

TP 4 : Programmation dynamique

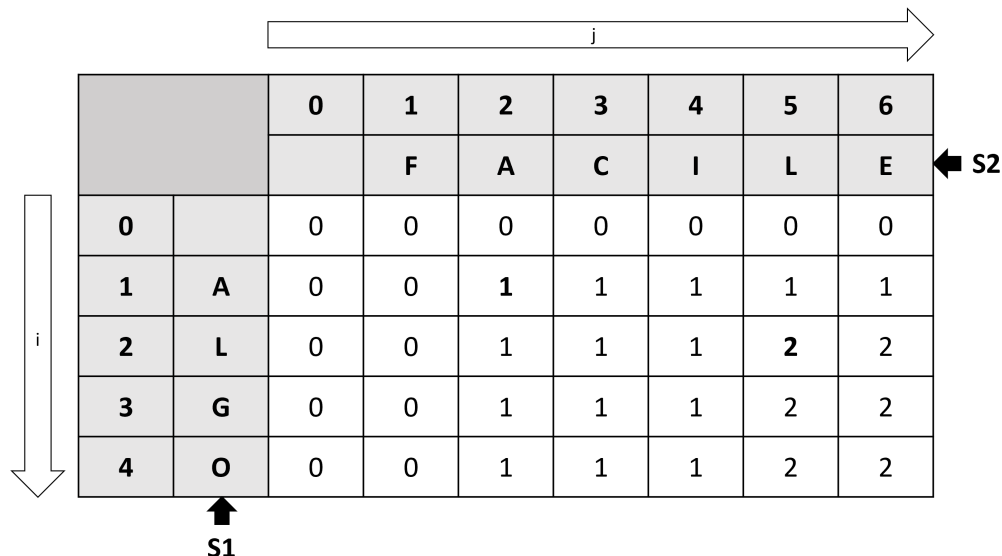
L'objectif de ce TP est d'explorer le concept de programmation dynamique, une technique qui permet de résoudre de façon exacte certains problèmes de grande complexité (NP-complets) en temps polynomial ou pseudo-polynomial. Cette méthode exploite notamment la propriété de sous-structure optimale des problèmes sur lesquels elle s'applique, qui garantit qu'une solution optimale du problème formulé avec n variables peut s'obtenir trivialement à partir de la solution optimale du même problème formulé avec $n - 1$ variables. Le principe général de la programmation dynamique consiste donc à construire incrémentalement les solutions en enregistrant tous les résultats intermédiaires dans une structure de données (en général un tableau ou une table de hachage) pour pouvoir les réutiliser au fur et à mesure pour la construction d'autres solutions.

Dans ce TP, on étudie notamment la résolution de 3 problèmes populaires par programmation dynamique : la recherche de plus longues sous-séquences communes entre deux chaînes de caractères, le problème du sac à dos et le calcul de dates au plus tôt / dates au plus tard.

PLUS LONGUE SOUS-SÉQUENCE COMMUNE

Ce problème consiste à trouver la plus longue séquence de caractères présente dans deux chaînes, sans que ceux-ci soient forcément consécutifs. Par exemple, la plus longue sous-séquence commune pour les chaînes de caractères "algo" et "facile" est de taille 2 et correspond à "al".

Résoudre ce problème par programmation dynamique consiste à construire le tableau suivant :



		j							
		0	1	2	3	4	5	6	
			F	A	C	I	L	E	
i	0	0	0	0	0	0	0	0	
1	A	0	0	1	1	1	1	1	
2	L	0	0	1	1	1	2	2	
3	G	0	0	1	1	1	2	2	
4	O	0	0	1	1	1	2	2	

À part pour la première ligne et pour la première colonne, qui contiennent la valeur triviale à retourner si aucune séquence n'est détectée (taille 0), la procédure pour calculer les valeurs intermédiaires est la suivante. Pour chaque cellule d'indice i, j :

$$tab[i][j] = \begin{cases} tab[i-1][j-1] + 1 & \text{si } S1[i] = S2[j], \\ \max(tab[i-1][j], tab[i][j-1]) & \text{sinon} \end{cases}$$

Le début de code ci-dessous est proposé pour la construction du tableau :

```

1 let tab_sous_sequence s1 s2 =
2   let taille_s1 = Array.length s1 in
3   let taille_s2 = Array.length s2 in
4   let tab_dynamique = Array.init (taille_s1 + 1) (function i -> Array.make (
   taille_s2 + 1) 0) in
5
6   let rec tab_sous_sequence_aux i j = ---
7   in tab_sous_sequence_aux 1 1 ;
8
9   tab_dynamique ;;

```

Question 1. Proposez une implémentation pour la fonction récursive terminale *tab_sous_sequence_aux* qui parcourt juste le tableau et calcule les valeurs des cellules avec les formules précédentes.

