Chapitre 3

PROGRAMMATION DYNAMIQUE

Table des matières

3 PROGRAMMATION DYNAMIQUE **1**

I PROBLÈME POSÉ PAR LA PROGRAMMATION RÉCURSIVE 2

II SOLUTION PROPOSÉE PAR LA PROGRAMMATION DYNAMIQUE 3

III MÉMOÏSATION 4

IV UN POINT SUR : ALGORITHMES GLOUTONS/RÉCURSIVITÉ/ PROGRAMMATION DYNAMIQUE/MÉMOÏSATION 5

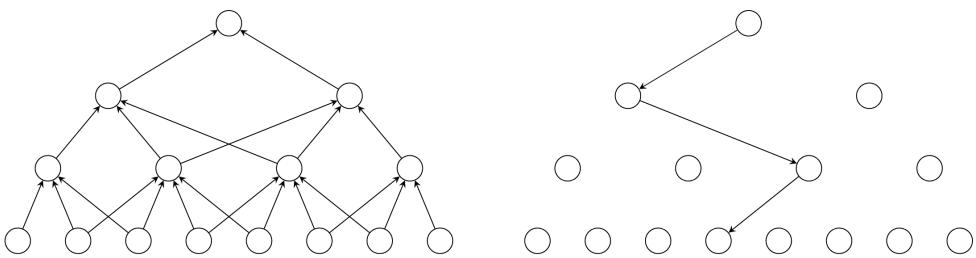
V EXERCICES 5

Ce chapitre fait suite à celui sur les méthodes gloutonnes. Il s'agit de programmation dynamique qui est une technique où l'on cherche à minimiser (ou maximiser) une certaine fonction. Néanmoins on va chercher à trouver une solution réellement optimale (*contrairement à l'approche gloutonne*).

Nous avons vu en première année l'algorithme de Dijkstra qui permet de déterminer les plus courts chemins entre deux sommets de graphe et nous avons vu que cet algorithme glouton était optimal.

Nous avons également vu le problème de rendu de monnaie, qui consiste à rendre une certaine somme d'argent en minimisant le nombre de pièces rendues, et que la solution proposée n'était pas toujours optimale.

La programmation dynamique, au prix d'une complexité en général plus élevée qu'une approche gloutonne, permet de résoudre une plus grande variété de problèmes d'optimisation.



programmation dynamique *programmation gloutonne*

I PROBLÈME POSÉ PAR LA PROGRAMMATION RÉCURSIVE

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$.

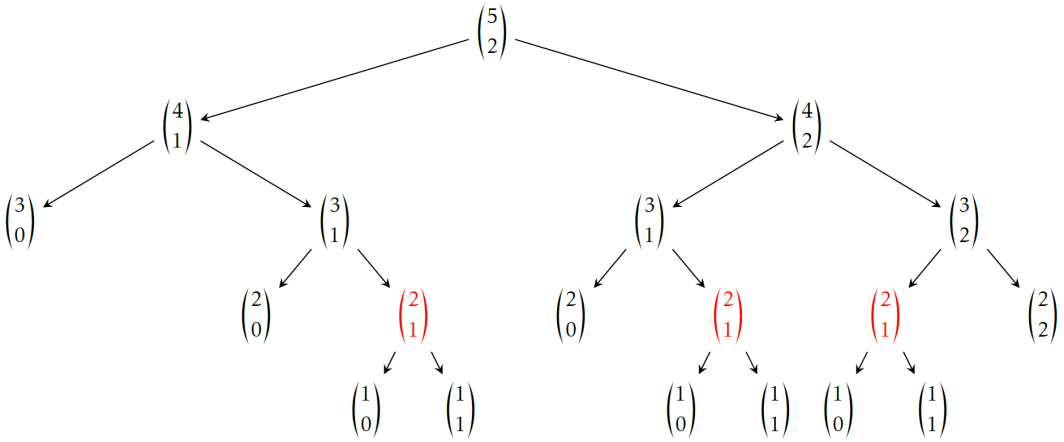
Une solution est d'utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

```
def binom(n,p):
  if p == 0 or n == p:
    return 1
  return binom(n-1,p-1) + binom(n-1,p)
```

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p : par exemple, il faut environ 80 secondes à mon ordinateur pour calculer $\binom{30}{15}$.

