# TD #2 — Programmation dynamique (avec solutions)

**Exercice 1.** *Stratégie de révision*

*Repris d'un exercice de Bruno Grenet*

Une élève prévoit son programme de révision avec la stratégie suivante : chaque jour peut être un jour de travail *tranquille*, un jour de travail *soutenu*, ou un jour de *repos*, avec la contrainte qu'un jour *soutenu* doit être précédé d'un jour de *repos*. L'élève sait estimer pour chaque jour $i$ le nombre $t_i$ de points qu'elle obtiendra avec un travail *tranquille* et le nombre $s_i$ de points avec un travail *soutenu*. Un jour de repos ne lui fait gagner aucun point.

Par exemple, sur la période suivante, une stratégie optimale est d'être en repos les jours 1 et 4, en travail tranquille le jour 3, et en travail soutenu les jours 2 et 5, ce qui rapportera $2,5$ points.

| Jour | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $t$ | 0,3 | 0,5 | 0,4 | 0,6 | 0,2 |
| $s$ | 0,5 | 1,2 | 1 | 0,8 | 0,9 |

1. Montrez que l'algorithme (à tendance gloutonne) suivant n'est pas optimal (avec $n$ le nombre de jours de la période de révision).

```
1 i ← 1
2 Tant que i ≤ n :
3    Si i ≤ n−1 et s_{i+1} > t_i + t_{i+1} :
4       Repos le jour i et travail soutenu le jour i+1
5       i ← i+2
6    Sinon :
7       Travail tranquille le jour i
8       i ← i+1
```

Cet algorithme ne prend pas en compte les jours à horizon plus que 1, et est donc nécessairement sous-optimal pour certaines entrées. Par exemple, pour $t_1 = t_2 = t_3 = 1$, $s_1 = 1$, $s_2 = 10$, $s_3 = 100$, le programme renvoyé serait *repos ; soutenu ; tranquille* pour 11 points, alors que *tranquille ; repos ; soutenu* rapporterait 101 points !

2. Proposez une formule récursive pour calculer $p_i$ le nombre maximum de points obtenu en travaillant jusqu'au jour $i$.

Le premier jour ne peut être qu'un repos (qui ne rapporte pas de points) ou tranquille et on a donc $p_1 = t_1$. Tout autre jour peut ou bien être précédé d'un repos ce qui permet de fournir un travail soutenu, ou précédé d'un jour de travail ce qui impose un travail « seulement » tranquille. La meilleure stratégie s'obtient récursivement en prenant le maximum des deux cas, soit $p_i = \max(p_{i-2} + s_i, p_{i-1} + t_i)$ (en prenant $p_{\leq 0} = 0$)·

3. Écrivez en Python un algorithme (itératif) de programmation dynamique pour calculer $p_n$ et analysez sa complexité en temps et en espace. On suppose que les valeurs $s_i$ et $t_i$ sont respectivement stockées dans un tuple `A` comme `A[i][0]` et `A[i][1]`, et que les indices $i$ commencent désormais à 0.

```python
def ST(A, n):
  P = [0]*(n+1)
  P[0] = A[0][0]
  P[1] = max(P[0] + A[1][1], A[1][0])
  for i in range(2,n+1):
     P[i] = max(P[i-1] + A[i][1], P[i-2] + A[i][0])
  return P[n]
```

Le coût en temps et en espace est un $O(n)$

4. Pouvez-vous (éventuellement) réduire la coût spatial de votre algorithme précédent ?

On peut obtenir un coût constant en ne stockant que `P[i-1]` et `P[i-2]`. Cela n'est cependant pas forcément significatif, puisque l'entrée même du problème a un coût de stockage en $O(n)$.

**5.** Modifiez votre algorithme précédent pour qu'il renvoie maintenant une stratégie optimale, en plus de la seule quantité de points qu'elle rapporte.

