Programmation dynamique et mémoïzation

C. Charignon

Table des matières

1 Mémoïzation

1.1 Principe de la mémoïzation

Le principe de la mémoïzation est relativement simple : dans une situation ou un algorithme naïf conduit à recalculer plusieurs fois une même valeur, on va enregistrer au fur et à mesure toutes les valeurs déjà calculées. Ceci demande plus de mémoire mais permet un gain de temps.

N.B. Attention : dans ce genre de technique on va mélanger des concepts de programmation impérative (utilisation d'un tableau ou autre structure mutable pour enregistrer les valeurs déjà calculées) et récursive (la structure de programme de base est souvent récursive.)

En général, voici comment on procède pour mémoïzer une fonction :

- Créer un objet qui servira à retenir chaque résultat calculé. Je nommerai souvent `cache` cet objet.
  NB : le cache devra être créé hors de la fonction récursive ! Sans quoi il serait réinitialisé à chaque appel récursif.
- *au début* de chaque appel récursif, on regarde si la valeur a déjà été calculée. Si oui, on renvoie immédiatement le résultat. Sinon, on procède aux mêmes calculs que dans la version naïve.
- *à la fin* du calcul, on n'oublie pas d'enregistrer le résultat obtenu dans le tableau avant de le renvoyer.

En pratique, il y a trois opérations à réaliser :

1. Transformer la fonction initiale en une fonction auxiliaire, et créer une fonction principale s'occupant de créer le tableau puis d'appeler la fonction auxiliaire.
2. Au début de la fonction auxiliaire, tester si la valeur a déjà été calculée, et si oui la renvoyer directement. Sinon effectuer le calcul de la fonction naïve.
3. En sortie de la fonction auxiliaire, enregistrer la valeur calculée.

Remarque : Dans certains langages, il est possible d'automatiser la mémoïzation. C'est-à-dire d'écrire une fonction `mémoïzator` qui prend en entrée une fonction quelconque et renvoie la fonction mémoïzée. En Caml, c'est peu pratique à cause du typage statique.

1.2 Premier exemple : suite de Fibonacci

On rappelle que l'algorithme naïf :

```
4  def fiboNaïf(n):
5      if n==0 : return 0
6      elif n==1 : return 1
7      else: return fiboNaïf(n-1) + fiboNaïf(n-2)
```

a une complexité exponentielle, car il recalcule un grand nombre de fois les mêmes valeurs.

On peut le mémoïzer en utilisant un tableau pour enregistrer les valeurs déjà calculées :

```
11 def fiboMémo(n):
12     cache = [-1 for i in range(n+1)] #Attention au +1
13
14     def aux(i):
15         if cache[i] != -1:
16             return cache[i]
17         elif i==0 :
18             cache[i]=0
19             return 0
20         elif i==1 :
21             cache[i]=1
22             return 1
23         else:
24             res = aux(i-1) + aux(i-2)
25             cache[i]=res
26             return res
27
28     return aux(n)
```

On peut alléger un peu le code : les cas de base peuvent être traités en remplissant à l'avance le cache ;

```
def fiboMémo2(n):
    cache = [-1 for i in range(n+1)] #Attention au +1

    # Cas de base:
    cache[0]=0
    cache[1]=1

    def aux(i):
        if cache[i] != -1:
            return cache[i]
        else:
            res = aux(i-1) + aux(i-2)
            cache[i]=res
            return res

    return aux(n)
```

1.3 Version impérative

Sur le cas particulier de la suite de Fibonacci, comme sur beaucoup d'autres, on peut simplifier la structure de l'algorithme. En effet, on constate que pour calculer un terme F_n, on a uniquement besoin des termes *précédents*. Par conséquent, il est possible de remplir le tableau de gauche à droite, par une simple boucle pour !

```
def fiboDyna(n):
    cache = [-1 for i in range(n+1)] #Attention au +1

    # Cas de base:
    cache[0]=0
    cache[1]=1

    # On remplit en commençant par les petites valeurs
    for i in range(2,n+1): # attention aux bornes...
        cache[i] = cache[i-1] + cache[i-2]

    return cache[n]
```

On obtient un algorithme purement impératif, plus simple. Il y a deux bémols :

- ceci n'est possible que dans les situations pas trop compliquées où on peut deviner l'ordre dans lequel faire les calculs ;
- ceci nécessite un peu de réflexion, et donc comporte plus de risques d'erreur. Au contraire, la méthode générale de mémoïzation fonctionne toujours de la même manière.
- On remplit systématiquement tout le tableau alors que dans certains cas certaines cases peuvent être inutile (voire les exemple à suivre).

Le type de programmation utilisé pour cette version impérative s'appelle la *programmation dynamique*.

Remarque : Vocabulaire : la méthode impérative où on remplit les cases du tableau les unes après les autres s'appelle la méthode « bottom-up » car on commence en général à remplir le cache pour les petites valeurs de l'argument. Quand on colle à la fonction récursive initiale, c'est « up-bottom ». En effet, l'appel initial est pour la valeur finale à calculer, en général pour la plus grande valeur des arguments, et les appels récursifs se font sur des arguments de plus en plus petits.