La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate que des appels récursifs sont identiques et donc superflus. On appelle ceci le **chevauchement de sous-problèmes.**

C'est le problème de la programmation récursive et de son approche avec **calcul de haut en bas**.



Par exemple, le calcul de $\binom{5}{2}$ fait appel trois fois au calcul de $\binom{2}{1}$.

L'expérience montre que le calcul de $\binom{30}{15}$ fait appel 40 116 600 fois (!) au calcul de $\binom{2}{1}$.

- **Complexité temporelle**

Pour évaluer la complexité de cette fonction, on note $\mathrm{C}(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

$$\mathrm{C}(n,0) = \mathrm{C}(n,n) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad \mathrm{C}(n,p) = \mathrm{C}(n-1,p-1) + \mathrm{C}(n-1,p) + 1$$

On démontre alors par récurrence sur $n \in \mathbb{N}$ que pour tout $p \in [\![0,n]\!], C(n,p) = \binom{n}{p} - 1$.

Or la formule de Stirling permet d'établir que $\binom{2n}{n} \sim \dfrac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ est donc de complexité exponentielle.

- **Où est le problème ?**

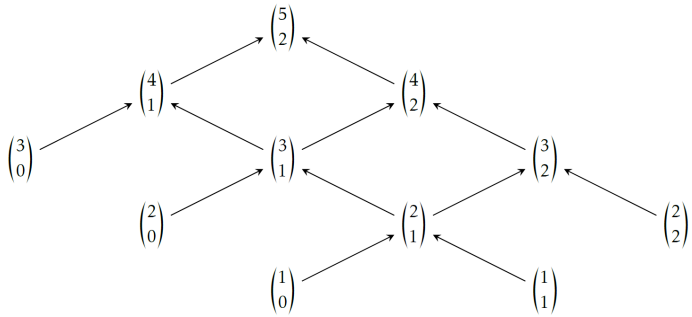
Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$, **sous-problèmes qui sont en interaction**.

Par exemple, on constate sur la figure précédente que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$.

Ainsi, la présence de sous-problèmes en interaction peut faire croitre très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

II SOLUTION PROPOSÉE PAR LA PROGRAMMATION DYNAMIQUE

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.
C'est le **calcul de bas en haut.**
Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma ci-contre :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ ou matrice (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits.

Il faut faire attention à bien respecter la relation de dépendance (flèches) pour remplir les cases de ce tableau : la case recevant la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases recevant $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

