Dungeons&Diagrams

Pour résoudre ce problème, on partira du fichier d_and_d.c.

I Présentation du problème et de la résolution

I.1 D&D

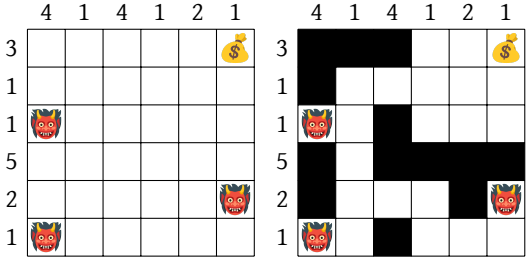
Dungeons&Diagrams (D&D) est un casse-tête type *Sudoku* inventé par Zach Barth et présenté initialement sur son site en version papier et, plus récemment, au sein du jeu *Last Call BBS* de *Zachtronics*.

Une grille de D&D est la donnée d'une grille rectangulaire sur laquelle figure des monstres 👹 et des trésors 💰, ainsi que des indications chiffrées sur les colonnes et les lignes. Le but est de remplir certaines cases vides par des murs en respectant les règles suivantes :

- Le nombre de murs sur une ligne (resp. une colonne) correspond au nombre associé à cette ligne (resp. cette colonne)
- Tous les monstres sont dans des impasses et toutes les impasses contiennent des monstres. Une impasse étant une case ayant exactement une case voisine libre.
- Chaque trésor est dans une *salle au trésor* qui est un carré sans mur de taile 3x3 ne contenant aucun monstre et un seul trésor et qui, de plus, est bordé de murs sauf en exactement une case qui doit être libre.
- Le graphe induit par la grille avec des connexions directes, mais pas en diagonale, est connexe.

On suppose que les grilles admettent une unique solution.

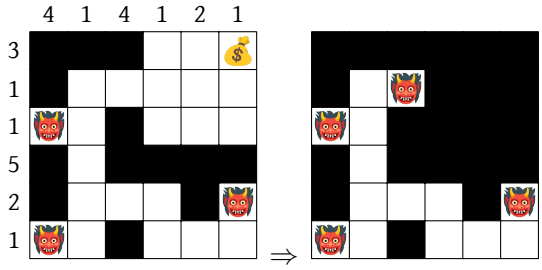
Ainsi, par exemple, une grille 6x6 et son unique solution



Question I.1 Résoudre manuellement les 9 problèmes du PDF donné en lien au-dessus.

Question I.2 Que peut-on dire du graphe d'une solution pour un problème sans trésor ?

Si on considère une grille satisfaisant les contraintes de lignes et de colonnes, une fois une salle au trésor identifiée, il est possible de la remplir et de placer un monstre à son unique entrée.



On en déduit ainsi une méthode de vérification des solutions :

- tester qu'il y a le bon nombre de murs dans chaque ligne et chaque colonne
- vérifier que chaque trésor est dans une salle au trésor conforme
- remplir les salles au trésor et placer un monstre à leur entrée
- vérifier que le graphe est un arbre dont les feuilles sont les cases contenant des monstres.

En pratique, n'est pas nécessaire de vérifier la dernière condition avec un parcours de graphe, on peut se contenter de vérifier que graphe est localement arborescent :

- chaque case avec un monstre est de degré 1
- chaque case libre est de degré au moins 2
- il n'existe pas de carré de cases libres

Question I.3 Pourquoi cette vérification n'est pas suffisante *a priori* ?
Bonus trouver une grille pour laquelle cette approximation ne permet pas d'obtenir la solution.

Compte tenu de la vérification de la présence d'une unique entrée dans une salle au trésor, on peut réordonner les vérifications :

- tester qu'il y a le bon nombre de murs dans chaque ligne et chaque colonne
- vérifier que les cases contenant des monstres sont de degré 1
- vérifier que les cases libres sont de degré au moins 2
- vérifier que chaque trésor est dans une salle au trésor conforme
- remplir les salles au trésor
- vérifier que la grille résultante ne contient pas de carré de cases libres

I.2 Représentation des plateaux en 64bit

On va résoudre ce casse-tête par *backtracking* en C. Les grilles étant de taille 8x8, on peut ainsi représenter une information booléenne sur chaque case à l'aide d'entiers non signés sur 64bit : chaque bit est associé à une case.

