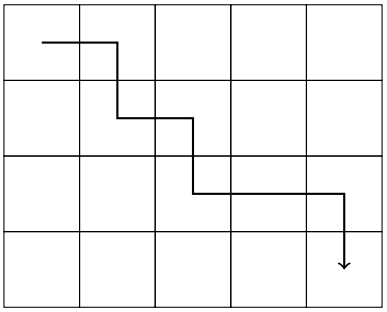
TP14: Stratégies algorithmiques

MP2I Lycée Pierre de Fermat

Vous déposerez votre code ainsi que votre document réponse sous la forme d'une archive sur Cahier de Prépa avant dimanche 28 mai à 22h00. Une archive

Nettoyage de grille

On considère une grille de taille $n \times m$. Vous habitez sur la case en haut à gauche de cette grille, de position $(0, 0)$ et votre travail se situe en bas à droite, sur la case $(n-1, m-1)$. Comme chaque matin, vous allez au travail en traversant la grille. Pour cela, vous pouvez vous déplacer d'une case à la fois, soit vers la droite, soit vers le bas. Par exemple:



De plus, sur chaque case de la grille se trouve un certain nombre de déchets que vous voulez ramasser pour garder la grille 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.

Commençons par écrire de quoi lire une instance du problème dont les données sont stockées dans un fichier. Pour représenter une instance du problème, on utilisera le format suivant:

- Sur la première ligne, deux entiers n, m indiquant les dimensions de la grille
- Sur les n lignes suivantes, m entiers indiquant le contenu de chaque ligne.

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

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

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

- Une grille de taille 400×750 dont la solution optimale est 25918

Question 1. Proposer une structure C pour stocker les grilles.

On appellera le type créé `GRID`.

Question 2. Écrire une fonction `GRID* 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

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.

Question 3. Écrire une fonction `int valeur(GRID* 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'` ou s'il sort de la grille.

Question 4. Écrire une fonction `char* chemin_aleatoire(GRID* 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.

Question 5. Lancez la fonction précédente plusieurs fois sur un même exemple.

A 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 parmis les deux cases adjacentes.

Question 6. Écrire une fonction `bool choix_glouton(GRID* g, int i, int j)` qui renvoie true si à partir de la case (i, j), l'algorithme glouton choisit la case à droite de (i, j), et false s'il choisit la case en bas. Cette fonction prendra en compte les cas où la case (i, j) se trouve au bord de la grille.

Question 7. Écrire une fonction `char* chemin_glouton(GRID* g)` qui renvoie une chemin généré par l'algorithme glouton.

Question 8. Lancez l'algorithme sur les exemples fournis pour vérifier qu'il fonctionne.

Question 9. Trouver un exemple où l'algorithme n'est pas optimal.

B 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)

Question 10. (Bonus) Combien y a-t-il de chemins distincts entre $(0, 0)$ et $(n-1, m-1)$?

Question 11. Combien vaut $C(0, 0)$?

Question 12. Trouver une formule récursive pour exprimer $C(i,j)$ en fonction de $C(i-1,j), C(i,j-1)$ et $G[i][j]$.

Question 13. Si l'on stocke les $C(i,j)$ dans un tableau, dans quel ordre peut-on les calculer pour une approche de bas en haut ?

On stocke donc les valeurs de $C(i,j)$ dans un tableau à deux dimensions.

Question 14. Écrire une fonction `int** dechets_progdyn(GRID* g, int n, int m)` qui calcule et renvoie le tableau de programmation dynamique décrit ci-dessus.

Question 15. Vérifier que votre fonction trouve bien les valeurs optimales pour les trois grilles données dans l'archive.

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 $(n-1, m-1)$ puis tant que l'on n'est pas en $(0,0)$, on calcule si l'on vient du haut ou de la gauche en réutilisant les mêmes formules que pour le calcul du tableau.

Question 16. Écrire une fonction `char* reconstruction(int** C, GRID* g)` qui reconstruit un chemin optimal dans g en utilisant le tableau C, qui contiendra le résultat de la programmation dynamique.

Question 17. 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 $C(i,j,k)$ la quantité maximale de déchets que l'on peut ramasser entre $(0,0)$ et $(n-1, m-1)$ en ayant un sac de contenance k.

Question 18. Exprimer $C(i,j,k)$ en fonction de certains $C(i',j',k')$.

Question 19. Implémenter un algorithme de programmation dynamique permettant de résoudre le problème avec cette nouvelle contrainte.

Question 20. Vérifier sur l'exemple moyen qu'avec un sac de contenance 1000, on peut trouver un chemin permettant de ramasser exactement 1000 déchets.

Problème des n reines

Le problème des n reines est d'arriver à placer n reines sur un échiquier $ntimesn$ sans qu'aucune n'en menace une autre. On rappelle que les reines peuvent se déplacer verticalement, horizontalement et diagonalement, aussi loin qu'elles le veulent.

Ce problème est dur à résoudre, mais est bien adapté au backtracking. Une remarque intéressante est qu'il doit y avoir une et une seule reine sur chaque ligne. Cette remarque va motiver le choix suivant d'algorithme. Étant donné une grille $n \times n$ sur lequel on a déjà placé des reines sur les lignes $1, 2, \ldots, i$, en notant q_i la colonne de la reine sur la ligne i, on se demande s'il est possible de placer des reines sur toutes les lignes suivantes. On peut donc tester de mettre une reine sur chaque case non menacée de la ligne $i+1$. L'algorithme de backtracking sera donc:

Algorithme 1 : Backtracking pour les n reines:

Entrée(s) : n taille d'échiquier, $q_1, \ldots q_i$ colonnes des reines déjà placées sur les lignes $1, \ldots i$, sans qu'aucune n'en menace une autre

Sortie(s) : Oui si l'on peut placer des reines sur les lignes $i+1, \ldots, n$ sans qu'aucune reine ne soit menacée par une autre, Non sinon

```
1 si i = n alors
2     retourner Oui
3 pour j = 1 à n faire
4     si la case (i, j) n'est pas menacée alors
5         Lancer récursivement l'algorithme pour (q_1, ..., q_i, j);
6         si La réponse est Oui alors
7             retourner Oui
8     retourner Non
```

Implémentons cet algorithme en OCaml

Question 1. Écrire une fonction `valid_queen (queens: (int * int) list) (i: int) (j: int): bool` qui détermine si la case (i, j) est menacée par une des reines de la liste `queens`.

