TD #19 — Paradigmes & texte (avec solutions)

Exercice 1. *Stratégie de révision*

Repris d'un exercice de Bruno Grenet

Un·e 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 que permettra d'obtenir un travail *tranquille* et le nombre s_i de points d'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 OCaml un algorithme (itératif) de programmation dynamique pour calculer p_n et analysez sa complexité en temps et en espace. On suppose que les valeurs t_i et s_i sont respectivement stockées dans un tuple `(float * float) array` comme `fst a.(i)` et `snd a.(i)`, et que les indices i commencent désormais à 0.

```
let strat a =
    let n = Array.length a in
    let p = Array.make n 0. in
    let () = assert (n > 1) in
    let () = p.(0) <- fst a.(0) in (* t0 *)
    let () = p.(1) <- max (p.(0) +. fst a.(1)) (snd a.(1)) in (* max (t0 + t1) s1) *)
    for i = 2 to (n - 1) do
        p.(i) <- max (p.(i- 1) +. fst a.(i))
                     (p.(i- 2) +. snd a.(i))
    done ;
    p.(n - 1)
```

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) +. snd a.(i)` 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.

```
type stratt = Undef | Soutenu | Tranquille | Repos

let strat' a =
    let n = Array.length a in
    let p = Array.make n 0. in
    let s = Array.make n Undef in
    let scnt = ref (n -1) in
    let () = p.(0) <- fst a.(0) in
    let () = p.(1) <- max (p.(0) +. fst a.(1)) (snd a.(1)) in (
    for i = 2 to (n - 1) do
        p.(i) <- max (p.(i- 1) +. fst a.(i))
                     (p.(i- 2) +. snd a.(i))
    done ;
    while !scnt > 0 do
        if p.(!scnt) = p.(!scnt - 1) +. fst a.(!scnt) then (
            s.(!scnt) <- Tranquille ;
            decr scnt )
        else (
            s.(!scnt) <- Soutenu ;
            s.(!scnt - 1) <- Repos ;
            scnt := !scnt - 2 )
    done ;
    if s.(0) = Undef then s.(0) <- Tranquille ;
    p.(n - 1), s )
```

Exercice 2. *Plus longue sous-suite croissante*

Étant donnée une suite finie d'entier (s_i) de longueur n, le problème de la *plus longue sous-suite croissante* consiste à en trouver une sous-suite maximale dont les termes sont tous croissants (pour un ordre non strict $\leq$).

1. Esquissez un algorithme qui résout ce problème par une recherche exhaustive. Quel est son coût en fonction de n ? (On demande une évaluation *assez précise* du coût, qui explicite les éventuelles hypothèse faites.)

   Il y a $2^n - 1$ sous-suites non vides à considérer. Calculer une représentation d'une sous-suite, et vérifier si elle est croissante peut se faire en temps $O(n)$ (les représentations peuvent par exemple être données par l'écriture en base deux d'un compteur que l'on incrémente 1 à $2^n - 1$). On peut donc résoudre ce problème par recherche exhaustive pour un coût $O(n2^n)$.

Pour résoudre ce problème par programmation dynamique, on définit (les solutions à) la famille de sous-problèmes intermédiaires ℓ_k indexée par $k \in [\![n]\!]$ consistant à trouver la **longueur** de la plus longue sous-suite croissante *terminant à l'indice* k.

2. Que vaut ℓ_0 ?

$\ell_0 = 1$.

3. Donnez une expression récursive (en k) de ℓ_k.

$\ell_k = 1 + \max_{i<k \wedge s_i \leq s_k} \ell_i$

4. Pourquoi une implémentation naïve de l'expression récursive ci-dessus serait elle inefficace ?

Une telle implémentation donnerait un coût satisfaisant (à constantes près) la relation de récurrence $T(0) = 1$, $T(k > 0) \leq 1 + \sum_{0 \leq i < k} T(i)$, avec égalité dans le pire cas. Ceci donne alors $T(k) = 2^k$, et donc notamment $T(n) = 2^n$. Il n'est donc pas (beaucoup) plus efficace de résoudre de cette façon le problème de seulement déterminer la longueur d'une plus longue sous-séquence croissante (ce qui nécessite de calculer ℓ_i pour tout $i \in [\![n]\!]$) que par recherche exhaustive.

5. Proposez un algorithme **itératif** efficace pour calculer l'ensemble des solutions ℓ_k, et **analysez son coût**.

On propose (en OCaml) :

```
let plssc_aux a =
    let n = Array.length a in
    let ell = Array.make n 1 in
    for k = 1 to n - 1 do
        let m = ref 0 in
        for i = 0 to k - 1 do
            if a.(i) <= a.(k) then
                m := max !m ell.(i)
        done ;
        ell.(k) <- 1 + !m
    done ;
    ell
```

de coût évidemment quadratique en n.

6. Adaptez votre algorithme pour qu'il permette pour tout $k \in [\![n]\!]$ de reconstruire une plus-longue sous-suite croissante terminant en k.