Question 2. Dans le fichier *tp4_code.ml* fourni avec ce TP, récupérez le code des fonctions *string_to_array* et *sous_sequence* et testez votre implémentation en vérifiant que la plus longue sous-séquence commune à "algo" et "facile" est bien "al".

ALGORITHME DU SAC À DOS

Le problème du sac à dos consiste à remplir de façon optimale un sac possédant une contenance maximale W avec des objets. Chaque objet possède un poids et une utilité et l'objectif final est d'obtenir la composition du sac qui possède la plus grande utilité possible sans que son poids total ne dépasse W .

Ce problème possède une sous-structure optimale puisqu'il est trivial de construire une solution avec n objets si on connaît une solution optimale avec $n - 1$ objets. Il suffit alors d'ajouter le dernier objet si son poids le permet, et de ne pas l'ajouter sinon. Pour résoudre le problème du sac à dos par programmation dynamique, on peut utiliser un tableau similaire à celui de l'exercice précédent :

			Capacité							
Objet	Poids	Utilité	0	1	2	3	4	5	6	7
0			0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
2	3	5	0	1	1	5	6	6	6	6
3	5	2	0	1	1	5	6	6	6	6
4	3	1	0	1	1	5	6	6	6	7
5	1	3	0	3	4	5	8	9	9	9

À part pour les valeurs par défaut quand i et/ou j valent 0, les valeurs des cellules du tableau sont calculées à partir des formules suivantes :

$$u_{\max}[i][c] = \begin{cases} u_{\max}[i-1][c] & \text{si } c < \text{Poids}[i], \\ \max(u_{\max}[i-1][c], u_{\max}[i-1][c - \text{Poids}[i]] + \text{Utilité}[i]) & \text{sinon} \end{cases}$$

Question 3. En vous inspirant du code de *tab_sous_sequence*, programmez une fonction *tab_sac_a_dos* qui calcule le contenu du tableau pour un problème du sac à dos donné. Les variables suivantes seront passées en paramètres : capacité max du sac, tableau des poids, tableau des valeurs.

Question 4. Dans le fichier *tp4_code.ml* fourni avec ce TP, récupérez le code de la fonction *sac_a_dos* et testez votre implémentation avec les mêmes paramètres que ceux de l'exemple ci-dessus ($W = 7$).

MEMOÏSATION

Quand les valeurs intermédiaires sont enregistrées dans une table de hachage plutôt que dans un tableau et que seules les opérations intermédiaires utiles sont faites, on parle de **mémoïsation**. La fonction suivante permet de mémoïser n'importe quelle fonction passée en paramètre :

```
1 let memoiser fonction =
2   let hashtable = Hashtbl.create 123 in
3   let rec fonction_aux x param =
4     try Hashtbl.find hashtable x
5     with Not_found ->
6       let resultat = fonction fonction_aux x param
7       in Hashtbl.add hashtable x resultat ;
8       resultat
9   in (fonction_aux , hashtable) ;;
```

Question 5. Exécutez cette fonction dans l'interpréteur OCaml. Quel est son type ? A quoi doit ressembler la fonction en paramètre ?

Question 6. Mémoïsez la fonction *fibonacci* suivante, puis calculez la valeur de la suite de Fibonacci pour $n = 50$. Si vous le souhaitez, vous pouvez retourner sur l'énoncé du TP 2 pour récupérer le code du calcul naïf de fibonacci et comparer avec cette nouvelle implémentation.

```
1 let rec fibo fibo_aux x _ =
2   if x < 2 then x
3   else (fibo_aux (x-1) ()) + (fibo_aux (x-2) ()) ;;
```

