Pour finir le chapitre sur la programmation dynamique, on s'intéresse ici à plusieurs problèmes algorithmiques, que l'on peut résoudre par programmation dynamique.

> **Ce TP est volontairement très long, je vous encourage à allouer du temps pendant l'été pour le terminer. Vous pourrez m'écrire pour me montrer ce que vous aurez fait, n'hésitez pas !**

On travaillera en OCaml ou en C, depuis la machine virtuelle ClefAgreg2019 et Emacs, ou depuis `https://BetterOCaml.ml` et `https://www.onlinegdb.com/`.

Problème 1 : Sous-séquence contiguë maximale (en OCaml)

On considère un tableau `t` de taille `n` ($n \geqslant 1$), contenant des entiers (positifs ou négatifs), et on cherche à trouver le couple (i, ℓ) avec $0 \leqslant i \leqslant n-1-\ell$ et $0 \leqslant \ell < n-1$, tel que la somme

$$t[i] + \cdots + t[i+\ell]$$

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 $t[7] + t[8] + t[9] + t[10] = 43$.

Résolution naïve par exploration exhaustive

0. Combien y a-t-il de telle sommes différentes, en fonction de `n` ?
   - En déduire un algorithme naïf qui résout le problème par exploration exhaustive.
   - 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 $t[i] + \cdots + t[i+l]$.
   - Quelle est sa complexité temporelle, en fonction de `n` ?

Même si cette approche exhaustive n'est pas hors de prix (ie. 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

$$f(i) := \max_{i \leqslant k < n} \sum_{j=i}^{k} t[j]$$

 Sous licence MIT, par Lilian Besson, 2022

avec $f(i)$ qui représente la plus grande somme possible démarrant à $t[i]$.

Cependant, contrairement à certains problèmes déjà étudiés en cours, ici la connaissance d'une seule valeur de f (par exemple $f(0)$) ne sera pas suffisante pour déterminer la plus grande somme quelconque. Mais il va suffire de *calculer un maximum* des valeurs prises par f.

Pour le tableau donné en exemple précédemment, on obtient les valeurs suivantes pour f :

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
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 la relation de récurrence suivante :

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

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

Pour rappel, en OCaml, une fonction polymorphe `max : 'a -> 'a -> 'a` existe dans la librairie standard. Vous êtes libres de l'utiliser en TP, DS/DM, et aux concours.

1. Malgré tout, il est bon de savoir réécrire rapidement ce genre de petite fonction utilitaire ! Écrire une fonction `max` polymorphe, par vous même.
2. 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.
3. 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 $f(i)$.

4. Écrire une fonction `max_array : 'a array -> 'a` qui prend un tableau `tab` non vide et renvoie sa plus grande valeur $\max_i \text{tab}[i]$.
5. 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 $0 \leqslant i < n$, puis renvoie son maximum.

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 :

 Sous licence MIT, par Lilian Besson, 2022

```
let cache = Hashtbl.create 42 (* taille quelconque ici *);;

let rec f t 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 `max_somme_contigue_naive` de ne pas recalculer les valeurs mais on rend la fonction à usage unique : il faudrait vider le cache au début de `max_somme_contigue_naive`.

6. Compléter cette fonction et tester la.

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` :

```
let f t =
  let n = Array.length t in
  (* taille correcte, ici *)
  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 =
        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
  in
  (fun i -> f_memo i) ;;
```

 Sous licence MIT, par Lilian Besson, 2022

7. Compléter cette fonction et tester la.
8. En déduire une fonction non récursive `max_somme_contigue_memoisee` et tester la.

Résolution par tabulation

Après une résolution par mémoïsation, on cherche généralement à faire mieux en utilisant un tableau (ici, uni-dimensionnel) pour stocker les résultats, ce qui nécessite de savoir deux choses :

— a) le nombre total de résultats à calculer (ici les différents $f(i)$),

— b) 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.

9. Déterminer les informations a) et b).
10. En déduire une fonction non récursive `max_somme_contigue_tabulee`, et tester la.

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 $f(i) = \sum_{k=i}^{i+l(i)} t[k]$, alors on a $l(n-1) = 0$, et la relation de récurrence suivante :

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

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

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
f(i)	13	-3	-4	21	1	4	20	43	25	5	12	-5	-4	18	3	7
l(i)	0	0	8	7	6	5	4	3	2	1	0	0	3	2	1	0

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

11. 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.*

 Sous licence MIT, par Lilian Besson, 2022

12. Tester la sur le tableau `t` de l'exemple. On doit obtenir `s = 43`, `i = 7` et `l(i) = 3` ($s = t[7] + t[8] + t[9] + t[10]$).

Problème 2 : Chemins monotones (en C)

On considère ici le problème de la plus grande somme en partant du sommet d'un triangle de nombres et descendant soit en bas à gauche ou en bas à droite d'un cran.

Voici un exemple d'instance, pour laquelle la somme la plus grande vaut 1074 (on ne demande **pas** de le vérifier !) :

