DEVOIR MAISON VI

Projet Euler 215

Ce DM est en OCaml.

Conditions de rendu

Vous pouvez rendre un travail pour deux. Vous devez rendre les questions théoriques sur papier au début du cours d'informatique de lundi 16 juin 2025. Vous devez déposer le zip créé par `make zip` sur le lien suivant avant dimanche 15 juin 2025, 20h00 (heure de Paris) :

`https://nuage04.apps.education.fr/index.php/s/TRFmjsxQWojcBnY`

Vous avez le droit d'utiliser librement les fonctions des modules `List`, `Hashtbl`, `Int` et `Array`.

Fichiers fournis

- Un `Makefile` :
  - `make main` compile avec votre `main`. L'exécutable produit s'appelle `main.exe` .
  - `make test` compile avec le testeur. L'exécutable produit s'appelle `testeur.exe` .
  - `make zip` crée le zip.
  - `make clean` efface les intermédiaires de compilation.
  - `make cleanall` efface les intermédiaires et les exécutables.
- `euler215.ml` : c'est ici que vous programmerez.
- `euler215.mli` : l'interface qui spécifie les fonctions. Merci de ne pas la modifier sans me le demander.
- `main.ml` : votre main à vous, pour faire vos tests. Faites-y ce que vous voulez, je n'y toucherai pas !
- `testeur.cmo` et `testeur.cmi` : une version compilée du testeur.

Projet Euler

Le projet Euler est un site d'entrainement mathématique et informatique, qui propose plusieurs centaines de problèmes : `https://projecteuler.net/` . Vous pouvez ajouter ce site à la liste des sites que je vous recommande pour vous entrainer, aux côtés de France-IOI, Prologin, Advent of Code.

Dans ce DM, on s'intéresse au problème 215 :

`https://projecteuler.net/problem=215`

Je vous laisse aller en lire l'énoncé.[1] Dans la suite de cet énoncé, je traduirai ainsi :

- Les briques sont des 2-brique et de 3-brique
- les deux dimensions s'appeleront « horizontale » (notée h) et « verticale » (notée v).
- Une *running crack* est une collision.

I - Modélisation

Dans ce DM, on utilise la modélisation suivante : un mur est une succession de lignes ; et une ligne est une succession d'emplacements qui contiennent un « trait vertical » ou un « pas trait vertical » :

1. Oui, il est en anglais... mais la documentation que vous aurez aux oraux de concours aussi, peut-être même celle que vous aurez aux écrits... Cela ne me réjouit pas, mais vous *devez* pouvoir lire de l'anglais.

1) Encodage binaire

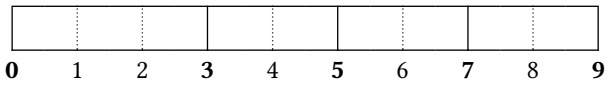

FIGURE VI.1 – La première ligne du mur de l'énoncé et ses emplacements indiqués en indice. En gras les emplacements correspondant à un trait, en non-gras ceux qui correspondent à une absence de trait.

3. Le mur ci-dessus est de dim. horizontale $h = 9$. Combien a-t-il d'emplacements ? Plus généralement, combien a d'emplacements une ligne de dimension horizotnale h ?

On va modéliser cette succession d'emplacements contenant un trait ou non par une suite de (respectivement) 1 et de 0. Ainsi, le mur de la figure précédente correspond à :

1001010101

On peut interpréter cela comme un entier (en l'occurence, l'entier 597). Pour stocker une ligne, on va donc se contenter de stocker l'entier qui code la suite de 0 et de 1 ainsi que son nombre d'emplacements[2] :

euler215.ml

```
8 type ligne = {code : int; nb_empl : int}
```

4. (Au brouillon uniquement, à ne pas rendre) : trouver les lignes (code et nombre d'emplacements) correspondant au mur :

- **a.** contenant uniquement une 2-brique
- **b.** contenant uniquement une 3-brique
- **c.** contenant une 3-brique puis une 2-brique
- **d.** vide, c'est à dire le mur qui n'a aucune brique (pas évident, je vous conseille de l'obtenir à l'aide de la question suivante).

Vous pouvez vérifier vos résultats grâce au début de `euler215.ml` .

Astuce : pour traduire 1001010101 en entier, en OCaml on peut évaluer `int_of_string "0b1001010101"` . Le « 0b » qui préfixe la string sert à indiquer que ce qui suit est du binaire.

5. On considère une ligne dont on note c l'entier qui la code. Expliquer pourquoi :

- **a.** Si on rajoute une 2-brique à la fin, on obtient une ligne dont l'entier associé est $4 * c + 1$.
- **b.** Si on rajoute une 3-brique à la fin, on obtient une ligne dont l'entier associé est $8 * c + 1$.