On choisit ici l'ordre suivant :

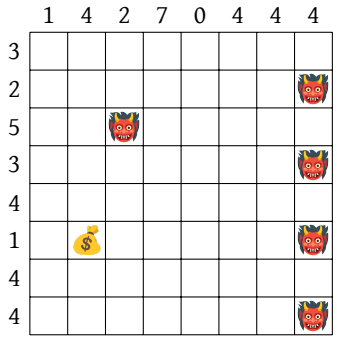
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
...							

qui correspond au mot binaire $a_{63}...a_0$ sur 64bits avec a_i représentant l'information sur la case i.

On va associer trois entiers à une grille :

- le plateau (*board*) où un booléen vrai indique la présence d'un mur
- les monstres où un booléen vrai indique la présence d'un monstre
- les trésors où un booléen vrai indique la présence d'un trésor

Par exemple, pour la grille :



on a les monstres dans les positions 15, 18, 31, 47 et 63. Cela correspond alors à l'entier s'écrivant en binaire

1000000000000000100000000000000010000000000001001000000000000000

et qui vaut 9223512776490909696 en décimal ainsi que `0x8000800080048000` en hexadécimal.

On pourra écrire en C :

```c
// au début du fichier
#include <stdint.h>
typedef uint64_t board;

// puis
board monsters = 0x8000800080048000;
```

On pourrait considérer une représentation *developpée* sous la forme d'un tableau de 64 booléens. Dans ce cas-là, on obtiendrait la valeur suivante en C :

```c
// au début du fichier
#include <stdbool.h>
typedef bool map[64];

// puis

map monsters = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
};
```

Question I.4 Écrire une fonction de signature

```c
board map_to_board(map m);
```

qui renvoie la représentation entière associée à un tableau.

Question I.5 Écrire une fonction de signature

```c
void board_to_map(board b, map m);
```

qui prend en entrée une représentation entière `b`, un tableau `m` de 64 cases et place dans `m` le tableau de booléens déduit de `b`.

Question I.6 Faire des tests pour vérifier que ces deux fonctions sont correctes. Notamment, qu'elles sont réciproques l'une de l'autre.

I.3 Représentation du problème

Pour représenter le problème, on utilise le type suivant en C :

```c
typedef char constraint[8];

struct problem {
    board monsters;
    board treasures;
    constraint rows;
    constraint columns;
};

typedef struct problem problem;
```

Pour définir un problème, on peut le faire directement comme ici :

```c
problem p = {
    0x8000800080048000,
    0x20000000000,
    { 3,2,5,3,4,1,4,4 },
    { 1,4,2,7,0,4,4,4 }
};
```

Mais par lisibilité, on peut préciser les noms des champs comme ici :

```c
problem p = {
    .monsters = 0x8000800080048000,
    .treasures = 0x20000000000,
    .rows = { 3,2,5,3,4,1,4,4 },
    .columns = { 1,4,2,7,0,4,4,4 }
};
```

Cette dernière syntaxe a l'avantage d'initialiser à 0 tous les champs omis. On peut ainsi utiliser les fonctions précédemment définies pour présenter le problème de manière plus lisible dans le code source :

```c
problem p = {
    .rows = { 3,2,5,3,4,1,4,4 },
    .columns = { 1,4,2,7,0,4,4,4 }
};

map monsters = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
};

map treasures = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
};
// ou plus rapidement
map treasure = { [41] = 1 }; // met 1 à l'indice 41 et 0 ailleurs
```

```
p.monsters = map_to_board(monsters);
p.treasures = map_to_board(treasures);
```

I.4 Choix pour le backtracking

On va effectuer des remplissages incrémentaux de la grille pour résoudre ce problème par backtracking. On reprend donc la terminologie de plateau et de coups permettant de passer d'un plateau à un autre.

Une majeure partie des conditions de résolution portant sur des grilles complètes, on pourrait se contenter de vérifier uniquement les grilles remplies. En faisant cela, on obtient un arbre déraisonnablement grand.

Par exemple, avec comme coup le fait, pour une position donnée, de choisir ou pas de placer un mur, on obtient un arbre binaire ayant 2^{64} feuilles.

