TP 14 : Programmation dynamique

L'objectif de ce TP est d'appliquer les principes de la programmation dynamique, d'abord sur des exemples simples ou vus en cours (Exercice 1), puis sur un exemple classique dont l'analyse et l'expression demande un peu plus de doigté (Exercice 2). L'Exercice 3 est un problème ouvert (non guidé) où l'on peut aussi appliquer la programmation dynamique.

Exercice 1 – Exemples classiques

Langage libre (la correction en ligne sera en OCaml).

▶ **Question 1** Réimplémenter la fonction récursive naïve calculant la suite de Fibonacci.

▶ **Question 2** Appliquer une approche descendante de programmation dynamique pour calculer le $n$ième terme de la suite : on crée un tableau fib de taille $n+1$ de type `int option array` remplis initialement de `None`. On pourra compléter le code suivant :

OCaml
```
let fib_descendant n =
  let t = Array.make (n+1) None in
  (* [fib k] renvoie le kième terme de la suite de Fibonacci
     si [t.(k)] contient un [Some], on renvoie directement sa valeur *)
  let rec fib k = (* int -> int *)
    match t.(k) with
    | Some v -> v (* on avait déjà calculé [fib k] *)
    | None -> (* à compléter : attention à stocker le résultat dans t *)
  in fib n
```

▶ **Question 3** Déterminer sa complexité temporelle et spatiale. Pour la complexité spatiale, on comptera séparément la complexité spatiale sur la pile (la quantité de mémoire maximale stockée sur la pile d'appels récursifs).

▶ **Question 4** Appliquer une approche ascendante de programmation dynamique pour calculer le $n$ième terme de la suite de Fibonacci : on crée un tableau `t` de taille `n + 1`, on initialise les deux premières cases à 1, puis on calcule `t.(i)` dans l'ordre croissant de `i` jusqu'à `n`.

▶ **Question 5** Déterminer la complexité temporelle et spatiale de cette seconde version, avec la même séparation qu'à la Question 3.

On a déjà vu qu'il était possible de les calculer en gardant en mémoire seulement les deux dernières valeurs calculées de la suite.

▶ **Question 6** Reprendre le poly et vérifier que l'on a compris l'exemple de la découpe optimale du cours (Section VI.B).

Exercice 2 – Sous-séquence contiguë maximale

On considère un tableau `t` de taille `n` ($n \geq 1$), contenant des entiers (positifs ou négatifs), et on cherche à trouver le couple (i, ℓ) avec $0 < \ell \leq n$ $0 \leq i \leq n - \ell$, tel que la somme $t[i] + \cdots + t[i + \ell - 1]$ soit la plus grande possible.
Par exemple, si on considère le tableau t suivant :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Ici, la plus grande somme contiguë possible est $t[7] + t[8] + t[9] + t[10] = 43$, ce qui correspond au couple $(i, \ell) = (7, 4)$.

Résolution naïve par exploration exhaustive

▶ **Question 1** Combien y a-t-il de telles sommes différentes, en fonction de n ?

▶ **Question 2** En déduire un algorithme naïf qui résout le problème par exploration exhaustive. *On l'écrira en pseudocode ou en langage naturel, et on déterminera sommairement sa complexité.*

▶ **Question 3** Implémenter cette fonction en OCaml : `max_somme_contigue_exhaustive : int array -> int`. *Indication : on pourra commencer par écrire une fonction* `somme_sous_tableau : int array -> int -> int -> int` *telle que* `somme_sous_tableau t i l` *calcule la somme* $t[i] + \cdots + t[i+l-1]$.

▶ **Question 4** Quelle est sa complexité temporelle, en fonction de n ?

Même si cette approche exhaustive n'est pas hors de prix (c'est-à-dire exponentiel ou pire), on va quand même chercher à faire mieux !