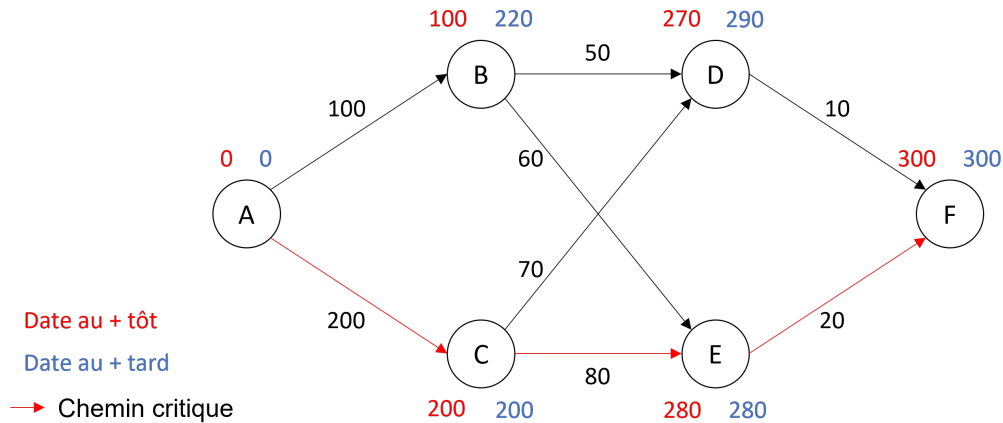
POUR ALLER PLUS LOIN : DATES AU PLUS TÔT / AU PLUS TARD

Le calcul des dates au plus tôt et au plus tard dans un diagramme de Pert fait partie des problèmes à sous-structure optimale qui peuvent être résolus par programmation dynamique. En l'occurrence, nous allons ici le résoudre par mémoïsation (pour varier les plaisirs).

Pour rappel, un diagramme de Pert est un graphe acyclique dirigé (DAG en anglais) qui permet de calculer les dates de commencement au plus tôt et au plus tard de différentes tâches, par exemple pour planifier un projet. Dans un diagramme de Pert :

- chaque sommet représente un état d'avancement du projet ;
- chaque arête entre un sommet A et un sommet B représente une tâche à accomplir pour faire passer le projet de l'état A à l'état B ;
- chaque arête possède un poids équivalent au temps nécessaire pour accomplir la tâche associée.

Un diagramme de Pert permet également d'identifier les chemins critiques, c'est-à-dire les séquences de tâches où les dates au plus tôt et au plus tard sont égales et où toute fluctuation du temps mis pour accomplir une des tâches aurait un impact positif ou négatif sur toutes les suivantes. Voici un exemple de diagramme de Pert :



```

1 let sommets = [| 'A'; 'B'; 'C'; 'D'; 'E'; 'F' |] ;;
2 let arcs = [| ('A', 'B', 100); ('A', 'C', 200); ('B', 'D', 50); ('B', 'E', 60);
  ('C', 'D', 70);
3 ('C', 'E', 80); ('D', 'F', 10); ('E', 'F', 20) |] ;;
4 let source = 'A' ;;
5 let terminal = 'F' ;;

```

La date au plus tôt de la première tâche est toujours égale à 0. La date au plus tard de la dernière tâche est toujours égale à sa date au plus tôt. Sinon :

- pour obtenir la **date au plus tôt** d'une tâche, 1) on regarde toutes ses arêtes **entrantes**, 2) on **additionne** leur poids avec la **date au plus tôt** de leur **source**, et 3) on garde uniquement la valeur **maximale** ;
- pour obtenir la **date au plus tard** d'une tâche, 1) on regarde toutes ses arêtes **sortantes**, 2) on **soustrait** leur poids avec la **date au plus tard** de leur **destination**, et 3) on garde uniquement la valeur **minimale**.

Question 7. Complétez la fonction *au_plus_tot* suivante, puis programmez la fonction *au_plus_tard*. Pour finir, mémorisez ces deux fonctions à l'aide de *memoiser*. *NB: une fonction print_ht_pert se trouve dans le fichier tp4_code pour pouvoir vérifier le contenu de vos tables de hachage.*

```

1 let rec plus_tot plus_tot_aux sommet (arcs, arcs_restants, sommet_source,
  duree_default, duree_max) =
2   if sommet = sommet_source then ---
3   else ( match arcs_restants with
4         | [] -> ---
5         | (source, destination, duree)::reste ->
6           if destination <> sommet
7           then --- sommet (arcs, ---, sommet_source, duree_default, ---)
8           else --- sommet (arcs, ---, sommet_source, duree_default,
9             (max duree_max ((--- source (arcs, ---, sommet_source,
              duree_default, duree_default)) --- ---))) ;

```

Question 8. (Pour aller plus loin). Programmez une fonction qui prend en entrée les tables de hachage des dates au plus tôt et au plus tard et qui affiche les chemins critiques.