2) Conflit entre deux lignes

Cette représentation en binaire a pour but de pouvoir tester efficacement si deux lignes sont en conflit : on va faire le « ET binaire »[3] bit à bit (noté & dans cet énoncé) des codes des deux lignes et conclure.
Par exemple, pour les deux premières lignes de l'énoncé, cela donne :

Première ligne	1	0	0	1	0	1	0	1	0	1
Seconde ligne	1	0	1	0	1	0	0	1	0	1
Et bit à bit	1	0	0	0	0	0	0	**1**	0	1

FIGURE VI.2 – « ET binaire » bit à bit

Ainsi, deux lignes sont en collision si et seulement si l'entier correspond à leur ET binaire bit à bit n'est pas 10...01 (les seuls 1 sont au premier et dernier emplament).

6. Coder une fonction `ligne_creuse : int -> ligne` telle que `ligne_creuse nb` vérifie :

- Si `nb >= 2`, c'est la ligne à `nb` emplacements dont seul le premier et le dernier contiennent un trait. On pourra utiliser l'opérateur infixe `lsl` : `x lsl k` prend l'entier `x` et rajoute `k` zéros à la fin. Ainsi, `1 lsl 3` vaut `8` . On admet que cet opérateur s'exécute en temps constant.

2. Et non sa dimension horizontale.
3. 0 ET _ = 0 ; _ ET 0 = 0 ; 1 ET 1 = 1

- Sinon, on déclenchera l'exception `Invalid_argument "ligne_creuse"`. On rappelle que `raise` est la fonction qui prend en argument une exception et permet de la déclencher.

7. Coder une fonction `est_collision : ligne -> ligne -> bool` telle que `est_collision p q` :
   - si les lignes ont le même nombre d'emplacements, renvoie `true` si et seulement si les deux lignes ont une collision (rappel : le mur tout à gauche et tout à droite *ne* sont *pas* des collisions).
   - sinon, lève `Invalid_argument "est_collision"`.

   Vous pouvez utiliser l'opérateur infixe `land` qui correspond au ET binaire bit à bit. On admet que cet opérateur s'effectue en temps constant[4].
8. Prouver la complexité de `est_collision`.

II - Résolution

Pour résoudre le problème, nous allons commencer par générer la liste des lignes possibles. Dans un second temps, nous utiliserons cette liste pour calculer toutes les façons valides de créer le mur.

1) Génération des lignes possibles

On dit qu'une ligne q **étend** une ligne p si le code de p est un préfixe du code de q.

Exemple. Voici la liste des extensions à 15 emplacements de la ligne contenant uniquement une 2-brique :

```
(* etend 15 ligne_b2 = *)
[{code = 21065; nb_empl = 15}; {code = 21077; nb_empl = 15}; {code = 21141; nb_empl = 15};
 {code = 21157; nb_empl = 15}; {code = 21161; nb_empl = 15}; {code = 21653; nb_empl = 15};
 {code = 21669; nb_empl = 15}; {code = 21673; nb_empl = 15}; {code = 21797; nb_empl = 15};
 {code = 21801; nb_empl = 15}; {code = 21833; nb_empl = 15}; {code = 21845; nb_empl = 15}
]
```

9. Coder la fonction `etend` telle que `etend nb_goal lgn` est la liste des extensions à `nb_goal` emplcaments qui sont des extensions de `lgn`.
10. En déduire `calcule_lignes` telle que `caclule_lignes h` est la liste de toutes les lignes valides de dimension horizontale `h`.

2) Résolution

11. Coder la fonction `sum_map` telle que `sum_map f [x1; ...; xN]` soit `fx1 + ... + f xN`.
12. Coder la fonction `euler215` telle que `euler215 h v` soit le nombre de façons de monter un mur sans collisions ayant les dimensions indiquées.
    Pour vous aider à tester : `euler215 9 3 = 8`; `euler215 9 10 = 66`; `euler215 20 10 = 94082988`; `euler215 32 10 = 806844323190414`
13. Prouver sa complexité, en ignorant le coût de calcul de `calcule_ligne`.

Remarque. On pourrait accélérer le code en n'utilisant pas le type `ligne` mais uniquement les codes qu'il contient[5]. Cependant, mémoriser aussi les nombres d'emplacements (et vérifier la cohérence lors des tests de collision) permet d'éviter un certain nombre de bugs. Il y aurait encore tout un tas d'autres petites améliorations possibles.

3) Indication

Des indications sont sur la prochaine page. Avant que vous ne la regardiez, je vous encourage fortement à chercher au brouillon, échanger avec votre peluche, échanger avec vos camarades, et faire encore plus de dessins au brouillon.

4. En fait, `lsl` et `land` sont des opérations encore plus élémentaires qu'une addition.
5. Sur l'entrée (h=32,v=10), je suis deux fois plus rapide avec ce changement !

III - Indications

Q9 : Les extensions de la ligne sont les extensions de la ligne plus une 2-brique et de la ligne plus une 3-brique.

Q12 : Utilisez une fonction auxiliaire qui prend un argument supplémentaire : la ligne de base (celle toute en haut du mur).
Cette fonction auxiliaire est récursive.
Si l'auxiliaire est trop lente, utilisez une méthode pertinente vue en cours pour l'accélérer.
Vous aurez peut-être envie de faire « pour chaque ligne possible, ...». Utilisez `iter` ou `map` ou `sum_map`.

Q13 : On pourra faire apparaître le nombre de lignes possibles dans la complexité.