On peut ajouter une variable supplémentaire qui enregistre pour chaque $i$ le type de jour pour une stratégie se terminant au jour $i$, ou déduire cette information en utilisant la donnée de `P` : par exemple si `P[i] == P[i-2] + A[i][0]` on sait qu'une stratégie optimale *possible* se terminant en $i$ est de se reposer le jour $i-1$ et fournir un travail soutenu le jour $i$ ; on reconstruit ensuite la stratégie en partant de $n$. On peut remarquer qu'ici utiliser une variable supplémentaire ne nécessite pas vraiment de stockage supplémentaire car elle peut ensuite être utilisée pour stocker le résultat, mais il est parfois intéressant de chercher à s'en passer. Dans tous les cas, on ne peut pas (efficacement) se passer d'un stockage en $O(n)$ si l'on souhaite renvoyer une stratégie.

```
def ST(A, n):
  P = [0]*(n+1)
  P[0] = A[0][0]
  P[1] = max(P[0] + A[1][1], A[1][0])
  for i in range(2,n+1):
     P[i] = max(P[i-1] + A[i][1], P[i-2] + A[i][0])
  S = [' ']*(n+1)
  j = n
  while j > 0:
    if P[j] == P[j-1] + A[j][1]:
        S[j] = 'T'
        j = j - 1
    else: # P[j] == P[j-2] + A[j][0]
        S[j] = 'S'
        S[j-1] = 'R'
        j = j - 2
  if S[0] == ' ':
    S[0] = 'T'
  return P[n], S
```

**Exercice 2.** *Plus grande somme, plus grand produit consécutif*

*Grand classique, dans une version adaptée de Jeff Erickson.*

Soit une liste $A$ de nombres non nécessairement positifs et non nécessairement entiers, on souhaite trouver la valeur de la plus grande somme de termes consécutifs de $A$. Par exemple, pour :

`A = [-6, 12, -7, 0, 14, -7, 5]`

cette valeur est 19 et est obtenue en sommant les éléments d'indice 1 à 4 (inclus). Pour `A = [-1]`, cette valeur est 0 et est obtenue en sommant les éléments de la sous-liste vide d'indice 0 à 0 (exclus).

Dans tout ce qui suit, on considère que l'accès à l'élément d'une liste ainsi que sa modification, les tests d'(in)égalité, les opérations arithmétiques on toutes un coût constant. Vous pouvez également si vous le souhaitez utiliser la fonction Python `max` qui renvoie en temps linéaire l'élément maximum d'une `list` (par exemple).

**1.** Décrivez un algorithme qui permet de résoudre ce problème en un temps quadratique en le nombre d'éléments $l$ de $A$. (On ne demande pas ici d'écrire formellement l'algorithme, mais une argumentation convaincante justifiant sont existence et son coût.)

Pour chaque indice $i$, on calcule les sommes des sous-listes d'indice $i$ à $j$, pour $j$ de $i+1$ à $l-1$ et on garde un maximum. Ceci coûte au plus $O(l)$ opérations de coût constant pour un indice $i$ donné (à condition de calculer les sommes de façon incrémentale), et donc au plus $O(l^2)$ opérations au total.

**2.** Écrivez une fonction Python `largestSum(A:list[int])` qui résout le problème ci-dessus, et **dont le coût est linéaire en le nombre d'éléments de** $A$. Il n'est pas demandé d'établir explicitement le coût de votre fonction, mais celui-ci devra effectivement être linéaire.
*Indice* : on conseille d'utiliser une approche « par programmation dynamique » qui utilise une liste auxiliaire $Aj$ construite de façon à ce que `Aj[i]` contienne la valeur de la plus grande somme d'éléments consécutifs se terminant en $i$.

```
def largestSum(A):
    l = len(A)
```

```
    Aj = [None]*l
    if A[0] < 0:
        Aj[0] = 0
    else:
        Aj[0] = A[0]
    for i in range(1,l):
        if Aj[i-1] <= 0: # sommer avec ce qui précède n'augmente pas la valeur
            if A[i] < 0: # négatif : inutile pour la suite
                Aj[i] = 0
            else:
                Aj[i] = A[i]
        else: # Aj[i-1] > 0
            t = Aj[i-1] + A[i]
            if t > 0:
                Aj[i] = t
            else: # la somme est négative : inutile pour la suite
                Aj[i] = 0
    return max(Aj)
```

On considère maintenant la variante du problème consistant à calculer la valeur (positive) du plus grand produit de termes consécutifs de $A$. Par exemple, pour `A = [-6, 12, -7, 0, 14, -7, 5]`, cette valeur est 504 et est obtenue en multipliant les trois premiers éléments de $A$. Pour `A = [-7]`, cette valeur est 1 et est obtenue en multipliant les éléments de la sous-liste vide d'indice 0 à 0 (exclus).