Pour finir sur cet exemple, on pour améliorer la complexité mémoire. En effet pour calculer un terme de F on a uniquement besoin des deux termes précédents. Ce qui fait qu'il est inutile de garder les autres en mémoire. Au final en guise de cache deux variables entières suffisent.

```
def fiboDynaOpti(n):
    fi=1
    fi_moins_un=0
    for i in range(2,n+1):
```

```
        sauv=fi_moins_un
        fi_moins_un = fi
        fi = sauv+fi
        # ici, fi==f_i
    return fi
```

1.4 Deuxième exemple : coefficients binomiaux

Appliquons les méthodes de mémoïzation puis de programmation dynamique à l'exemple du calcul de coefficient binomiaux par la formule de Pascal.

On utilisera comme cache un tableau à deux dimensions.

Voici une version naïve inefficace :

```
def cbNaïf(p,n):
    if p==0:
        return 1
    elif p>n:
        return 0
    else:
        return cbNaïf(p-1,n-1) + cbNaïf(p,n-1)
```

On peut la mémoïzer ainsi :

```
# rema : on peut préférer utiliser numpy pour pouvoir créer des matrice plus facilement.
def cbMémo(p,n):
    cache = [[ -1 for k in range(p+1)] for l in range(n+1)] # cache[l][k] contiendra k parmis
    ↪ l. Convention perturbante du triangle de Pascal.

    # cas de base
    for l in range(n+1):
        cache[l][0] = 1
        # on pourrait optimiser ici...
        for k in range(l+1,p+1):
            cache[l][k]=0

    #la fonction principale
    def aux(k,l):
        if cache[l][k] != -1:
            return cache[l][k]
        else:
            res = cbNaïf(k-1,l-1) + cbNaïf(k,l-1)
            cache[l][k] =  res
            return res

    #ne pas oublier de lancer la fonction aux !
    return aux(p,n)
```

Et la « dynamiser » ainsi :

```
def cbDyna(p,n):
    cache = [[ 0 for k in range(p+1)] for l in range(n+1)] # Plus besoin d'avoir une valeur sp
    ↪ éciale pour savoir ce qu'on a déjà rempli. Autant mettre des 0 comme ça la partie haut-
    ↪ droite du rectangle de Pascal est déjà prête.

    for l in range(n+1):
        cache[l][0] = 1
        for k in range(1,min(l+1,p+1)): # Attention aux bornes
            cache[l][k] = cache[l-1][k] + cache[l-1][k-1]

    return cache[n][p]
```

Remarque : Démonstration : pour démontrer la correction de cette algorithme, on peut utiliser le prédicat de récurrence suivant :

« $\forall (i,j) \in [\![0,n]\!] \times [\![0,p]\!]$, l'appel à `aux i j` renvoie $\binom{j}{i}$, et en outre, après cet appel on a pour tout $(k,l) \in [\![0,n[\![\times[\![0,p[\![$, `cache.(k).(l)` contient -1 ou $\binom{k}{l}$. »

Le fait que `aux` soit à la fois une fonction (elle renvoie le résultat) et une procédure (elle modifie le tableau `cache` en effet de bord) nous oblige à mettre ces deux volets dans le prédicat de récurrence.

Remarque : Dans cet exemple, la méthode dynamique conduit à calculer des valeurs inutiles de $\binom{j}{i}$.

Par ailleurs comme pour Fibonacci, on remarque qu'on peut améliorer la complexité spatiale car le calcul d'une ligne du triangle de Pascal ne nécessite que les coefficients de la ligne précédente. Il est donc inutile de garder les autres en mémoire.

Une petite astuce rend le code très simple : utiliser pour `cache` un tableau de format $(2, n+1)$, en faisant en sorte que `cache.(0)` contienne des lignes d'indice pair du triangle de Pascal et `cache.(1)` des lignes d'indice impair.

```
136 def cbDyna2(p,n):
137     cache = [[ 0 for k in range(p+1)] for l in range(2)]
138     # cache[0] contiendra les ligne d'indice pair
139     # cache[1] contiendra les lignes d'indice impair
140 
141     for l in range(n+1):
142         cache[l%2][0] = 1
143         for k in range(1,min(l+1,p+1)): # Attention aux bornes
144             cache[l%2][k] = cache[(l-1)%2][k] + cache[(l-1)%2][k-1]
145 
146     return cache[n%2][p]
```

Remarque : Si l'on souhaite calculer un seul coefficient binomial, il vaut cependant mieux prendre la formule suivante (pour moi c'est la définition des coefficients binomiaux) :

$$\forall (n,p) \in \mathbb{N}^2, \ \binom{p}{n} = \frac{\prod_{k=0}^{p-1}(n-k)}{p!}$$

mais surtout pas la formule à partir des factorielles :

$$\forall (n,p) \in \mathbb{N}^2, \ \binom{p}{n} = \begin{vmatrix} 0 & \text{p} > n \\ \frac{n!}{(n-p)!p!} & \text{sinon} \end{vmatrix} .$$

Pourquoi ?

2 Premier exemple de programmation dynamique

2.1 Présentation du problème

On cherche à mesurer la distance entre deux mots (utilisé dans les correcteurs orthographiques, mais aussi par exemple sur des textes entiers pour une recherche de plagiat). On fixe un alphabet Σ. L'ensemble des mots formés à partir des lettres de Σ est noté Σ^*. Le mot vide (contenant 0 lettre) est noté ϵ. Pour tout $u \in \Sigma^*$, son nombre de lettres sera noté $|u|$.

