TP15: OCaml impératif
Programmation dynamique

MP2I Lycée Pierre de Fermat

Tableaux, boucles for

On rappelle les opérations principales permettant de manipuler un tableau en OCaml:

- `Array.make n x` crée un tableau de `n` cases, toutes de valeur initiale `x`
- `t.(i)` donne la valeur de la case `i` d'un tableau `t`
- `t.(i) <- y` stocke la valeur de `y` dans la case `i` de `t`. Cette expression est de type `unit`
- `Array.length t` donne la longueur du tableau `t`

Boucle for

La syntaxe OCaml pour une boucle for est comme suit:

```
for INDICE = DEBUT to FIN do
  instruction 1;
  instruction 2;
  ...
  instruction k
done
```

avec `INDICE` un nom de variable (l'indice de boucle), et DEBUT et FIN les bornes incluses de la boucle. Une boucle for est une expression OCaml, et son type est `unit`.

Par exemple, la fonction suivante modifie un tableau d'entiers pour doubler le contenu de toutes ses cases:

```
let doubler (t: int array) : unit =
  let n = Array.length t in
  for i = 1 to n-1 do
    t.(i) <- t.(i) * 2
  done

let t = [| 2; 1; 3; 5 |] ;;
doubler t ;; (* t contient maintenant [|4; 2; 6; 10 |] *)
```

Q1. Écrire une fonction `array_range : int -> int array` telle que `array_range n` est un tableau de longueur n contenant $0, 1, \ldots, n-1$.

Références

On s'intéresse à l'écriture d'une fonction `somme: int array -> int` qui calcule la somme des éléments d'un tableau d'entiers. On souhaiterait pouvoir écrire:

```
let somme (t: int array) : int =
  let res = 0 in
  let n = Array.length t in
  for i = 0 to n-1 do
    (* modifier res en lui assignant res + t.(i) *)
  done;
  res
```

Cependant, en OCaml, les *variables* ne sont pas mutables, on ne peut pas "modifier" res pendant la boucle. Comme les tableaux, eux, sont mutables, on pourrait utiliser un tableau de taille 1, et stocker la somme dans son unique case:

```
let somme (t: int array) : int =
  let res = [|0|] in
  let n = Array.length t in
  for i = 0 to n-1 do
    res.(0) <- res.(0) + t.(i)
  done;
  res.(0)
```

Q2. Recopier la fonction ci-dessus, et vérifier qu'elle fonctionne correctement.

En C, nous avons vu que les tableaux et les pointeurs sont des objets très proches. En OCaml, il existe un équivalent des pointeurs, appelé les **références**, que l'on peut voir comme des tableaux d'une seule case. Pour les manipuler:

- `let x = ref a` assigne à `x` un pointeur vers une case mémoire contenant la valeur de `a`. Si `a` est de type `T`, alors `x` est de type `T ref`. On dit que `x` est une référence. C'est l'équivalent d'avoir écrit `T* x = &a` en C.
- Si `x` est une référence de type `T ref`, alors `!x` (se lit "déref x" ou "bang x") donne la valeur stockée dans `x`, de type `T`. C'est l'équivalent d'avoir écrit `*x` en C.
- Si `x` est une référence de type `T ref` et `a` une expression de type `T`, alors `x := a` **modifie** la valeur pointée par `x` en la valeur de `a`. L'expression `x := a` est une expression, de type `unit`.

Par exemple, le code suivant calcule la factorielle d'un entier à l'aide d'une référence et d'une boucle:

```
let fact (n: int) : int =
  let res = ref 1 in
  for i = 2 to n do
    res := !res * i
  done;
  !res
```

Les références s'utilisent donc comme les variables classiques du C, ce sont des cases mémoires.

Q3. Modifier le code de la fonction `somme`, afin qu'elle utilise une référence pour stocker le résultat.

Q4. Écrire une fonction `argmax: int array -> int` telle que `argmax t` donne l'indice du plus grand élément de `t`.

Q5. Implémenter un tri par sélection sur les tableaux (en sélectionnant systématiquement le **maximum** des éléments non traités et en le positionnant à la bonne position).