```
            75
           95 64
          17 47 82
         18 35 87 10
        20 04 82 47 65
       19 01 23 75 03 34
      88 02 77 73 07 63 67
     99 65 04 28 06 16 70 92
    41 41 26 56 83 40 80 70 33
   41 48 72 33 47 32 37 16 94 29
  53 71 44 65 25 43 91 52 97 51 14
 70 11 33 28 77 73 17 78 39 68 17 57
91 71 52 38 17 14 91 43 58 50 27 29 48
63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

Implémentation en C

Tout d'abord, on va représenter le problème sous la forme d'un tableau de tableaux d'entiers, un `int[][]`, ligne par ligne, tel que la i-ème ligne ait `i+1` éléments utiles, et le reste remplis de zéros. On aura aussi besoin de stocker le nombre de ligne, car en C les tableaux ne connaissent pas leurs longueurs...

Par exemple, le sous problème issu de l'instance précédente réduite à ces 4 premières lignes sera représenté ainsi :

```
int n = 4;
int tab[4][4] = {
  {75, 0, 0, 0},
  {95, 64, 0, 0},
  {17, 47, 82, 0},
  {18, 35, 87, 10},
};
```

Les deux successeurs possibles de `t[i][j]` sont donc `t[i+1][j]` (en bas à gauche) et `t[i+1][j+1]` (en bas à droite).

 Sous licence MIT, par Lilian Besson, 2022

Notion de sous-problèmes et relation de récurrence Ici on va considérer $f(i,j)$ la plus grande somme possible, à partir de la ième ligne et jème colonne.

1. Quelle est la valeur $f(i,j)$ que l'on doit calculer pour répondre au problème initial ?
2. Exprimer $f(n-1,j)$ (pour la dernière ligne, d'indice $i = n-1$) en fonction d'une valeur du tableau `t`.

Pour les autres éléments, il faut considérer les deux possibilités pour le premier déplacement en partant de (i,j) : en bas à gauche (vers $(i+1,j)$) ou en bas à droite (ver $(i+1,j+1)$). Il faut prendre le choix qui aboutit à la plus grande somme, et donc on en déduit la relation de récurrence suivante :

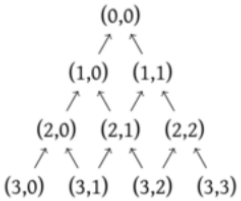
$$\forall i [\![0, n-2]\!], \forall j \in [\![0, n-1]\!], f(i,j) = t[i][j] + \max\big(f(i+1,j), f(i+1,j+1)\big).$$

Résolution naïve de la récurrence

3. Implémenter une fonction `int max(int x, int y)`. Contrairement à OCaml, le langage C ne propose pas de telle fonction dans sa librairie standard. Pas grave, c'est rapide à redéfinir nous-même.
4. Implémenter une fonction naïve (récursive) `int maxpath_naive(int n, int tab[][], int i, int j)` qui calcule $f(i,j)$ selon cette relation de récurrence.
5. Dans le `main`, définir l'instance d'exemple de taille $n = 4$ lignes, et vérifier que la somme la plus grande possible est $308 = 75 + 64 + 82 + 87$. A quel chemin (suite de "bas gauche" ou "bas droite") cette somme correspond-elle ?
6. Quelle est la complexité temporelle de votre fonction `maxpath_naive` appelée en `(i,j) = (0,0)`, en fonction de `n` ? On pourra introduire C_n le nombre d'appels récursifs sur un triangle de taille n, et chercher une relation de récurrence vérifiée par C_n. La complexité temporelle totale est un $\Theta(C_n)$ pour une instance quelconque de taille `n`.