On définit trois opérations élémentaires :

- la substitution : remplacer une lettre par une autre ;
- l'insertion : insérer une nouvelle lettre ;
- la suppression : supprimer une lettre.

La distance d'édition entre deux mots u et v est le nombre minimal de telles opérations élémentaires permettant de transformer u en v. On la notera $d(u,v)$.

Proposition 2.1. *La fonction d ainsi définie est une distance sur Σ^*, ce qui signifie :*

- $\forall (u,v) \in (\Sigma^*)^2$, $d(u,v) = d(v,u)$ *(symétrie) ;*
- $\forall (u,v) \in (\Sigma^*)^2$, $d(u,v) = 0 \Leftrightarrow u = v$ *(caractère séparé) ;*
- $\forall (u,v,w) \in (\Sigma^*)^3$, $d(u,w) \leqslant d(u,v) + d(v,w)$ *(inégalité triangulaire).*

Remarque : On a immédiatement $d(u,v) \leqslant |u| + |v|$: on peut toujours supprimer toutes les lettres de u (donc $|u|$ suppressions) puis insérer toutes les lettres de v (soit $|v|$ insertions).

Et même plus précis, en supposant que u est le plus petit des deux mots, $d(u,v) \leqslant |u| + |v| - |u| = |v|$: transformer u en le début de v par $|u|$ substitutions, puis insérer les $|v| - |u|$ lettres de v manquantes.

On peut démontrer que :

- Pour tout mot u, $d(u,\epsilon) = |u|$: il faut faire $|u|$ suppressions pour passer de u à ϵ.
- Pour tout mot u, $d(\epsilon,u) = |u|$: il faut faire $|u|$ insertions.
- Pour tous mots u, v et lettre a, $d(au,av) = d(u,v)$: il suffit de transformer u en v en laissant en permanence le a devant.
- Pour tous mots u, v et lettres a, b distinctes,

$$d(au,bv) = 1 + \min\big(d(au,v), d(u,bv), d(u,v)\big).$$

En effet, il existe un chemin minimal de au à bv qui se termine par ajouter b devant v ou supprimer a dans au, ou transformer a en b.

2.2 Résolution

Cet exemple est présenté sous forme d'exercice.

1. Dans le dernier cas, préciser quelle est l'opération élémentaire cachée derrière le "1+", selon que le minimum est $d(au,v), d(u,bv)$, ou $d(u,v)$.
2. Écrire une fonction récursive calculant la distance entre deux mots.
   Il pourra être plus simple d'utiliser une fonction auxiliaire `aux` telle que pour tout i, j `aux i j` calcule $d(u[i:], v[j:])$ c'est-à-dire `aux i j` calcule la distance entre le mot formé des $|u| - i$ dernières lettres de u et celui formé des $|v| - j$ dernières lettres de v.
3. Pour tout $n \in \mathbb{N}$, on note C_n le nombre de comparaisons entre lettres lorsque $|u| + |v| = n$. Montrer qu'il existe des cas où $2^n = O(C_n)$. Ainsi la complexité est exponentielle.
4. Écrire une version memoïzée de la fonction précédente. (Un simple tableau suffira.)
5. Quelle est la nouvelle complexité ?
6. Faire tourner l'algorithme à la main sur un papier pour calculer la distance entre « bla » et «blog ».
7. Dans quel ordre simple peut-on remplir les cases du tableau ? En déduire une version dynamique de l'algorithme précédent.
8. La version dynamique remplit-elle des cases inutiles ?
9. Quel est l'espace mémoire utilisé ? Bonus : optimiser votre algorithme pour que sa complexité spatiale soit $O(\max(|u|,|v|))$.
10. *Calcul de l'argmin :* Écrire une fonction prenant en entrée le tableau créé par un des programmes précédents et un déduisant la liste des transformations à effectuer pour transformer u en v en un minimum d'étapes.

1

- Dans le cas où $d(au,bv) = 1 + d(au,v)$, c'est qu'il existe un plus court chemin de au à bv qui consiste à transformer au en v puis à insérer b devant.
- dans le cas où $d(au,vb) = 1 + d(u,bv)$, c'est qu'il existe un plus court chemin de au à vb qui consiste à supprimer le premier a puis à transformer u en bv.
- Enfin dans le cas où $d(au,bv) = 1 + d(u,v)$, c'est qu'il existe un plus court chemin de au à bv qui consiste à changer u en v, en laissant le a devant, puis à changer ce a en b.

2

```python
def d_naïf(u,v):
    n, p = len(u), len(v)

    def aux(i,j):
        """ Renvoie d(u[i:], v[j:]) """
        if i==len(u):
            return p-j
        elif j==len(v):
            return n-i
        elif u[i]==v[j]:
            return aux(i+1,j+1)
        else:
            return 1+ min(
                aux(i+1, j  ),
                aux(i  , j+1),
                aux(i+1, j+1)
                )

    return aux(0,0)
```

3

4

```python
def d_memo(u,v):
    n, p = len(u), len(v)
    cache = [ [-1 for j in range(p+1)] for i in range(n+1)] # format (n+1, p+1). cache[i][j] contient -1 ou d(u[i:], v[j:])

    # Cas de base
    for i in range(n+1):
        cache[i][p] = n-i
    for j in range(p+1):
        cache[n][j] = p-j

    # Fonction principale
    def aux(i,j):
        if cache[i][j] != -1:
            return cache[i][j]
        elif u[i]==v[j]:
            res = aux(i+1, j+1)
            cache[i][j] = res
            return res
        else:
            res = min(
                1 + aux(i+1, j),
                1 + aux(i, j+1),
                1 + aux(i+1, j+1)
                )
            cache[i][j] = res
            return res

    # Lancement de la fonction principale
    return aux(0,0)
```

5 Comme chaque case du tableau est remplie au plus une fois, il y a au plus $|u| \times |v|$ appels récursifs. Chacun de ces appels coûte $O(1)$, d'où une complexité en $O\left(|u| \times |v|\right)$.