**3.** Écrivez une fonction Python qui résout ce problème en temps linéaire en le nombre d'éléments de $A$.

On procède de même que pour le calcul de la somme en version « programmation dynamique » en utilisant une liste *Apos* contenant en `Apos[i]` la valeur du plus grand produit se terminant en $i$, mais en maintenant également une liste *Aneg* contenant en `Aneg[i]` la valeur du plus petit produit (négatif) se terminant en $i$. Cette liste est utile car un « grand » produit peut éventuellement être obtenu en multipliant des nombres négatifs. Il faut alors considérer un certain nombre de cas pour remplir itérativement ces listes, en fonction du signe et de la valeur de `A[i]`.
À d'éventuelles erreurs d'analyse de cas près, on obtient :

```
def largestProd(A):
    l = len(A)
    Aneg = [None]*l
    Apos = [None]*l
    if A[0] < 1:
        Apos[0] = 1
    else:
        Apos[0] = A[0]
    if A[0] < 0:
        Aneg[0] = A[0]
    else:
        Aneg[0] = 1

    for i in range(1,l):
        if A[i] > 1:
            if Apos[i-1] > 1:
                Apos[i] = A[i] * Apos[i-1]
            else:
                Apos[i] = A[i]
            if Aneg[i-1] < 0:
                Aneg[i] = A[i] * Aneg[i-1]
            else:
                Aneg[i] = 0
        elif A[i] < 0:
            if Aneg[i-1] < 0:
                t = A[i] * Aneg[i-1]
                if t > 1:
                    Apos[i] = t
```

```
                else:
                    Apos[i] = 1
            else:
                Apos[i] = 1
            if Apos[i-1] > 1:
                Aneg[i] = A[i] * Apos[i-1]
            else:
                Aneg[i] = A[i]
        else: # 0<= A[i] <= 1
            if Aneg[i-1] < 0:
                Aneg[i] = Aneg[i-1] * A[i]
            else:
                Aneg[i] = 0
            t = A[i] * Apos[i - 1]
            if t > 1:
                Apos[i] = t
            else:
                Apos[i] = 1
    return max(Apos), Apos, Aneg
```

**Exercice 3.** *Sac-à-dos*

*Un autre grand classique, dans une version adaptée de Bruno Grenet*

On pose le *problème de sac-à-dos* suivant : soit $\mathcal{O}$ un ensemble de $n$ objets donnés sous la forme de couples $(p_i, v_i)_{0 \le i < n}$ de leur *poids* $p_i$ et de leur *valeur* $v_i$ (supposés tous-deux être des entiers naturels), et soit $T$ le poids total maximum qui peut être transporté dans un sac-à-dos (sa « capacité »), on souhaite remplir le sac-à-dos avec un sous-ensemble $\mathcal{C} \subseteq \mathcal{O}$ de valeur maximale en respectant la contrainte de poids. Autrement dit, on cherche $\mathcal{C}$ maximisant $\sum_{(p_i,v_i)\in\mathcal{C}} v_i$ sous la contrainte $\sum_{(p_i,v_i)\in\mathcal{C}} p_i \le T$ ; on note $V_{\max}$ la valeur maximale ainsi atteinte.

**1.** Décrivez un algorithme résolvant ce problème par recherche exhaustive, et analysez sont coût (spatial et temporel).

> Il suffit d'énumérer les $2^n$ éléments de l'ensemble des parties de $\mathcal{O}$ et de garder la valeur maximale de ceux respectant la contrainte de poids. Le coût en espace est constant, ou un $O(n)$ si l'on souhaite aussi stocker la solution. Le coût temporel (dans le pire cas) est un $O(n2^n)$ si l'on calcule la valeur et le poids de chaque sous-ensemble indépendamment des autres, mais peut être réduit à $O(2^n)$ si l'on utilise une énumération plus efficace, par exemple à partir de code « de Gray ».

**2.** Pour $m < n$ et $t \le T$, on note $V(m, t)$ la valeur maximale que l'on peut atteindre en ne prenant que des objets parmi $(p_0, v_0)$, ..., $(p_m, v_m)$, et avec un poids total maximum $\le t$.

— Donnez un couple $m, t$ tel que $V_{\max} = V(m, t)$.

> Par définition, on a $V_{\max} = V(n-1, T)$

— Que vaut $V(m, t)$ si $t < p_m$ ? Et si $m = 0$ ?