On va donc chercher à être bien plus efficace, en utilisant une matrice pour stocker les résultats de $f(i,j)$ et ne jamais devoir recalculer plusieurs fois le même résultat.

Tabulation On peut représenter le graphe de dépendances des sous-problèmes :



On remarque facilement qu'on peut obtenir un tri topologique en traitant ce graphe de dépendances du bas vers le haut. Ici, on peut donc tabuler dans le même format triangulaire que `t` (en fait on fera dans une matrice carrée, par simplicité), et de remplir les valeurs `f[i][j]` de cette matrice par `i` décroissant du bas vers le haut) et `j` croissant (ou décroissant, peu importe).

7. En déduire une version tabulée, non récursive : `int maxpath_tabulee(int n, int tab[][])` qui calcule $f(0,0)$ selon sa relation de récurrence, dans l'ordre expliqué.

 Sous licence MIT, par Lilian Besson, 2022

8. Tester la sur la même instance et vérifier qu'elle donne le même résultat que la première fonction.
9. Quelle est sa complexité en fonction de n ? Est-ce meilleur que l'approche récursive naïve ?

Expériences numériques

On aimerait vérifier empiriquement la complexité exponentielle que l'on a calculé pour la fonction `maxpath_naive`.

On rappelle que l'on peut exécuter un binaire dans le terminal en le préfixant de `time` pour obtenir une mesure de temps d'exécution :

```
$ time ./TP25_correction_pb2.exe 4
Instance de taille n = 4 :
75
95 64
17 47 82
18 35 87 10

Calcul récursif naïf de la plus grande somme = 308.
Calcul tabulé de la plus grande somme = 308.

[...]

real 12,023     user 12,012     sys 0,009       pcpu 99,99
```

Recopier la fonction suivante :

```
// Génération pseudo-aléatoire d'un entier tiré dans [upper, lower], bornes incluses
int rand_range(int lower, int upper) {
    return lower + (rand() % (upper - lower + 1));
}
```

et initialiser le générateur de nombres pseudo-aléatoires comme cela dans votre `main` (ce qui demande d'importer `time.h` en haut du fichier) : `srand(time(NULL));`.

10. Écrire une fonction `void benchmark_naive(int n)`
    - qui commence par allouer un tableau `int tab[n][n]` (sans faire de `malloc`), à le remplir de zéros partout, puis de valeurs aléatoires (par exemple `rand_range(-100, 100)`) sous forme d'un triangle de taille n comme dans les exemples précédents,
    - puis qui appelle `maxpath_naive(n, tab, 0, 0)`.
11. Dans le `int main(int argc, char* argv[])`, terminer le `main` en utilisant `int n = atoi(argv[1])` pour lire la taille `n` depuis l'argument donné en ligne de commande, et appeler une fois `benchmark_naive(n)`.
12. Expérimenter un peu en testant des valeurs de `n` entre 10 et 30, en augmentant `n` de 1 en 1. Il faut regarder le temps total qu'a pris le binaire à s'exécuter, et normalement on observe qu'augmenter `n` de 1 multiplie quasiment le temps d'exécution par 2, ce qui est cohérent avec le résultat d'une complexité temporelle en $\Theta(2^n)$ pour `maxpath_naive`.

 Sous licence MIT, par Lilian Besson, 2022

Avec une granularité d'une milli-secondes, dans la correction je propose un "benchmark" qui donne les résultats suivants :

```
Premier benchmark : la méthode naïve devrait être en temps Theta(2^n) :
Calcul par la méthode naïve pour une instance de taille n = 20.
        Temps = 8 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 21.
        Temps = 17 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 22.
        Temps = 29 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 23.
        Temps = 53 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 24.
        Temps = 105 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 25.
        Temps = 210 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 26.
        Temps = 423 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 27.
        Temps = 863 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 28.
        Temps = 1687 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 29.
        Temps = 3372 milli-seconde(s)...
Calcul par la méthode naïve pour une instance de taille n = 30.
        Temps = 6835 milli-seconde(s)...
```

Bonus : implémentation en OCaml

Faire de même une implémentation en OCaml :

— réfléchir à la représentation possible d'une instance. On devrait pouvoir faire mieux qu'une matrice carrée dans laquelle les cases vides sont égales à 0.