7 Pour remplir une case `cache[i][j]` il nous faut connaître les cases `cache[i+1][j+1]`, `cache[i+1][j]` et `cache[i][j+1]` (les deux derniers uniquement dans le cas où `u[i]v[j]`). En conséquence de quoi il nous faut utiliser des boucle for décroissantes.
Remarque : On dirait toujours qu'on a une approche « bottom-up » ici bien qu'on commence par les grandes valeurs de i et j.

```python
def d_dyna(u,v):
    # Le début est le même que pour la version mémoïsée
    n, p = len(u), len(v)
    cache = [ [-1 for j in range(p+1)] for i in range(n+1)]

    for i in range(n+1):
        cache[i][p] = n-i
    for j in range(p+1):
        cache[n][j] = p-j

    # Maintenant on remplit le tableau par des boucles
    # Attention : des boucles à l'envers ici
    for i in range(n-1,-1,-1):
        for j in range(p-1,-1,-1):
            if u[i]==v[j]:
                cache[i][j] = cache[i+1][j+1]
            else:
                cache[i][j] = 1 + min(cache[i+1][j], cache[i][j+1], cache[i+1][j+1] )

    return cache
```

8 Oui !

10

```python
def argmin( u, v):
    """ Liste des transformations changer u en v """

    cache = d_dyna(u,v)
    res=[]
    def aux(i,j):
        """ Rajoute dans res les transformations pour changer u[i:] en v[j:]."""
        if i==len(u) and j==len(v):
            None
        elif i==len(u):
            res.append(f"insérer {v[j:]}")
        elif j==len(v):
            res.append(f"supprimer {u[i:]}")
        elif cache[i][j]==cache[i+1][j+1]:
            #pas de tranformation à faire à ce moment
            aux(i+1, j+1)
        elif cache[i][j]==1+cache[i+1][j]:
            res.append( f"supprimer {u[i]}" )
            aux(i+1,j)
        elif cache[i][j]==1+cache[i][j+1]:
            res.append( f"insérer {v[j]}")
            aux(i,j+1)
        else:
            res.append(f"changer {u[i]} en {v[j]}")
            aux(i+1,j+1)
    aux(0, 0)
    return res
```

3 Programation dynamique

3.1 Méthode générale

Souvent, la programmation dynamique sera utilisée dans des problèmes d'optimisation, c'est-à-dire de calcul de maximum (ou de minimum, cela revient au même après échange de la relation d'ordre). Pour fixer les idées supposons que c'est le cas ici. Notons f la fonction dont on cherche le maximum et E son domaine de définition.

1. On trouve une relation entre la solution du problème et les solutions de problèmes plus petits (chercher le maximum de f sur des sous-ensembles de E), ce qui permet d'écrire une fonction récursive pour calculer le minimum ou le maximum cherché.

2. On optimise le temps de calcul de cette fonction par mémoïzation.
3. On peut souvent écrire aussi une version impérative consistant à remplir une matrice dans un certain ordre. C'est la version « bas en haut» (bottom-up), la précédente étant la version « haut en bas » (top-down).
   La version impérative aura la préférence des afficionados de la boucle. Au niveau des avantages :
   - Code final plus court, avec de simples boucle for ;
   - Possibilité selon le cas d'utiliser moins de mémoire (si on peut effacer les données devenues inutiles au fur et à mesure).

   Et pour les inconvénients :
   - Pas toujours facile, voire possible (voir 4.1) à mettre en place. Dans tous les cas, nécessite de la réflexion, ce qui entraîne un risque d'erreur supplémentaire. Au contraire la version mémoïzée suit exactement la relation de récurrence mathématique, ce qui est plus sûr.
   - En général on va calculer des valeurs inutiles puisqu'on remplit quoi qu'il arrive *tout* le cache.
4. On peut modifier la fonction pour renvoyer non seulement le maximum cherché, mais également la solution (l'élément de E) permettant d'obtenir ce maximum (l'« argmax »).
5. De manière alternative, on peut aussi prendre le cache rempli par la fonction qui a calculé le maximum, et reconstruire l'argmax à partir de celui-ci. Méthode plus compliquée mais plus efficace dans le cas où les éléments de E sont représentés en mémoire par un type mutable et que la méthode précédente nécessiterait de nombreuses copies.
   Ainsi avec des listes Caml on peut s'en tenir à la version précédente, en revanche avec les tableaux Python, il vaut mieux utiliser celle-ci.

3.2 Liste d'autres exemples

- Calculer le nombre minimal de pièces à utiliser pour payer une certaine somme (problème du rendu de monnaie).
- Calculer où placer les parenthèses lors du produit de plusieurs matrices pour minimiser le nombre de muliplications de coefficients effectué.
- Calculer le nombre maximum d'objets qu'on peut mettre dans un récipient de taille fixée (problème du sac à dos) [1]. Sans doute l'exemple le plus connu, voir le problème correspondant.
- Répartir plusieurs tâches entre plusieurs machines pour minimiser le temps total d'exécution.
- Calculer la « ressemblance » entre deux séquences d'ADN.