Question 2. Implémenter une fonction `n_queens (n: int) (queens: (int * int) list) (i: int)` utilisant le principe de backtracking, qui détermine si le problème des n reines a une solution sachant que des reines ont déjà été placées sur les lignes 1 à i, et que leurs positions sont stockées dans `queens`.

Question 3. Vérifier que le problème des 3 reines n'a pas de solution, mais que le problème des 8 reines en a une.

Question 4. Modifier l'algorithme pour qu'il renvoie la liste des reines à placer pour obtenir une solution (lorsqu'une solution existe).

Question 5. Résoudre le problème des 8 reines, des 24 reines, des 30 reines.

Découpage d'une barre

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 cours de la barre de métal est particulier: le coût d'un tronçon n'est pas proportionnel à sa longueur, ni même croissant en fonction de sa longueur. Cependant, vous disposez d'une grande table vous donnant le prix auquel vous pouvez vendre n'importe quel tronçon.

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

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

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 les tronçons par une liste $[x_1; \ldots; x_k]$ telle que $x_1 + \cdots + x_k = n$. Enfin, le prix est modélisé par une fonction $f : \mathbb{N} \to \mathbb{N}$ telle que $f(k)$ donne le prix en euros d'un tronçon de k centimètres.

A Glouton

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

Question 1. Écrire une fonction `glouton_prix (f: int -> int) (n:int) : int` qui calcule un découpage d'une barre de n centimètres selon cette méthode, sachant que la fonction `f` donne le prix d'une longueur donnée de barre de métal.

Question 2. 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.

Question 3. Implémenter cette méthode avec une fonction `glouton_ratio`

Question 4. Trouver un exemple sur lequel cet algorithme n'est pas optimal.

B Approche récursive

On s'intéresse à une approche plus récursive du problème. On note $C(n)$ le prix maximal auquel on peut vendre une barre de n centimètres.

Question 5. En considérant la taille du premier tronçon que l'on découpe, donner une formule de récurrence Liant $C(n)$ aux $C(i)$ pour $i < n$ et aux prix $f(i)$ pour $i \leq n$.

Question 6. En utilisant cette formule, écrire une fonction récursive `decoupe_optimale (f: int -> int) (n: in`

Question 7. Quelle est sa complexité ? Pour quelles valeurs de n n'est-il plus envisageable de l'utiliser ?

C Programmation dynamique

Nous allons maintenant améliorer cette fonction, en utilisant la programmation dynamique, par une approche de haut-en-bas. Nous allons donc mettre en place un système de mémoïsation. Pour cela, nous allons utiliser un aspect impératif d'OCaml: les tableaux.

Le type `'a array` est le type des tableaux. Voici un exemple de tableau:

```
let t = [|12;17;6;8|];;
let x = t.(2);; (* vaut 6 *)
```

Un tableau ressemble beaucoup à une liste, mais se comporte de manière très différente:

- Un tableau est de taille fixe
- On peut accéder en temps constant à n'importe quelle case d'un tableau avec la syntaxe `tab.(i)`.
- Un tableau est mutable !

Cette dernière propriété est **totalement nouvelle** par rapport à ce que nous avons fait en OCaml jusqu'à maintenant. Un tableau peut être modifié à l'intérieur d'une fonction, ce qui aura un effet permanent et global. Pour écrire x dans la case i d'un tableau t, on écrit `t.(i) <- x`. Par exemple:

```
(* modifie la première case de t en la mettant au carré, et renvoie
   l'ancienne valeur *)
let modif t =
  let x = t.(0) in
  t.(0) <- (x*x); x
;;

let t = [|5;6;7|];;
let a = modif t;;
t.(0);; (* vaut maintenant 25 *)
```

Comme tout en OCaml, `t.(i) <- x` est une expression. Son type est `unit`, tout comme les instructions d'affichage.

En OCaml, on peut créer des tableaux de tailles arbitraire avec la fonction `Array.make`: `Array.make n x` renvoie un tableau de taille n avec x dans chaque case.

En OCaml, l'approche de haut-en-bas de la programmation dynamique est assez naturelle à implémenter en utilisant les tableaux. 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:

```
(* Valeur optimale pour découper une barre de longueur n, selon
   la fonction de prix f. *)
let decoupe_prog_dyn (f: int -> int) (n: int) : int =
  let t = Array.make (n+1) (-1) in (* t: tableau de mémoisation *)

  (* Renvoie le meilleur prix possible pour une barre de taille i. Commence
     par regarder dans t si la valeur a déjà été calculée, et si ce n'est
     pas le cas, la calcule puis l'ajoute dans le tableau. *)
  let rec calculer_decoupe i =
    if t.(i) = -1 then
```

```
11        (* calculer récursivement C(i) et le stocker dans t.(i) *);
12      t.(i)
13    in calculer_decoupe n
14  ;;
```

Question 8. Compléter le code précédent pour implémenter la programmation dynamique sur le découpage de barre.

Question 9. Modifier le code pour pouvoir calculer non seulement la valeur optimale mais aussi la liste des tronçons à vendre pour l'atteindre.