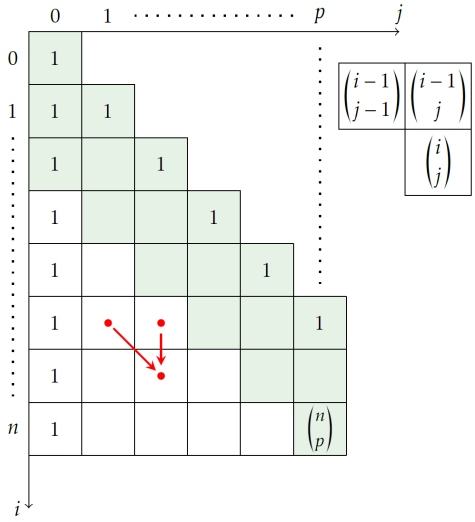


Schéma de dépendance du calcul de $\binom{n}{p}$

```python
def binom(n,p):
    t = [[0 for j in range(p+1)] for i in range(n+1)]
    for i in range(0,n+1):
        t[i][0] = 1
    for i in range(1,p+1):
        t[i][i] = 1
    for i in range(2,n+1):
        for j in range(1,min(p,i)+1):
            t[i][j] = t[i-1][j-1] + t[i-1][j]
    return t[n][p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficient que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $\mathcal{O}(np)$.

Remarque.
Notons que cette solution n'est pas encore optimale : il est facile de constater sur le schéma de dépendance que l'algorithme ci-dessus remplit des cases inutiles pour le calcul de $\binom{n}{p}$: seules celles qui sont colorées sont nécessaires.

On peut d'ailleurs observer qu'on peut se contenter d'utiliser un tableau unidimensionnel de $p+1$ cases contenant les valeurs $\binom{i+j}{i}$ pour $j \in [\![0,p]\!]$, partant de $i=0$ puis faisant varier i par pas de 1 jusqu'à $n-p$:

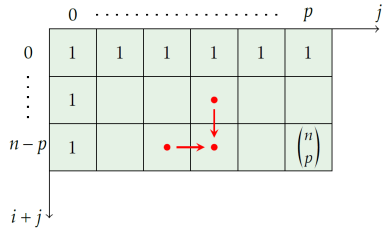


Schéma de dépendance du calcul de $\binom{i+j}{j}$

```python
def binom(n,p):
    t = [1 for j in range(p+1)]
    for i in range(1,n-p+1):
        for j in range(1,p+1):
            t[j] = t[j] + t[j-1]
    return t[p]
```

La complexité temporelle est maintenant un $\mathcal{O}(p(n-p))$ et la complexité spatiale $\mathcal{O}(p)$.

III MÉMOÏSATION

Un inconvénient de la programmation dynamique réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.

La solution existe, elle porte le nom de **mémoïsation**. Elle consiste à associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire (recherche en $\mathcal{O}(1)$ donc !) si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.

Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
1 binom_dict = {}
2
3 def binom(n,p):
4   if (n, p) not in binom_dict:
5     if p == 0 or n == p:
6       binom_dict[(n,p)] = 1
7     else:
8       binom_dict[(n,p)] = binom(n-1,p-1) + binom(n-1,p)
9   return binom_dict[(n,p)]
```

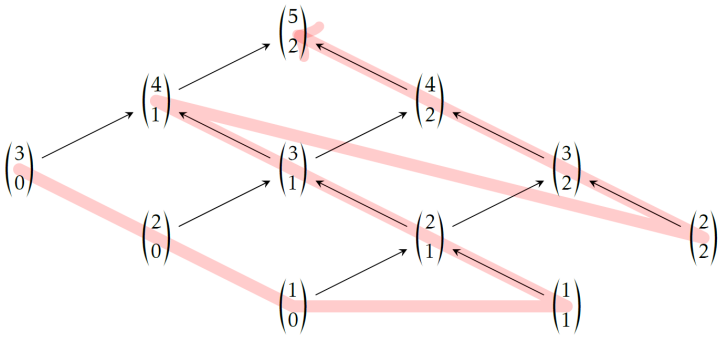
On peut observer que le programme récursif se retrouve presque mot pour mot lignes 5 à 8.

Calculons $\binom{5}{2}$ avec cette fonction, puis observons le contenu du dictionnaire :