— implémenter une fonction récursive naïve `maxpath_naive : int array array -> int -> int -> int`,

— et une version tabulée non récursive.

— Bonus : ajouter une fonction mémoïsée avec un cache (une table de hachage en utilisant le module `Hashtbl`).

 Sous licence MIT, par Lilian Besson, 2022

Problème 3 : Produit de matrices

Implémenter dans le langage de votre choix l'algorithme de programmation dynamique qui permet de résoudre efficacement le problème de trouver un ordre de parenthésage optimal pour le calcul d'un produit de n matrices rectangles $P := A_1 \times A_2 \times \cdots \times A_n$, dont on fournit les dimensions : A_i est de dimension (n_{i-1}, n_i), pour $1 \leqslant i \leqslant n$ (ie. $A_i \in \mathcal{M}_{n_{i-1},n_i}(\mathcal{A})$ à valeurs dans un anneau $\mathcal{A}$). On fournit leurs dimensions comme la donnée d'un tableau de $n+1$ entiers $[n_i]$.

L'algorithme a été étudié dans le cours.

En OCaml

La signature de la fonction peut être la suivante : `nb_min_multiplications : int array -> int`, qui calcule juste le nombre minimal de multiplications (dans l'anneau $\mathcal{A}$ de valeurs de matrices, pas de matrices) nécessaires pour calculer ce produit P.

On peut ensuite raffiner l'implémentation pour aussi fournir les informations nécessaires pour savoir où découper le produit $A_i \times \cdots \times A_j$ pour minimiser le nombre total de multiplications (dans l'anneau $\mathcal{A}$), et écrire par exemple une fonction de signature `optimisation_multiplications : int array -> int array array` qui renvoie un tableau bidimensionnel `T`, celui noté $T[i,j]$ dans le cours. $T[i,j]$ fournit le (plus petit des) $k \in \{i, \dots, j-1\}$ qui minimise le min qui définit $m^*[i,j] = \min_k(m^*[i,k] + n_{i-1} * n_k * n_j + m^*[k+1,j])$.

En C

La signature de la fonction peut être la suivante : `int nb_min_multiplications(int n, int d[])` qui suppose que le tableau des dimensions, `d`, est de taille `= n+1`.

On peut aussi raffiner l'implémentation en écrivant une fonction `int** optimisation_multiplications(int n, int d[])` qui renvoie ce tableau bidimensionnel `T`.

Problème 4 : Distance d'édition (de Levenshtein)

Implémenter dans le langage de votre choix l'algorithme de programmation dynamique qui permet de calculer efficacement la distance d'édition entre deux chaînes de caractères `A` et `B`.

L'algorithme a été étudié dans le cours.

En OCaml

La signature de la fonction peut être la suivante : `distance_edition : string -> string -> int`, qui calcule la distance d'édition entre deux chaînes de caractères (des `string` en OCaml).

Pour rappel, si `a : string` est une `string`, on peut obtenir sa longueur par `String.length a`, et accéder à sa i-ème valeur (un `char`) avec la syntaxe `a.[i]` (attention ce n'est **pas** `a.(i)` qui est réservé aux `array`).

 Sous licence MIT, par Lilian Besson, 2022

En C

La signature de la fonction peut être la suivante : `int distance_edition(int taille_a, char a[], int taille_b, char b[])`. Pour simplifier, on peut fournir les dimensions des deux chaînes `a` et `b` (des `char[]`), qui peuvent aussi se calculer (une fois, et en stockant le résultat) en temps linéaire avec `int strlen(char str[])` de la librairie `string.h` qu'il faudra donc inclure en haut de votre fichier.

 Sous licence MIT, par Lilian Besson, 2022