TD #2 — Programmation dynamique

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).

   $i \leftarrow 1$
   Tant que $i \leq n$:
   Si $i \leq n-1$ et $s_{i+1} > t_i + t_{i+1}$:
   Repos le jour i et travail soutenu le jour $i+1$
   $i \leftarrow i+2$
   Sinon :
   Travail tranquille le jour i
   $i \leftarrow i+1$

2. Proposez une formule récursive pour calculer p_i le nombre maximum de points obtenu en travaillant jusqu'au jour i.
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.
4. Pouvez-vous (éventuellement) réduire la coût spatial de votre algorithme précédent ?
5. Modifiez votre algorithme précédent pour qu'il renvoie maintenant une stratégie optimale, en plus de la seule quantité de points qu'elle rapporte.

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.)
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.

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).

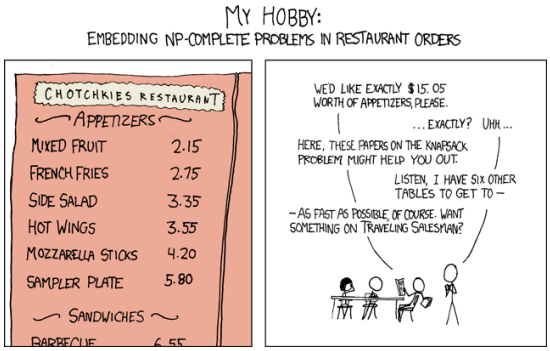
3. Écrivez une fonction Python qui résout ce problème en temps linéaire en le nombre d'éléments de A.

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 $(p_i, v_i)_{0\leq i<n}$ 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 $\sum_{(p_i,v_i)\in\mathcal{C}} p_i \leq T$; 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).
2. Pour $m < n$ et $t \leq T$, 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 $\leq t$.
   - Donnez un couple m, t tel que $V_{\max} = V(m,t)$.
   - Que vaut $V(m,t)$ si $t < p_m$? Et si $m = 0$?
   - Déduisez une formule récursive pour $V(m,t)$ de ce qui précède.
   - Écrivez en Python un algorithme de calcul de V_{max}, en utilisant un dictionnaire pour la mémoïsation des résultats des appels récursifs.
   - Analysez le coût spatial et temporel de votre algorithme, et comparez le avec celui d'une recherche exhaustive.
3. Modifiez votre algorithme afin qu'il renvoie également un sous-ensemble $\mathcal{C}$ atteignant la valeur maximale.
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/

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 ?

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