Application: vente de barre de métal

Vous êtes en possession d'une barre de métal de N centimètres. Vous voulez la vendre, et pour cela vous décidez de la découper et de vendre les tronçons. Cependant, le fonctionnement du marché est particulier: le coût d'un tronçon n'est pas forcément proportionnel à sa longueur, ni même croissant en fonction de sa longueur, et il est impossible de vendre des tronçons d'une longueur strictement supérieure à un seuil $K \in \mathbb{N}$. Cependant, vous disposez d'une grande table vous donnant le prix auquel vous pouvez vendre les différents tronçons.

Par exemple, si le prix de la barre de métal est comme suit:

Longueur (en cm)	0	1	2	3	4	5	6	7	8
Prix (en euros)	0	1	2	4	1	7	9	11	10

On suppose qu'un tronçon de longueur 0 se vendra toujours pour un prix 0.

Si notre barre fait $N = 8$ centimètres, on peut la vendre intacte pour 10 euros, ou bien la couper en un tronçon de 6 et deux tronçons de 1 pour 11 euros.

Modélisons ce problème et essayons de le résoudre en OCaml. On pourra représenter une barre de métal par un entier n, et le découpage d'une barre en tronçons par une liste $[x_1; \ldots; x_p]$ telle que $x_1 + \cdots + x_p = n$. Enfin, la table des prix est modélisé par un tableau C de taille $K + 1$, tel que pour $i \in [\![0, K]\!]$, $C[i]$ donne le prix de vente d'un tronçon de longueur i. Un tronçon de longueur strictement supérieure à K ne peut pas être vendu.

Glouton On propose tout d'abord une méthode gloutonne, qui consiste à couper au début de la barre un tronçon le plus cher possible, puis à réitérer le procédé avec le reste.

Q6. Écrire une fonction récursive `glouton_prix (c: int tab) (n:int) : int list` qui calcule le découpage d'une barre de `n` centimètres selon cette méthode, sachant que le tableau `c` donne le prix d'une longueur donnée de barre de métal. Attention, `n` peut a priori être plus grand ou plus petit que la longueur de `c`.

Q7. Trouver un exemple sur lequel cet algorithme n'est pas optimal.

On propose maintenant d'enlever à la barre le tronçon le plus rentable possible à chaque fois.

Q8. Implémenter cette méthode avec une fonction `glouton_ratio`

Q9. Trouver un exemple sur lequel cet algorithme n'est pas optimal.

Programmation dynamique On considère C un tableau de prix (de taille $K+1$), et n une longueur de barre. Pour $i \in [\![0, n]\!]$, on note $D(i)$ le meilleur prix auquel une barre de longueur i peut être vendue en la découpant. On a les trois formules suivantes:

- $D(0) = 0$
- $D(i) = \min_{k=1}^{K}(C[k] + D(i-k))$ si $i \geq K$
- $D(i) = \min_{k=1}^{i}(C[k] + D(i-k))$ sinon

Les deux dernières relations expriment que l'on essaie toutes les longueurs possibles pour le premier tronçon.

Utilisons ces formules pour implémenter un algorithme de programmation dynamique de haut-en-bas donnant une solution optimale au problème.

On rappelle que le principe de cette méthode est d'utiliser une fonction récursive "intelligente", qui, lorsqu'on l'appelle sur un argument, regarde dans un **tableau de mémoïsation** si le résultat a déjà été calculé plus tôt. Si l'on applique cette méthode au problème du découpage de barre, on obtient le schéma suivant:

```
1  (* Valeur optimale pour découper une barre de longueur n, selon
2     la fonction de prix f. *)
3  let decoupe_prog_dyn (c: int array ) (n: int) : int =
4    let mem = Array.make (n+1) (-1) in
5    (*
6       Initialisation de mem
7    *)
8
9    (* Renvoie le meilleur prix possible pour une barre de taille i, et
10      stocke le résultat dans la case mem.(i) *)
11   let rec calculer_decoupe (i: int) : int =
12     (* vérifier si la case a déjà été remplie ou non *)
13     if mem.(i) = -1 then begin
14       (* calculer récursivement le prix, et le stocker dans mem.(i) *)
15     end;
16     mem.(i)
17   in calculer_decoupe n
```

Q10. Compléter le code précédent pour implémenter la programmation dynamique sur le découpage de barre.