À chaque fois, le point est de définir les bons sous-problèmes, et d'identifier une relation de récurrence entre le problème initial ses sous-problèmes.

4 Utilisation d'un dictionnaire

4.1 Exemple : suites de Syracuse

Soit $p \in \mathbb{N}^*$. La suite de Syracuse de premier terme p est la suite u^p telle que :

$$u_0^p = p \text{ et } \forall n \in \mathbb{N}, u_{n+1} = \left| \begin{array}{ll} \frac{u_n}{2} & \text{si n est pair} \\ 3u_n + 1 & \text{sinon} \end{array} \right. .$$

On écrit facilement un programme pour calculer les termes de la suite de Syracuse. Je choisis ici de renvoyer tous les termes successifs pour mieux voir son comportement.

```
266 def terme_suivant(un):
267     if un%2==0:
268         return un//2
269     else:
270         return 3*un+1
```

1. variantes plus "sérieuse" : optimisation le transport de marchandise en bateau ou avion, minimiser les chutes lors de la découpe d'un matériaux....

```
def syra(p,n):
    u=p
    res=[p]
    for k in range(n):
        # ici, u==u^p_k
        u=terme_suivant(u)
        res.append(u)
        # Maintenant, u==u^p_{k+1}
    return res
```

Une conjecture classique affirme que quelque soit $p \in \mathbb{N}^*$, la suite u^p finit par retomber sur 1 (et à partir de là elle va boucler : 1,4,2,1,4,2,...)

Définition 4.1. *Pour tout $p \in \mathbb{N}^*$, on note $syr(p)$ le plus petit entier tel que $u^p_{syr(p)} = 1$. On l'appelle le temps de vol de u^p.*

On calcule facilement le temps de vol :

```
def temps_de_vol(p):
    t=0
    u=p
    while u!= 1:
        u = terme_suivant(u)
        t+=1
    return t
```

Imaginons maintenant que nous voulions calculer plusieurs valeurs. Par exemple si nous cherchons pour $pmax \in \mathbb{N}^*$ le maximum de $\text{syr}(1), ..., \text{syr}(pmax)$.

Une version naïve est la suivante :

```
def temps_de_vol_max(pmax):
    res=temps_de_vol(1)
    for p in range(2,pmax):
        res = max(res, temps_de_vol(p))
    return res
```

Cependant on peut faire bien plus efficace. En effet soit $p \in \mathbb{N}^*$ et $n \in \mathbb{N}$. Plaçons-nous dans la situation suivante : nous avons calculé u^p_n, et ce n'est toujours pas 1. Mais nous avions déjà calculé le temps de vol de u^p_n, notons-le t_n. Alors nous savons que dans encore t_n étapes, nous arriverons enfin à 1. Donc le temps de vol de u^p est $n + t_n$, inutile de poursuivre les calculs !

Ainsi, il serait judicieux d'enregistrer tous les temps de vol calculés au fur et à mesure. Mais la suite de Syracuse étant complètement imprévisible, on ne sait pas à l'avance de quelles valeurs on aura besoin ! Par exemple $u^3_1 = 10$, donc pour calculer syr(3) on passe par syr(10).

Dans cette situation, l'utilisation d'un tableau est malcommode car on ne peut même pas savoir combien de cases prévoir au moment de créer le tableau. Éventuellement un tableau redimensionnable, mais il reste le défaut qu'il y aura de nombreuses cases créées inutilement.

La structure la mieux adaptée est ici le dictionnaire.

Attention : une mémoïzation automatique telle que nous l'avons fait jusqu'ici dans ce chapitre ne serait pas efficace. En effet, il n'a a pas ici d'appels récursifs redondants. Le principe est que pendant le calcul d'un temps de vol, d'une part nous calculons de nombreux autres temps de vol : celui de toutes les valeurs intermédiaires, et d'autre part nous pouvons interrompre le calcul dès que nous rencontrons une valeur connue.

C'est donc la fonction `temps_de_vol` que nous allons modifier. Mais le cache devra être créé par la fonction `temps_de_vol_max` afin d'être utilisable pour tous les appels à `temps_de_vol`. On comprend qu'il va nous falloir placer la fonction `temps_de_vol` dans la fonction `temps_de_vol` pour qu'elle puisse accéder au cache.

Enfin, pour gérer le fait de pouvoir renvoyer directement un résultat dès qu'on rencontre une valeur connue, je trouve plus commode d'utiliser une fonction récursive.

À titre d'échauffement, commenons par la fonction `temps_de_vol` simple, mais en version récursive.

```
def temps_de_vol(p):
    if p==1:
        return 0
    else:
        return 1+ temps_de_vol(terme_suivant(p))
```

Puis passons à la version mémoïzée de `temps_de_vol_max` :

```
314 def temps_de_vol_max_mémo(pmax):
315     cache = {1:0} #cache[p] contient le temps de vol de u^p
316 
317     def temps_de_vol(p):
318         if p in cache :
319             return cache[p]
320         else:
321             res = 1 + temps_de_vol(terme_suivant(p))
322             cache[p]=res
323             return res
324 
325     maxi=0
326     for p in range(2,pmax):
327         maxi=max(maxi, temps_de_vol(p))
328     return maxi
```

4.2 Mémoïzation systématique

N'importe quelle fonction peut être mémoïzée au moyen d'un dictionnaire. En effet les deux difficultés lors de la création d'un cache sont :

- Déterminer la taille à prévoir ;
- Trouver un moyen de reconnaître les cases remplies des autres.