Pour permettre une telle reconstruction, il est suffisant de construire un tableau p dont l'élément à l'indice k donne un indice i tel que ℓ_i est maximal pour tous les i tels que $s_i \leq s_k$, ou -1 si aucun tel indice n'existe.
On propose (en OCaml) :

```
let plssc_aux' a =
    let n = Array.length a in
    let ell = Array.make n 1 in
    let p = Array.make n (-1) in
    for k = 1 to n - 1 do
        let m = ref 0 in
        for i = 0 to k - 1 do
            if a.(i) <= a.(k) && ell.(i) > !m then (
                m := ell.(i) ;
                p.(k) <- i
            )
        done ;
        ell.(k) <- 1 + !m
    done ;
    ell, p
```

7. Donnez un algorithme qui résout le problème initial.

Il suffit de calculer les solutions aux sous-problèmes ℓ_i's, trouver un maximum parmi ceux-ci, et d'appliquer un algorithme de reconstruction à l'information auxiliaire telle que renvoyée par l'algorithme modifié à la question précédente.
On propose (en OCaml) :

```
let build_sol p k =
    let rec loop i r =
```

```
        if i = (-1) then
            r
        else
            loop p.(i) (i::r)
    in
    loop k []

let plssc a =
    let ell, p = plssc_aux' a in
    let k = ref 0 in
    let () = Array.iteri (fun i v -> if v > ell.(!k) then k := i) ell in
    build_sol p !k
```

Exercice 3. *Sous-mots*

On rappelle qu'un sous-mot u d'un mot v (sur un alphabet quelconque) est une sous-suite de v : $\forall i \in [\![i, |u|]\!]$, $u_i = v_{\varphi(i)}$ pour une certaine fonction strictement croissante $\varphi : [\![1, |v|]\!] \to [\![1, |v|]\!]$

1. Écrivez une fonction C de signature :
   `bool is_subw(char *u, char *v)`
   qui renvoie `true` si la chaîne de caractères `u` représente un sous-mot du mot représenté par la chaîne de caractère `v`, et `false` sinon. Cette fonction devra être de coût linéaire en la taille de ses arguments.

Pour chaque lettre de `u`, on trouve la première lettre de `v` qui lui est égale (et qui n'a pas déjà été mise en correspondance avec une lettre de `u` précédente). On a un sous-mot ssi. l'on arrive à mettre en correspondance toutes les lettres de `u` avec des lettres de `v` de cette façon. Ceci donne par exemple :

```
bool is_subw(char *u, char *v)
{
    size_t us = strlen(u);
    size_t vs = strlen(v);

    size_t i = 0;
    for (size_t j = 0; i < us && j < vs; j++)
    {
        if (u[i] == v[j])
        {
            i += 1;
        }
    }

    return i == us;
}
```

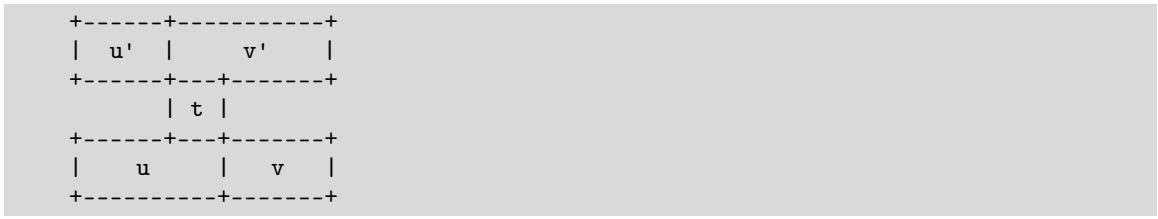
Exercice 4. *Deux propriétés sur les préfixes & suffixes*

Dans tout ce qui suit, on fixe un alphabet Σ arbitraire.

1. Montrez le lemme « de Levi » :
   Soit $u, v, u', v' \in \Sigma^*$ tels que $uv = u'v'$, alors il existe un unique mot t tel que :

   — $u = u't \wedge v' = tv$, ou symétriquement
   — $u' = ut \wedge v = tv'$

Si $u = u'$ alors $v = v'$ et t (unique) est donné par le mot vide ε. Sinon l'on considère sans perte de généralité que $|u'| < |u|$ et l'on se contente de montrer le premier cas (le second est le symétrique pour $|u| < |u'|$). On pose $k' = |u'|$ et $k = |u|$. Soit $s = uv = u'v'$, on a par identification que pour tout $i < k'$, $s_i = u_i = u'_i$. On note alors t le mot de longueur $|k| - |k'|$ défini par $t_i = s_{i+k'}$; par construction $u = u't$, et le préfixe de taille $|t|$ de v' est égal à $|t|$. Enfin soit $n = |s|$, $\ell' = |v'|$ et $\ell = |v|$, on a par identification que pour tout $n - \ell \leq i < n$, $v_i = v'_{|t|+i}$, et donc $tv = v'$. Tout ceci se résume avantageusement par un schéma :



2. Montrez que si u, u' sont tous deux préfixes (resp. suffixes) d'un même mot v, alors u est préfixe (resp. suffixe) de u' ou u' est préfixe (resp. suffixe) de u.

On montre uniquement le cas préfixe. On a alors $v = u's' = us$ pour certains mots s' & s. Par le lemme de Levi, il existe t tel que $u't = u$ ou $ut = u'$, et donc u' est préfixe de u ou vice-versa.