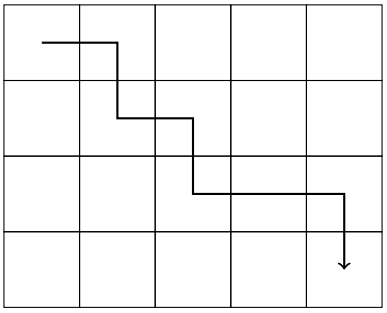
Q11. Modifier le code pour pouvoir calculer la liste des tronçons à vendre plutôt que le prix total atteint.

Nettoyage de grille

Cette partie est à faire en C.

Le jardin Rectangulo est un jardin rectangulaire, comme son nom l'indique, ressemblant à une grille de taille $n \times m$. Chaque matin, pour aller au travail, vous entrez dans le parc Rectangulo par la case en haut à gauche de cette grille, de position $(0, 0)$, et vous sortez par la case en bas à droite, de position $(n-1, m-1)$.

Pour cela, vous pouvez vous déplacer d'une case à la fois, soit vers la droite, soit vers le bas. Vous ne pouvez ni vous déplacer en diagonale, ni faire de détour vous ralentissant. Par exemple:



Suite à une coupe budgétaire, la ville ne peut plus payer le nettoyage régulier du parc. Sur chaque case de la grille se trouve un certain nombre de déchets. Vous voulez ramasser le plus de déchets possible en allant au travail, pour garder le parc propre.

On se donne donc un tableau 2D G tel que $G[i][j]$ est la quantité de déchets sur la case (i, j) de la grille. Votre but est de trouver le chemin permettant de **ramasser le plus de déchets**.

Parsing

On stocke les instances du problème dans des fichiers textes, contenant:

- Sur la première ligne, deux entiers n, m indiquant les dimensions de la grille
- Sur les n lignes suivantes, m entiers séparés par des espaces indiquant la quantité de déchets sur chaque case.

Trois exemples de tailles variables vous sont donnés dans l'archive du TP sur Cahier de Prépa:

- Une petite grille de taille 4×5
- Une grille moyenne de taille 23×34, dont la valeur optimale est 1189 et dont une version se trouve en ligne sur perso.ens-lyon.fr/guillaume.rousseau/mp2i/ramasse_miette/ [1]
- Une grille de taille 400×750 dont la valeur optimale est 25918

Q12. Proposer un type struct simple pour stocker les grilles. On appellera le type créé `grid_t`.

Q13. Écrire une fonction `grid_t* lire_grille (char* filename)` qui lit dans un fichier les données d'une grille selon le format précédent, et renvoie une structure contenant ces données

[1] Ne passez pas plus de 3 minutes à faire joujou SVP !

Pour représenter un chemin, on propose d'utiliser des chaines de caractères. Un `'D'` (comme Down) signifiera que l'on va vers le bas, et un `'R'` (comme Right) que l'on va vers la droite. Vous pouvez rentrer ces chaînes sur la page internet donnée plus haut pour visualiser vos chemins et vérifier vos résultats.

Q14. Écrire une fonction `int valeur(grid_t* g, char* chemin)` calculant la valeur d'un chemin dans une grille. Cette fonction renverra -1 et affichera un message d'avertissement si le chemin est invalide, i.e. s'il ne contient pas que des `'D'` et des `'R'`, s'il sort de la grille, ou s'il n'atteint pas la case en bas à droite.

Q15. Écrire une fonction `char* chemin_aleatoire(grid_t* g)` qui génère un chemin aléatoire entre $(0,0)$ et $(n-1,m-1)$. On ne demande pas à ce que la fonction choisisse chaque chemin avec une probabilité uniforme.

Q16. Lancez la fonction précédente plusieurs fois sur un même exemple, pour vérifier que vous obtenez des chemins valides, et distincts.

Q17. Combien y a-t-il de chemins distincts entre $(0,0)$ et $(n-1,m-1)$? Quelle serait la complexité de l'algorithme exhaustif consistant à tester tous les chemins ?