Et ces deux difficultés n'existent plus dès qu'on a décidé d'utiliser un dictionnaire !

Ci-dessous un squelette de fonction pour mémoïzer une fonction `f`, appliquable à n'inporte quelle fonction.

```
333 def f_mémo(args0):
334 
335     cache = {}
336     # Remplir cache avec les cas de base
337 
338     def aux(args):
339         if args in cache:
340             return cache[args]
341         else:
342             #mettre ici le code de f.
343             #On suppose que le résultat de f(args) a été enregistré sous le nom de res
344             cache[args]=res
345             return res
346 
347     return aux(args0)
```

4.3 Bonus : mémoïzation automatique

On peut même écrire une fonction qui prend en entrée une fonction quelconque et qui renvoie cette fonction mémoïzée. Avertissement : ce paragraphe est déconseillé aux âmes sensibles, et déborde largement du programme.

```
352 def memoize(f):
353     cache={}
354     def f_memo(*args):
355         if args not in cache:
356             cache[args] = f(*args)
357         return cache[args]
358     return f_memo
```

En l'état, cette fonction ne renvoie pas une version mémoïzée de la fonction f passée en argument. En effet, l'appel `f(*args)` qa'on retrouve dans `f_memo` lance la vieille fonction, inefficace.

Ainsi le code suivant :

```
362 def fiboNaif(n):
363     if n==0:return 0
364     elif n==1:return 1
365     else:
366         return fiboNaif(n-1)+fiboNaif(n-2)
367 
368 fibo = memoize(fiboNaif)
```

ne mémoïze aucunement la fonction `fiboNaïf`.

L'astuce est la suivante :

```
375 def fibo(n):
376     if n==0:return 0
377     elif n==1:return 1
378     else:
379         return fibo(n-1)+fibo(n-2)
380 
381 fibo=memoize(fibo)
```

Et enfin pour la frime, avec des décorateurs :

```
388 @memoize
389 def fibo(n):
390     if n==0:return 0
391     elif n==1:return 1
392     else:
393         return fibo(n-1)+fibo(n-2)
```

Si `F` est une fonction que prend une fonction et renvoie une fonction, alors la ligne `@F` placée avant la définition d'une fonction `f` consiste précisément à exécuter `f=F(f)` après avoir défini `f`, donc le code ci-dessus est équivalent au précédent.

Mais à présent, il suffit de taper `@memoize` avant la définition de *n'importe quelle* fonction pour qu'elle soit automatiquement mémoïzée !

Première partie

Exercices

Exercices : programmation dynamique

1 Programmation dynamique

Exercice 1. $S(n,k)$ = nombre de partitions de $[\![0,n[\![$ en k parties.
$S(n,k) = S(n-1,k-1) + kS(n-1,k)$.
Naïf puis dynamique.

Exercice 2. ** Appel (E3A 2017)

Dans une classe de n élèves, les élèves sont numérotés de 0 à $n-1$. Un professeur souhaite faire l'appel, c'est à dire déterminer quels élèves sont absents.

Une salle de classe est décrite par un tableau à n entrées. Si `tab` est un tel tableau et `i` un entier de $[\![0,n[\![$, alors `tab.(i)` donne le numéro de l'élève assis à la place i, ou -1 si cette place est vide.

1. Écrire une fonction `asseoir : int list → int → int vect`, qui prend en argument une liste non vide d'entiers distincts et un entier n, et renvoie un tableau représentant une salle de classe pour n élèves où chaque élève de la liste à été assis à la place numérotée par son propre numéro. Les entiers supérieurs ou égaux à n seront ignorés.
2. En déduire une fonction `absent2 : int list → int → int list` qui étant donné une liste non vide d'entiers distincts et un entier n, renvoie la liste des entiers de $[0;n-1]$ qui n'y sont pas. Les entiers supérieurs ou égaux à n seront ignorés.
3. En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de lectures et d'écritures dans un tableau (en fonction de n et k) de la fonction précédente ?

Exercice 3. **! Parenthésage optimal pour un produit de matrices

Soit $n \in \mathbb{N}$ et $A_1, \dots, A_n$ n matrices de formats tels que le produit $A_1 \times \cdots \times A_n$ soit bien défini. On rappelle que le produit matriciel est associatif : les parenthèses peuvent être placées comme on le souhaite dans ce produit.

On désire calculer comment placer les parenthèses pour minimiser le nombre de multiplications à effectuer. On notera pour tout $i \in [\![1,n]\!]$, l_i et c_i le nombre de lignes et colonnes de A_i.