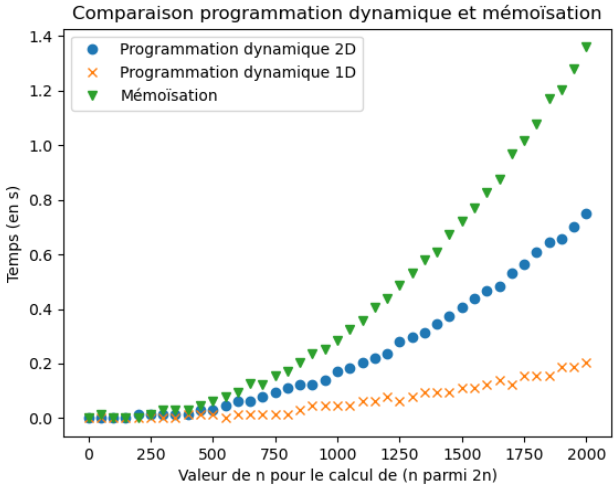
```
In [1]: binom(5,2)
Out[1]: 10
In [2]: binom_dict
Out[2]: {(3,0):1,(2,0):1,(1,0):1,(1,1):1,(2,1):2,(3,1):3,(4,1):4,(2,2):1,(3,2):3,
(4,2):6,(5,2):10}
```

On peut constater qu'on y retrouve les 10 valeurs nécessaires pour réaliser ce calcul.

Il est important que le **dictionnaire** soit **partagé** par toutes les instances (appels récursifs) de la fonction, *il ne doit donc pas être une variable locale à la fonction qui fait les appels récursifs.*



Ordre d'entrée dans le dictionnaire



Remarque.

• On constate que même avec la mémoïsation, la version récursive du calcul de coefficient binomial reste un peu moins performante que les versions itératives de programmation dynamique.

• Ne pas oublier que le nombre maximal d'appel récursif reste limité en Python (il peut être néanmoins modifié).

IV UN POINT SUR : ALGORITHMES GLOUTONS/RÉCURSIVITÉ/ PROGRAMMATION DYNAMIQUE/MÉMOÏSATION

• On peut faire de la récursivité efficace sans faire de la programmation dynamique (par exemple le tri fusion), on peut aussi faire de la programmation dynamique sans récursivité (voir le II).

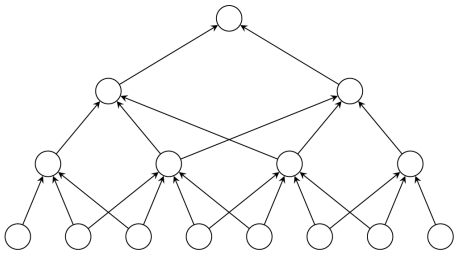
• La mémoïsation est utile lorsque lors d'un algorithme récursif, la fonction va s'appeler elle-même plusieurs fois avec les mêmes paramètres. Alors, les calculs vont être effectués plusieurs fois à l'identique par Python. C'est comme si vous calculiez 58786×874561 trente fois d'affilée et qu'à chaque fois vous jetiez le résultat et que vous deviez recommencer. À la place, il suffit de mémoriser. Rappelez-vous qu'il ne s'agit pas seulement de réduire la durée d'un calcul, il s'agit parfois de passer d'un calcul qui prendrait quasiment une éternité à un calcul qui va être fait en quelques secondes. La contrepartie c'est que l'on va perdre un peu en stockage dans la mémoire.

• À noter qu'on peut faire de la mémoïsation sans forcément faire de la programmation dynamique et on peut faire de la programmation dynamique sans nécessairement faire de la mémoïsation.

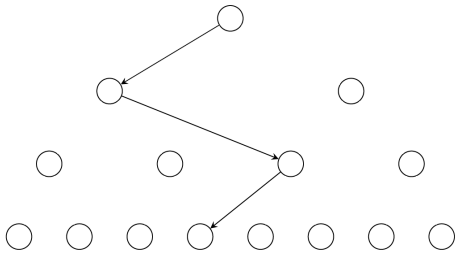
• Rappelons ce qu'est algorithme glouton : c'est un algorithme qui se résout par une succession d'étapes où à chaque étape :

— On a fait un choix qui a réduit au maximum la taille du problème.

— Et ce choix est définitif.



programmation dynamique



programmation gloutonne

Ainsi, le rendu de pièces de monnaies vu en PCSI est un algorithme glouton. À chaque étape, on cherche la plus grande pièce pour réduire au maximum le montant qui reste à rembourser. Et ce choix est définitif, on ne revient donc jamais dessus, quitte à passer à côté d'une solution plus optimale (dans ce cas, une solution qui prenne moins de pièces).