Notion de sous-problème et récurrence

Ici, on va introduire un sous-problème différent, qui est celui de calculer

$$f(i) := \max_{i+1 \leq k \leq n} \sum_{j=i}^{k-1} t[j]$$

C'est-à-dire $f(i)$ est la plus grande somme contiguë démarrant à $t[i]$, pour $i \in [\![0, n-1]\!]$. Ici la connaissance d'une seule valeur de f (par exemple $f(0)$) ne sera pas suffisante pour déterminer la plus grande somme contiguë quelconque, mais il va suffire de *calculer un maximum* des valeurs prises par f pour l'obtenir.
Pour le tableau donné en exemple précédemment, on obtient les valeurs suivantes pour f. *Ce tableau est sans doute beaucoup plus simple à interpréter de droite à gauche.*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
$f(i)$	13	-3	-4	21	1	4	20	43	25	5	12	-5	-4	18	3	7

Tout l'intérêt d'avoir introduit cette valeur $f(i)$ est de pouvoir exprimer une relation de récurrence simple sur f.

$$f(n-1) = t[n-1]$$
$$\forall i < n-1, f(i) = t[i] + \begin{cases} f(i+1) & \text{si } f(i+1) > 0 \\ 0 & \text{sinon} \end{cases}$$

Cela peut se réécrire $f(i) = t[i] + \max(f(i+1), 0)$ si $i < n-1$ n'est pas le dernier indice.

Récurrence naïve

▶ **Question 5** Écrire une fonction récursive naïve qui calcule `f`, de signature `f : int array -> int -> int`, tel que l'appel à `f t i` calcule ce $f(i)$ définit plus haut.

▶ **Question 6** Définir un tableau `t` représentant celui donné en exemple dans la figure précédente, et l'utiliser pour tester votre fonction `f`. *Indication : on peut construire le tableau des* $f(i)$ *en utilisant* `Array.init (Array.length t) (fun i -> f t i)`.

On peut alors répondre au problème initial, en calculant le maximum des valeurs de $f(i)$.

▶ **Question 7** Écrire une fonction `max_array : 'a array -> 'a` qui prend un tableau `tab` non vide et renvoie sa plus grande valeur $\max_i \text{tab}[i]$.

▶ **Question 8** En déduire une fonction `max_somme_contigue_naive : int array -> int` qui prend un tableau `t` et calcule d'abord le tableau des valeurs de $f(i)$, pour $0 \leq i < n$, puis renvoie son maximum.

★ **Question 9** Déterminer sa complexité.

Résolution par mémoïsation

On peut adapter le code précédent de la fonction récursive `f` pour ajouter un cache de mémoïsation.
Le squelette de code que l'on peut proposer pour commencer est le suivant :

OCaml
```
let rec f t cache i =
  if Hashtbl.mem cache i then
    Hashtbl.find cache i
  else begin
    let valeur_fi =
      let n = Array.length t in
      (* ...
         calcul de f(i) selon la relation de récurrence : à remplir !
         ... *)
    in
    Hashtbl.add cache i valeur_fi;
    valeur_fi
  end
```

En plaçant le cache en dehors de la fonction, on permet à la fonction `f` de ne pas recalculer les valeurs.

▶ **Question 10** Compléter cette fonction.

▶ **Question 11** Écrire une fonction `max_somme_contigue_memoisee`

Remarque 1

Il est beaucoup plus élégant de cacher le cache dans une "clôture", mais en faisant cela, on va se heurter au fait qu'il faut un cache partagé entre les différentes valeurs de i, mais commun à un `t` donné.
L'astuce réside dans le fait de couper la fonction en insérant la création du cache après le paramètre `t` :

OCaml
```
let f t =
  let n = Array.length t in
  let cache = Hashtbl.create (n+1) in
  let rec f_memo i =
    if Hashtbl.mem cache i then
      Hashtbl.find cache i
    else begin
      let valeur_fi =
        (* ...
           calcul de f(i) selon la relation de récurrence : à remplir !
           ... *)
      in
      Hashtbl.add cache i valeur_fi;
      valeur_fi
    end
  in
  (fun i -> f_memo i)
```

★ **Question 12** Compléter cette fonction.

★ **Question 13** En déduire une fonction `max_somme_contigue_memoisee` et la tester.