1. Quelle est la condition sur les c_i, l_i pour que le produit soit bien défini ?
   En pratique, les fonctions à suivre prendront en entrée le vecteur `[|c0; ...; cn|]` tel que pour tout i où cela a du sens, c_i est le nombre de lignes de la $(i+1)$-ème matrice et aussi le nombre de colonnes de la i-ème.
2. Soit $i \in [\![1,n]\!]$. Quel est le nombre de multiplications scalaires pour calculer $A_i \times A_{i+1}$?
3. Plus généralement, soit $(d,k,f) \in [\![1,n]\!]^3$ tel que $d \leqslant k \leqslant f$. Quel est le nombre de multiplications scalaires pour calculer la multiplication matricielle $\left(A_d \dots A_k\right)) \times \left(A_{k+1} \dots A_f\right)$?
4. Soit $(d,f) \in [\![1,n]\!]^2$ tel que $d \leq f$. Pour calculer le parenthésage optimal pour $(A_d \times \cdots \times A_f)$, le principe est d'essayer pour tout $k \in [\![d, f-1]\!]$ de faire la k-ème multiplication en dernier (c'est-à-dire un parenthésage de type $(A_d \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_f)$) et de garder le minimum.
   On note pour tout $(d,f) \in [\![1,n]\!]^2$ tel que $d \leqslant f$, $N_{d,f}$ le nombre minimal de multiplications scalaires pour calculer $A_d \dots A_f$. Écrire la formule basée sur le principe ci-dessus permettant d'exprimer $N_{d,f}$. Préciser le cas de base.
5. Écrire une fonction récursive basée sur ce principe.
6. Démontrer que votre fonction termine.
7. Montrer que sa complexité est exponentielle.
8. Mémoïzer cette fonction.
9. écrire une version impérative de cette fonction.
   Quelle est sa complexité ?
10. Écrire une version de cette fonction qui renvoie la liste des produits à effectuer, dans le bon ordre, pour que le calcul du produit matriciel soit optimal. Pour simplifier la programmation, on mettra la dernière multiplication à effectuer en tête de liste.

Exercice 4. * Stratégie gagnante pour un jeu de dépilage** [2]

On considère une pile d'entiers positifs `p` de longueur n et deux joueurs A et B.

- Les deux joueurs jouent l'un après l'autre. Chacun doit dépiler un certain nombre d'éléments de la pile, celui qui ne peut plus (car la pile est vide) perd.

2. Variante du jeu de Nim

- La règle est la suivante : à son tour un joueur a le choix entre :
  - ◇ dépiler un élément
  - ◇ dépiler k éléments, où k est l'entier actuellement au sommet de la pile. Opération autorisée uniquement s'il reste au moins k éléments dans la pile bien sûr.

A joue en premier. Le but est d'écrire une fonction qui calcule s'il existe une stratégie gagnante pour A.

On pourra maintenir un tableau de booléens `G` tel que pour tout i, `G.(i)` indique s'il est possible au joueur à qui c'est le tour de gagner lorsqu'il reste i éléments dans la pile.

Écrire une formule de récurrence permettant de remplir `G` (tout simplement de la case 0 à la case n), et programmer une fonction répondant au problème.

Exercice 5. ** Plus longue sous-séquence strictement croissante**

Écrire une fonction qui prend en entrée un tableau d'entiers `t` et qui renvoie la longueur de la plus longue sous-suite d'éléments de `t` strictement croissante.

Application : L'avenue des flots bleues est perpendiculaire à la plage. Le long de cette avenues s'élèvent des immeubles dont le nombre d'étages est 2,3,4,1,2,5,10,11,9,8. Quels immeubles faut-il raser pour que les immeubles restant aient tous vue sur la mer ?

Exercice 6. * Alignement de séquences génomiques**

Un génome est un mot formé à l'aide des lettres 'A', 'T', 'G', 'C'. Voici une méthode fréquemment utilisée pour estimer la différence entre deux génomes, très proche du calcul de la distance d'édition.

On introduit une nouvelle lettre, '-'.

Soient deux génomes u et v. Un « alignement » de u et v est une matrice de deux lignes dont la première contient u dans lequel sont éventuellement insérés un ou plusieurs '-', et la seconde ligne est v, dans lequel sont également éventuellement insérés un ou plusieurs '-'. Aucune colonne ne peut contenir deux '-'.

Par exemple $\left|\begin{array}{cccccc} A & G & - & A & - & - \\ A & - & G & C & T & A \end{array}\right|$ est un alignement entre "AGA" et "AGCTA".

Lorsque les deux lettres de la colonne sont identique, on dit qu'il y a appariement, lorsqu'il y a un '-' on dit qu'il y a une délétion, et lorsqu'il y a deux lettres différentes, on dit qu'il y a mésappariement.

On définit le « score » d'un alignement ainsi : un appariement vaut 4, une délétion ou un mésappariement vaut -1. Pour tout $(x, y) \in \{A, G, C, T, -\}^2$, on notera $\delta(x, y)$ le score entre les deux lettres x et y.

Le score total de l'alignement est obtenu en sommant le score des deux lettres de chaque colonne. Par exemple, le score de l'alignement ci-dessus est -1.

Le but est de déterminer l'alignement entre deux mots u et v qui réalise le score total maximal.

On notera pour tout $(i, j) \in [\![0, |u|]\!] \times [\![0, |v|]\!]$, $s_{i,j}$ le score maximal entre les sous-mots $u_1...u_i$ et $v_1...v_j$. On convient que $s_{0,0} = 0$.