> Si $t < p_m$, l'objet $(p_m, v_m)$ ne peut pas être sélectionné sans violer la contrainte de poids. La valeur maximale atteinte avec les objets d'indices $0, \ldots, m$ est donc la même qu'avec les objets d'indices $0, \ldots, m-1$, et $V(m,t) = V(m-1,t)$. Si $m = 0$, la valeur est maximale si l'unique objet $(p_0, v_0)$ est sélectionné, et il ne peut l'être que si $p_0 \le t$. On a donc $V(0, t) = v_0$ si $p_0 \le t$, et 0 sinon.

— Déduisez une formule récursive pour $V(m, t)$ de ce qui précède.

> On peut simplement prendre $V(0, < p_0) = 0$, $V(0, \ge p_0) = v_0$, et $V(m,t) = \max(V(m-1,t), V(m-1,t-p_m) + v_m)$ si $p_m \ge t$, et $V(m-1,t)$ sinon.

— Écrivez en Python un algorithme de calcul de $V_{max}$, en utilisant un dictionnaire pour la mémoïsation des résultats des appels récursifs.

```
def Vmax(T, O):
  def _V(m, t, O, D): # O = [(p_i, v_i)]
    if m == 0:
      if t < O[0][0]:
          return 0
      return O[0][1]
    # m >= 1
    if (m, t) in D:
      return D[(m,t)]
    elif t <= O[m][0]:
      D[(m,t)] = _V(m-1, t, O, D)
    else:
      v1 = _V(m-1, t, O, D)
      v2 = _V(m-1, t - O[m][0], O, D) + O[m][1]
      D[(m,t)] = max(v1, v2)
    return D[(m, t)]
  D = {}
  return _V(len(O)-1, T, O, D)
```

— Analysez le coût spatial et temporel de votre algorithme, et comparez le avec celui d'une recherche exhaustive.

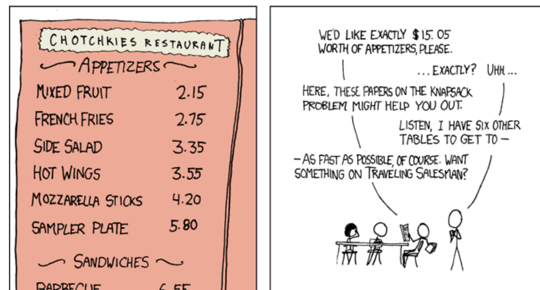
Grâce à la mémoïsation des résultats des appels récursifs, le coût en temps comme en espace est majoré par le nombre d'appel multiplié par le coût de recombinaison, qui est ici constant ; il suffit donc de majorer le nombre d'appels. Les valeurs $V(\cdot, \leq 0)$ et $V(\cdot, > T)$ n'étant jamais calculées, on a donc au plus $nT$ valeurs différentes, et le coût pire cas est un $O(nT)$. La comparaison avec la recherche exhaustive est favorable seulement si T est petit comparé à $2^n$.

**3.** Modifiez votre algorithme afin qu'il renvoie également un sous-ensemble $C$ atteignant la valeur maximale.

Il suffit de stocker pour chaque $V(m, t)$ un sous-ensemble atteignant la valeur calculée, ce qui augmente encore nettement le coût spatial de l'algorithme. Afin de limiter un peu cette augmentation, on ne stocke ici que les indices choisis, et on ne reconstruit réellement la solution complète qu'à la fin de l'exécution.

```
def Vmax(T, O):
  def _V(m, t, O, D): # O = [(p_i, v_i)]
    if m == 0:
      if t < O[0][0]:
          return 0, ()
      return O[0][1], (0,)
    # m >= 1
    if (m, t) in D:
      return D[(m,t)]
    elif t <= O[m][0]:
      D[(m,t)] = _V(m-1, t, O, D)
    else:
      v1, c1 = _V(m-1, t, O, D)
      v2, c2 = _V(m-1, t - O[m][0], O, D)
      v2, c2 = v2 + O[m][1], c2 + (m,)
      if v1 >= v2:
        D[(m,t)] = v1, c1
      else:
        D[(m,t)] = v2, c2
    return D[(m, t)]
  D = {}
  v, c = _V(len(O)-1, T, O, D)
  return v, [O[i] for i in c]
```