Résolution par tabulation

Après une résolution par mémoïsation (donc descendante, les appels récursifs sur les instances de grandes tailles provoquent la résolution du problème sur des instances de plus petite taille), on cherche généralement à faire mieux en utilisant un tableau (ici, uni-dimensionnel) pour stocker les résultats, ce qui nécessite de savoir deux choses :

1. le nombre total de résultats à calculer (ici les différents $f(i)$),
2. l'ordre de remplissage à utiliser pour remplir ce tableau de valeurs, pour calculer les valeurs dans l'ordre de dépendances donné par la relation de récurrence.

▶ **Question 14** Déterminer les informations 1 et 2.

▶ **Question 15** En déduire une fonction non récursive `max_somme_contigue_tabulee`, et la tester.

★ **Question 16** À l'aide de la fonction suivante, comparer le temps d'exécution des trois versions de la fonction `max_somme_contigue` sur des entrées aléatoires. *On fera attention à ce que la somme calculée ne puisse pas dépasser la capacité des entiers OCaml, ce qui serait dommage...*

OCaml
```
let temps_execution fonction entree message =
  let t0 = Sys.time () in
  let sortie = fonction entree in
  let t1 = Sys.time () -. t0 in
  Printf.printf "Execution time (%s) : %fs\n" message t1;
  sortie
```

Reconstruction

Lorsqu'on effectue le calcul du maximum (des valeurs de $f(i)$) dans les fonctions précédentes, il est facile de garder dans une variable l'indice où il se produit. Cependant, en procédant ainsi, on ne peut pas en déduire facilement la longueur de la somme.
Pour ce faire, on va créer un nouveau tableau `l`, indiquant à l'indice i la longueur de la plus grande somme commençant par $t[i]$.
En effet, si on note $l(i)$ la plus grande valeur telle que $f(i) = \sum_{k=i}^{i+l(i)-1} t[k]$, alors on a $l(n-1) = 1$, et la relation de récurrence suivante :

$$\forall i < n-1, l(i) = \begin{cases} 1 + l(i+1) & \text{si } f(i+1) > 0 \\ 1 & \text{sinon} \end{cases}$$

Sur l'exemple précédent on va donc calculer les valeurs suivantes :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
$f(i)$	13	-3	-4	21	1	4	20	43	25	5	12	-5	-4	18	3	7
$l(i)$	1	1	9	8	7	6	5	4	3	2	1	1	4	3	2	1

Il n'y a alors plus qu'à lire le couple $(i, l(i))$ correspondant au maximum pour $f(i)$.

▶ **Question 17** Implémenter cela dans une fonction `max_somme_contigue : int array -> int * int * int` qui renvoie un triplet `s, i, l` tel que `s` soit la plus grande somme contiguë possible, commençant à l'indice `i` et de longueur `l`. *On pourra commencer par introduire une fonction* `argmax : 'a array -> int` *qui renvoie l'indice* `i` *(le plus grand possible) qui maximise les valeurs du tableau donné en argument.*

▶ **Question 18** Tester la sur le tableau `t` de l'exemple. On doit obtenir s = 43, i = 7 et l(i) = 4 ($s = t[7] + t[8] + t[9] + t[10]$).

Exercice 3

Je résume ici le problème 345 du projet Euler (`https://projecteuler.net/problem=345`).
Vous avez à disposition une matrice de taille $M \in \mathcal{M}_{n,n}(\mathbb{N})$. Déterminer la quantité suivante :

$$s(M) = \max_{\sigma \in \mathcal{S}_n} \sum_{k=0}^{n-1} m_{k,\sigma(k)}$$

Avec $M = (m_{i,j})_{i,j \in [\![0,n-1]\!]}$ et $\mathcal{S}_n$ l'ensemble des permutations de $[\![0, n-1]\!]$.
On pourra commencer par étudier la fonction suivante et essayer de la calculer efficacement :

$$f(M, X) = \max_{\sigma : [\![0,|X|-1]\!] \to X \text{ bijective}} \sum_{i=0}^{|X|-1} m_{i,\sigma(i)}$$

avec $X \subseteq [\![0, n-1]\!]$. *Ensuite, il suffit d'évaluer cette fonction en* $X = [\![0, n-1]\!]$.
Votre programme devrait résoudre le problème en moins d'une minute pour la matrice de taille $(15, 15)$ proposée. Le langage choisi est libre.