On pourrait penser qu'un algorithme glouton est un algorithme de programmation dynamique mais ce n'est pas le cas. En effet, à chaque étape d'un algorithme glouton, on optimise l'étape, mais on ne change pas les étapes précédentes. Alors qu'en programmation dynamique, on résout de façon plus globale. Cependant la programmation dynamique et un algorithme glouton ont en commun d'avoir un programme qui se résout étapes par étapes.

V EXERCICES

Exercice 1 (**Fonctions récursives**)**.**

1. Écrivez une fonction récursive `sommeCarres` qui prend en argument un entier positif n et qui renvoie $1^2 + 2^2 + \cdots + n^2$. Par exemple $1^2 + 2^2 + \cdots + 4^2 = 30$.
2. Écrivez une fonction récursive `compter` qui prend en argument un entier n et affiche les entiers compris entre 1 et n. Par exemple, l'exécution de compter(4) doit donner :

```
1
2
3
4
```

3. Écrivez une fonction <u>récursive</u> `rebours` qui prend en argument un entier n et affiche en ordre décroissant les entiers compris entre n et 0. Par exemple, l'exécution de rebours(4) doit donner :

```
4
3
2
1
0
```

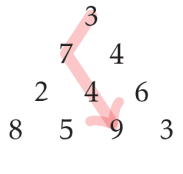
Exercice 2 (**La suite de Fibonacci avec ou sans mémoïsation**).

Soit $(F_n)_n$ la suite de Fibonacci : $F_0 = F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

1. Écrire une fonction récursive classique `F(n)` qui, étant donné un entier n, renvoie F_n.
   Le calcul de $F(50)$ est-il possible ? Pourquoi ?
   Esquisser l'arbre d'appels récursifs.
2. Écrire une version de la fonction précédente avec mémoïsation notée `Fb(n)`.
   Le calcul de `Fb(50)` est-il maintenant possible ?
   On vérifiera que `Fb(100)%1000` *vaut 101.*
   Esquisser l'arbre d'appels récursifs de la fonction `Fb`.

Exercice 3 (**Chemin de total maximum**).

En partant du sommet du triangle ci-dessous et en se déplaçant vers les nombres adjacents de la ligne inférieure, le total maximum que l'on peut obtenir pour relier le sommet à la base est égal à 23 :



$3 + 7 + 4 + 9 = 23$

```
T = [[3],
     [7, 4],
     [2, 4, 6],
     [8, 5, 9, 3]]
```

```
T = [[   ],
     [   ,    ],
     [   ,   ,   ],
     [   ,   ,   ,   ]]
```

On souhaite rédiger une fonction calculant le total maximum d'un chemin reliant le sommet à la base d'un tel triangle de hauteur n.

Pour cela, on pourra considérer que les valeurs de ce triangle sont stockées dans un tableau bi-dimensionnel `T` où la i-ième ligne contient i éléments pour $i \in [\![1, n]\!]$. Autrement dit, `T[i][j]` contient la $(j+1)$ème valeur de la $(i+1)$ème ligne.

1. L'idée est de modifier les valeurs du tableau `T` au fur et à mesure, en commençant par les premiers lignes, de sorte ce que `T[i][j]` soit remplacé par le total maximal des chemins possibles arrivant à l'indice (i, j).

   On s'autorise à utiliser la fonction `max` de Python qui renvoie l'élément maximum d'une liste de flottants ou d'entiers, ou même de plusieurs nombres entrés en arguments.

   (a) Compléter, à la main, le tableau `T` ci-dessus à droite et vérifier que le total maximum est bien de 23 parmi tous les chemins possibles.
   (b) Imaginons qu'on ait mis à jour `T` jusqu'à la ligne d'indice `i-1` ($i < n$). Par quelle valeur devra-t-on remplacer `T[i][0]` ? et `T[i][i]` ?
   (c) Pour les autres valeurs de la ligne d'indice `i` ($i < n$), comment remplacer la valeur de `T[i][j]` ?