Glouton

Un algorithme glouton naturel pour ce problème est de construire un chemin petit à petit, en partant de la case de départ et en allant à chaque fois sur la case contenant le plus d'ordures parmi les deux cases adjacentes.

Q18. Écrire une fonction `bool choix_glouton(grid_t* g, int i, int j)` qui renvoie true si à partir de la case (i,j), l'algorithme glouton choisit d'aller sur la case à droite, et false s'il choisit la case en bas. Cette fonction prendra en compte les cas où la case (i,j) se trouve sur un bord de la grille.

Q19. Écrire une fonction `char* chemin_glouton(grid_t* g)` qui renvoie une chemin généré par l'algorithme glouton.

Q20. Lancez l'algorithme sur les exemples fournis pour vérifier qu'il renvoie des chemins valides.

Q21. Trouver à la main un exemple où l'algorithme n'est pas optimal.

Programmation Dynamique

Ce problème se prête bien à la programmation dynamique, car on peut lui exhiber une structure récursive. On note $C(i,j)$ le nombre maximal de déchets que l'on peut ramasser en allant de $(0,0)$ à (i,j) (inclus)

Q22. Combien vaut $C(0,0)$?

Q23. Trouver une formule récursive pour exprimer $C(i,j)$ en fonction de $C(i-1,j), C(i,j-1)$ et $G[i][j]$.

On s'intéresse à une approche de bas en haut. On stocke donc les valeurs de $C(i,j)$ dans un tableau à deux dimensions.

Q24. Dans quel ordre peut-on remplir le tableau ?

Q25. Écrire une fonction `int** dechets_progdyn(grid_t* g, int n, int m)` qui calcule et renvoie le tableau de programmation dynamique décrit ci-dessus.

Q26. Vérifier que votre fonction trouve bien les valeurs optimales pour les trois grilles données dans l'archive.

Q27. Modifier votre fonction pour qu'elle construise et renvoie plutôt un tableau 2D D de caractères, tel que $D[i][j]$ donne la direction (Right ou Down) qui a été utilisée pour arriver sur la case (i, j).

Passons à la reconstruction du chemin. Tout se passe de manière très similaire au calcul de la distance de Levenshtein: on commence par se placer en $(n-1, m-1)$, et tant que l'on n'est pas en $(0, 0)$, on regarde si l'on vient du haut ou de la gauche en regardant dans le tableau construit à la question précédente.

Q28. Écrire une fonction `char* reconstruction(char** D, grid_t* g)` qui reconstruit un chemin optimal dans g en utilisant le tableau D, qui contiendra le résultat de la programmation dynamique. Cette fonction affichera la quantité totale de déchets ramassés.

Q29. Vérifier sur l'exemple en ligne que vous trouvez bien un chemin correct et optimal (sa valeur doit être 1189).

Bonus: Contrainte de place

Il y a plus de déchets que prévu, et vous vous rendez compte que votre sac poubelle ne peut pas contenir une infinité de déchets: il a une contenance K maximale. Cependant, vous détestez les déchets, et vous vous vous sentez obligés de ramasser tous les déchets des cases que vous visitez. Si votre sac ne peut pas contenir les déchets d'une case que vous visitez, vous êtes trop triste et vous rentrez chez vous. Vous devez donc choisir un chemin qui maximise la quantité de déchets ramassés **tout en en ramassant au plus** K.

On propose d'adapter la formule de récurrence obtenue à la partie précédente, en étudiant plutôt $C(i, j, k)$ la quantité maximale de déchets que l'on peut ramasser entre $(0, 0)$ et (i, j) en ayant un sac de contenance $k \leq K$, avec comme convention que $C(i, j, k) = -\infty$ s'il n'est pas possible d'aller de $(0, 0)$ à (i, j) avec un sac de contenance k.

Q30. Donner les cas de base et les formules de récurrence sur $C(i, j, k)$.

Q31. Justifier que la programmation dynamique **de haut en bas** est plus adaptée ici, et implémenter un algorithme permettant de résoudre le problème avec cette nouvelle contrainte.

Q32. Vérifier sur l'exemple en ligne qu'avec un sac de contenance 1000, on peut trouver un chemin permettant de ramasser exactement 1000 déchets.