1. Programmer la fonction δ.
2. Soit $i \in [\![1, |u|]\!]$, que vaut $s_{i,0}$? De même, soit $j \in [\![1, |v|]\!]$, que vaut $s_{j,0}$?
3. Justifier que $(i, j) \in [\![1, |u|]\!] \times [\![1, |v|]\!]$,

$$s_{i,j} = \max\left\{s_{i-1,j} + \delta(u_i, -), \quad s_{i,j-1} + \delta(-, v_j), \quad s_{i-1,j-1} + \delta(u_i, v_j)\right\}.$$

   Dans quel cas est atteint chacun des 3 nombres dans le calcul du maximum ?
4. En déduire une fonction pour calculer si, j.

Quelques indications

7 Vérifier que $\forall n \in \mathbb{N}$, $C_n \geq 2C_{n-1}$.

9 remplir le tableau de base en haut.

5 Maintenir un tableau `t` tel que pour tout `k`, `t.(k)` est le dernier élément d'une plus longue sous-suite croissante de longueur de k.

Quelques solutions

2

1 Pour tout $i \in [\![1, n-1]\!]$, il faut que $c_i = l_{i+1}$.

2 Le produit $A_i \times A_{i+1}$ coûte $c_{i-1} \times c_i \times c_{i+1}$ multiplications entre coefficients de la matrice.

3 $c_{d-1} \times c_k \times c_f$

4

$$\forall (d, f) \in [\![1, n]\!]^2 \text{ tq } d < f,\ N_{d,f} = \min_{k \in [\![d, f-1]\!]} N_{d,k} + N_{k+1,f} + c_{d-1} c_k c_f$$

et si $d = f$, alors $N_{d,f} = 0$ (une seule matrice, donc aucune multiplication à faire.)

5

```
let nb_mult c =
  (* c est le tableau des nb de colonnes. c.(0) est le nb de lignes de la première matrice.
   Renvoie le nb min de mult scalaires pour calculer A_1 × … × A_n *)

  let rec aux i j =
    (* Renvoie le nb min de mult scalaires pour calculer A_i × … × A_j *)

    (* Cas de base *)
    if i=j then 0


    else (* cas général *)
      (* Il faut calculer le minimum pour k de i à j-1 de :
           (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j)
       *)
      let essai k = (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j) in (* Pour éviter de
    ↪ recopier la formule plus tard *)

      let rec boucle k =
        if k = j-1 then essai (j-1)
        else min (essai k) (boucle (k+1))
      in
      boucle i

  in
  aux 1 (Array.length c -1 )
;;


(* En faisant le calcul du min par une boucle for *)
let nb_mult c =
  (* c est le tableau des nb de colonnes. c.(0) est le nb de lignes de la première matrice.
   Renvoie le nb min de mult scalaires pour calculer A_1 × … × A_n *)

  let rec aux i j =
    (* Renvoie le nb min de mult scalaires pour calculer A_i × … × A_j *)

    (* Cas de base *)
    if i=j then 0


    else (* cas général *)
      (* Il faut calculer le minimum pour k de i à j-1 de :
           (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j)
       *)
      let essai k = (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j) in

      let mini = ref (essai i) in
      for k = i+1 to j-1 do
        mini := min !mini (essai k)
```

```
      done;
      !mini

  in
  aux 1 (Array.length c -1)
;;
```

8

```
let nb_mult_memo c =
  (* c est le tableau des nb de colonnes. c.(0) est le nb de lignes de la première matrice.
   Renvoie le nb min de mult scalaires pour calculer A_1 × … × A_n *)
  let n = Array.length c - 1 in
  let cache = Array.make_matrix (n+1) (n+1) (-1) in

  let renvoie i j res =
    cache.(i).(j) <- res;
    res
  in

  let rec aux i j =
    (* Renvoie le nb min de mult scalaires pour calculer A_i × … × A_j *)
    if cache.(i).(j) <> -1 then cache.(i).(j)
    else

      (* Cas de base *)
      if i=j then renvoie i j 0


      else (* cas général *)
        (* Il faut calculer le minimum pour k de i à j-1 de :
           (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j)
         *)
        let essai k = (aux i k) + c.(i-1)*c.(k)*c.(j) + (aux (k+1) j) in (* Pour éviter de
    ↪ recopier la formule plus tard *)

        let rec boucle k =
          if k = j-1 then essai (j-1)
          else min (essai k) (boucle (k+1))
        in
        renvoie i j (boucle i)

  in
  aux 1 (Array.length c -1 )
;;
```

9

```
let nb_mult_dyna c =
  (* c est le tableau des nb de colonnes. c.(0) est le nb de lignes de la première matrice.
   Renvoie le nb min de mult scalaires pour calculer A_1 × … × A_n *)
  let n = Array.length c - 1 in
  let cache = Array.make_matrix (n+1) (n+1) 0 in

  (* cas de base -> initialisation : mettre des 0 sur la diag. C'est déjà fait !*)

  for i = n downto 1 do
    for j = i+1  to n do
      let essai k = cache.(i).(k) + c.(i-1)*c.(k)*c.(j) + cache.(k+1).(j) in

      let mini = ref (essai i) in
      for k = i+1 to j-1 do
        mini := min !mini (essai k)
```

```
16        done;
17        cache.(i).(j) <- !mini
18      done
19    done;
20    cache.(1).(n)
21  ;;
```

---

4 On trouve $G.(0) = faux$ et pour tout $i \in [\![1, n]\!]$, $G.(i) =$ non $(G.(i-1))$ ou non $(G.(i - p.(i)))$ si $p.(i) \leqslant i$ et $G.(i) =$ non $(G.(i-1))$ sinon.

5

6