2. A l'aide des remarques faites précédemment, écrire une telle fonction `Chemin_Max(T:list):->int`, qui renvoie le total maximal parmi tous les chemins possibles. Il n'est pas important que le tableau `T` soit modifié (mais il faudra le réinitialiser avant chaque appel à la fonction).

Exercice 4 (**Plus longue sous-séquence commune**)**.**

Étant donnés deux mots, on cherche la plus longue sous-séquence commune. C'est-à-dire une chaîne de caractères dont les lettres sont dans les deux mots à la fois tout en conservant l'ordre.

Et on cherche une chaîne la plus longue possible. Par exemple, c**i**rc**o**ns**tance** et **i**mp**o**r**tance**, la plus grande sous-séquence commune a pour longueur 7 avec **iotance** ou **irtance**.

Mais on pas **irotance** car l'ordre du « r » et du « o » ne serait pas respecté. On veut que la séquence soit issue des deux mots, en conservant l'ordre des lettres.

On peut en donner une définition plus formelle : si `mot` et `mot2` sont deux mots, on appelle séquence commune une chaine de caractère `c` de longueur p, telle que :

$$\forall i \in [\![0, p-1]\!],\ \texttt{c[i]=mot[}a_i\texttt{]=mot2[}b_i\texttt{]}$$

où $0 \leqslant a_0 < a_1 < \ldots < a_{p-1} <$ `len(mot)` et $0 \leqslant b_0 < b_1 < \ldots < b_{p-1} <$ `len(mot2)`, et parmi toutes les chaînes `c` on souhaite avoir une dont la longueur est la plus grande.

On autorise également dans cet exercice l'emploi des fonctions `max` et `min`.

1. On souhaite d'abord construire une fonction `longueur(s,t)` qui a deux chaînes de caractères `s` et `t` va compter la longueur de la plus grande sous-sequence commune.
   (a) Si `s` (ou `t`) est la chaîne vide, que vaut `longueur(s,t)` ?
   (b) Notons `n=len(s)` et `p=len(t)`, on est dans le cas où `s[n-1]=t[p-1]`.
   Comparer alors `longueur(s,t)` à `longueur(s[0:n-1],t[0:p-1])`.
   (c) Ici, on est dans le cas `s[n-1]` $\neq$ `t[p-1]`.
   Donner une relation entre `longueur(s,t)`, `longueur(s[0:n-1],t)` et `longueur(s,t[0:p-1])`.
   (d) Coder alors la fonction `longueur` de façon récursive avec mémoïsation.
   Ne pas oublier de réinitialiser le dictionnaire de mémoïsation avant chaque appel !
2. Programmer alors une fonction `PLSC(s,t)` qui trouve une sous-sequence commune de longueur maximal. On distinguera les cas comme pour la fonction `longueur`.
3. Télécharger le fichier `ListeMots.txt` qui est sur le Cahier de Prépa de la PSI du lycée de l'Essouriau pour constituer la liste des 1300 mots les plus utilisés de la langue française, trouver deux mots dont la plus longue sous-sequence commune est la plus longue.

   On doit trouver « appartement » et « parfaitement ».

Exercice 5 (**Way Back to the ~~Futur~~ Dynamic Programmation I**)**.**

Reprenez l'exercice 3 (Chemin de total maximum) et écrivez un script qui permet de déterminer par quel chemin du triangle on est passé pour obtenir le total maximum.

Exercice 6 (**Back to the ~~Futur~~ Dynamic Programmation II**)**.**

Reprenez l'exercice 3 mais de façon récursive avec mémoïsation (pour écrire une fonction calculant la valeur de `T[i][j]` dans un premier temps par exemple).