf_counter()
# remplacer ceci par l'appel de la fonction ou du code que l'on veut tester, retirer les affichages pour ne te
fin=perf_counter()
print("Temps de calcul pour afficher Pascal : ", fin-debut)
```

Mesurer le temps d'exécution des deux fonctions précédentes avec N=21, 22,23,24,25.Comparez les résultats. Que constatez-vous? Que se passe-t-il pour le temps d'exécution de ces affichages lorsque que l'on augmente N de 1 en 1. Que pouvez-vous en déduire?

Exercice 1 : le rendu de monnaie

On rappelle le contexte du problème : Etant donné un ensemble P (pour pièces) d'entiers, représentant la liste des pièces et billets disponibles, et une somme s, donner :

- Le nombre minimal de pièces/billets de P dont la somme vaut s ;
- Une liste minimale de pièces/billets de P dont la somme vaut s.

Partie A - Exemples et préparation

Question 1 - application

Dans le système P = Euro, donner les sorties pour $s = 13,35€$

Question 2 - le Glouton

Rappeler le principe de l'algorithme glouton. Écrire cet algorithme sur papier.

Question 3 - Optimalité ?

L'algorithme glouton est-il optimal pour le système impérial anglais ? Si oui, prouvez-le. Si non proposez un contre exemple.

Indications : (sources https://omnilogie.fr/ et wikipedia) :

La livre était divisée en 20 shillings et un shilling valait 12 pence (au pluriel, on dit "penny", mais au sigulier, c'est "pence"). Jusque là, c'est bizarre mais ça reste compréhensible. C'est ensuite que ça se complique un peu :

- Il y avait d'abord les sous-unités du penny : le farthing (1/4 de penny) et le demi-penny.
  - Puis les sous-unités du shilling : le penny évidemment (1/12 shilling),
  - le trois pence (1/4 de shilling),
  - le quatre pence ou groat (1/3 de shilling) et
  - le six pence (1/2 shilling).
- Enfin, il y avait les sous-unités de la livre :
  - le shilling (1/20 de livre),
  - le florin (1/10 de livre ou 2 shillings),
  - la demi-couronne (1/8 de livre ou 2 shillings et demi),
  - la couronne (1/4 de livre ou 5 shillings),
  - le demi-souverain (1/2 livre) et
  - le souverain or qui valait une livre.
- La guinée, utilisée de 1663 à 1818, est un cas particulier. Elle valait 21 shillings, soit 1 livre et 1 shilling. Elle était en or, un métal abondamment extrait en Guinée en Afrique de l'Ouest, pays auquel la monnaie doit son nom.
- Le mark valait 2/3 livres (soit 160 pennies ou 13 shillings et 4 pennies)

Remarque : si vous répondez à cette question en moins de 5 minutes, bravo ! Et sinon, n'y passez pas plus de 5 minutes, sautez la question.

La réponse est non. Pour ne pas faire d'anglophobie, précisons que tous les systèmes monétaires d'Europe avant la révolution française, puis Napoléon, étaient aussi complexes. C'est un héritage carolingien : une livre vaut 20 sous et 12 deniers.

Question 4 - Un exemple plus facile

Montrer que l'algorithme glouton ne donne pas la solution optimale avec P = [1, 4, 6] et s = 8.

Question 5 - Algorithme naïf

Principe :

- Si la somme à rendre vaut 0, alors le nombre de pièces nécessaire est 0. L'algorithme renvoie 0.
- Sinon, pour toutes les pièces p possibles, on soustrait à la somme s la valeur de la pièce p, et on cherche le nombre de pièces à rendre pour la somme .
- On a donc la formule récursive :
  - $ nb_pièces (s) = $
    - 0 si $s = 0$,
    - $1 + min(nb_{pieces}(s - p))$ pour $p \leq s$ sinon.

L'algorithme récursif correspondant est :

```
Rendu naïf (Pièces , s):
    Si s = 0 :
        renvoyer 0
    Sinon :
        nb_pieces = s # au pire on renvoie s pièces de 1 centime
    Pour chaque pièce p de Pièces :
        Si  p<= s:
            nb_pieces_bis = Rendu naïf (Pièces , s-p) + 1
            nb_pieces = min(nb_pieces , nb_pieces_bis)
    Renvoyer nb_pieces
```

Faire tourner à la main cet algorithme avec Pièces = [1, 4, 6] et s = 8. Pour cela, on construira l'arbre des appels récursifs. Que constatez-vous ? Quelle semble être la complexité de l'algorithme ? On pourra imaginer ce qui se passe si on prend s=9, s=10 … dans l'exemple.

Partie B - Programmation dynamique

On va calculer le nombre de pièces nécessaires pour rendre la somme s non pas par une approche descendante, mais par une approche ascendante (bottom-up). On va calculer successivement le nombre de pièces à rendre pour somme=0, somme=1 … jusqu'à somme=s.

Question 1 - Stockage des résultats / Initialisation

On crée un tableau `nombre_pieces` pour stocker ces résultats.

- Quelle est la taille du tableau ?
- Sachant que l'on va chercher à minimiser les éléments de ce tableau (le nombre de pièces à rendre pour chaque somme), à quelle valeur peut-on initialiser les éléments ?

Question 2 - Une boucle exemple

On boucle ensuite en remplissant ce tableau dans l'ordre croissant des indices, en réutilisant la formule : nb_{pieces} (somme) = 1+ min(nb_{pieces}(somme-p).

Faire tourner cette méthode sur l'exemple `Pièces = {1, 4, 6}` et `s = 8`, en modifiant le tableau `nombre_pièces = [8, 8, 8, 8, 8, 8, 8, 8, 8]` au fur et à mesure. On précisera toutes les étapes.

Question 3 - Programmation

Programmer la fonction renduProgDyn(P, s) en Python. La tester.

Question 4 - Complexité

Donner la complexité de l'algorithme de rendu de monnaie en programmation dynamique, en fonction de la longueur du tableau Pièces et de s.

Question 5 - Reconstruction de la solution

On utilise le tableau nombre_pièces = [0, 1, 2, 3, 1, 2, 1, 2, 2] obtenu à la question 2. On part de la dernière valeur 2 d'indice 8. Pour chaque pièce p le permettant, on calcule l'indice `8- p. Si nombre_pièces[8-p]=nombre_pièces[8]-1`, alors on a trouvé une valeur possible précédente (il peut y avoir plusieurs possibilités). Ici :

- `nombre_pièces[8]=2`
- `nombre_pièces[8-1] = nombre_pièces[7] =2` ne convient pas
- `nombre_pièces[8-4] = nombre_pièces[4] =1`convient • `nombre_pièces[8-6] = nombre_pièces[2] =2` ne convient pas Donc la première pièce rendue a pour valeur 4.

Appliquer cette méthode pour prouver que la deuxième pièce rendue est aussi 4.

Question 6 - Programmation de la reconstruction

Programmer la fonction `python enduReconstruction(P, s, nombre_pieces)` en Python. Quelle est la complexité de cette fonction ?