**4.** Donnez une entrée $T, O$ qui permet d'utiliser l'algorithme de la question précédente pour trouver une solution à la variante de sac-à-dos donnée dans l'image ci-dessous. Pensez-vous que cette résolution sera efficace ?




https://xkcd.com/287/

Dans cette variante du problème, les objets peuvent être choisis plusieurs fois et le poids et la valeur sont confondus. On peut donc utiliser l'algorithme précédent sans modification à condition de fournir une entrée où chaque objet de prix $i$ est présent sous la forme d'un couple $(i,i)$ au moins autant de fois qu'il peut être choisi. Le fait que les prix ne sont pas entiers n'est pas important, car on peut au besoin renormaliser toutes les quantités pour se ramener à un cas entier (mais ce n'est en fait même pas nécessaire pour utiliser l'algorithme donné dans la solution précédente). Une entrée possible est donc : `O = [(215, 215)]*7 + [(275,275)]*5 + [(335,335)]*4 + [(355,355)]*4 + [(42,42)]*3 + [(58,58)]*2` pour l'ensemble, et `1505` pour la capacité. Cette entrée étant de longueur 25, même une résolution exhaustive serait relativement rapide, mais pas autant que l'approche par programmation dynamique qui aura au plus $25 \times 1505 = 37625$ valeurs $V(m,t)$ à calculer (et en fait nettement moins). Une solution au problème est de commander un *mixed fruit*, deux *hot wings* et un *sampler plate*, et une autre est de simplement commander sept *mixed fruits* ! On remarquera cependant qu'une approche gloutonne consistant à toujours commander l'amuse-bouche le plus cher possible ne marche pas : elle ne permet de dépenser que $14.95.

5. Comment pourriez-vous modifier votre algorithme précédent pour calculer la capacité $T(m,v)$ minimale d'un sac-à-dos de valeur au moins $v$ contenant des objets d'indice 0 à $m$ ?

Ce problème « dual » donne lieu à la récurrence suivante :
$T(0,v) = p_0$ si $v_0 \geq v$, et $\infty$ (ou tout autre symbole approprié) sinon ;
$T(m,v) = \infty$ si $\sum_{i \leq m} v_i < v$, et $\min(T(m-1, v-v_m) + p_m, T(m-1,v))$ sinon. On peut alors utiliser cette récurrence suivant le même schéma que l'algorithme précédent.

**Exercice 4.** *Coefficients binomiaux*

*Encore un classique*

Les coefficients binomiaux $\binom{n}{k}$ vérifient la récurrence suivante : $\binom{n}{0} = \binom{n}{n} = 1$ pour $n \geq 0$ et $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour $0 < k < n$.

1. Écrivez en Python un algorithme de programmation dynamique pour calculer $\binom{n}{k}$ et analysez sa complexité.

```
def CB(n, k):
  def _CB(n, k, L):
    if k == 0 or k == n or n <= 0:
      return 1
    if L[n-1][k-1] == None:
      L[n-1][k-1] = _CB(n-1, k-1, L)
    if L[n-1][k] == None:
      L[n-1][k] = _CB(n-1, k , L)
    return L[n-1][k-1] + L[n-1][k]

  assert(k <= n)
  L = [[None]*(i+1) for i in range(n+1)]
  return _CB(n, k, L)
```

Le coût mémoire est donné par le stockage de la liste L, soit $O(nk)$ (possiblement grands) entiers. Le coût temporel est de $O(nk)$ *additions sur les entiers* : chaque entrée de L est remplie exactement une fois, au coût d'une addition ; pour de grands entiers ce dernier coût n'est cependant généralement pas constant, mais plutôt linéaire en la taille des entiers.

On peut remarquer que cet algorithme n'est rien d'autre que le « triangle de Pascal »

**2.** Écrivez une variante de l'algorithme qui minimise l'espace mémoire utilisé.

Le calcul des coefficients $\binom{n}{*}$ peut se faire en fonction des seuls $\binom{n-1}{*}$ : il suffit donc de stocker deux lignes du « triangle » :

```
def CBrow(n, k):
  assert(k <= n)
  if n <= 1:
    return 1
  P = [1, 1]
  for i in range(2, n+1):
    C = [0]*(min(i+1, k+1)); C[0]=1
    if i <= k:
        C[i]=1
    for j in range(1, min(i, k+1)):
        C[j] = P[j-1] + P[j]
    P = C.copy()
  return C[k]
```