On va plutôt considérer les coups consistants à placer directement k murs sur une ligne dont on sait qu'elle doit en contenir k.

Question I.7 Justifier que si les indications de contraintes sur les lignes sont données par les nombres $l_1, \dots, l_8$ alors on obtient ainsi un arbre de backtracking ayant

$$\prod_{k=1}^{8} \binom{8}{l_k}$$

feuilles.

Dans le cas où $l_1 = \cdots = l_8 = 4$, comparer ce nombre avec le nombre obtenu en effectuant des mouvements case par case.

Même si ce nombre est beaucoup plus petit que le précédent, cela reste prohibitif pour faire une recherche exhaustive.

On va donc adopter une détection des nœuds pour lesquels on s'avère certain de ne pas pouvoir résoudre le problème.

Tout d'abord, on restreint les lignes à celles qui ne placent pas un mur sur la même case qu'un monstre ou un trésor.

Ensuite, si on un nœud correspondant à avoir rempli les k premières lignes, on vérifie :

- que le nombre de murs dans une colonne n'est pas strictement supérieur à la contrainte (**column overload**)
- qu'aucun monstre n'est sur une case de degré 0 (**trapped monsters**)
- qu'aucune case libre sur les k premières lignes n'est de degré ≤ 1 (**no deadend upto row**)

On a indiqué entre parenthèses les noms des fonctions qu'on programmera dans la suite pour réaliser ces tests.

II Opérations bit à bit en C

Dans ce paragraphe, on notera `0b10101` une valeur en binaire en C. Cette notation, calquée sur la notation hexadécimale, n'est pas valide, mais rendra les exemples plus lisibles. De même, dans les formules, on notera 10101_2 un nombre écrit en binaire.

On va étudier ici des opérateurs opérant directement sur l'écriture binaire des entiers. Aucun de ces opérateurs n'est explicitement au programme, cependant, leur usage permet, selon les contextes, d'avoir code plus efficace et plus lisible.

II.1 Opérateurs de décalage

On utilise l'opération `x >> k` pour décaler l'écriture de x de k bits vers la droite, cela revient donc à réaliser une division par 2^k.

De même, on utilise l'opération `x << k` pour décaler son écriture de k bits vers la gauche *en rajoutant des 0*.

Exemple :

- `123 >> 3` vaut 15 car `123=0b1111011` donc `123 >> 3 = 0b1111 = 15`
- `123 << 2` vaut 492 car `123 << 2 = 0b111101100 = 492`

Pour calculer 2^k on peut ainsi écrire `1 << k` mais cela pose un problème pour $k \geq 32$ parce qeu 1 est un `int` et le calcul fait apparaitre un dépassement de capacité qui induit une troncature à 0. Pour $k \leq 63$, on peut régler ce problème en utilisant `1UL << k` car `1UL` est la constante `1` en `unsigned long`.

Cependant, comme ce problème n'est pas propre aux constantes, on préfèrera utiliser une coercition comme ici : `(uint64_t)x << k` qui fonctionne quand x est un int.

On donne dans le code fourni une fonction

```c
uint64_t bit(int pos)
{
    return (uint64_t) 1 << pos;
}
```

afin de pouvoir écrire directement `bit(n)` pour calculer 2^n avec $n \in [\![0, 63]\!]$.

II.2 Opérateurs logiques

Il est possible de réaliser les opérations logiques connues sur les booléens en les appliquant *bit à bit*. Pour cela, on utilise :

- `a & b` pour réaliser un *et* logique entre chaque bit. Ainsi, en écrivant abusivement en binaire, on a `0b10110 & 0b11010` qui vaut `0b10010`.
- `a | b` pour réaliser un *ou* logique entre chaque bit. Ainsi, on a `0b10110 | 0b11010` qui vaut `0b11110`.
- `a ^ b` pour réaliser un *ou exclusif* logique entre chaque bit. Ainsi, on a `0b10110 ^ 0b11010` qui vaut `0b01100`.
- `~a` pour réaliser la *négation* logique de chaque bit. Ainsi, on a `~0b10101` qui vaut `0b01010`.

Une manière classique d'utiliser `&` est de réaliser des masquages : on stocke dans une variable `mask` les bits que l'on veut étudier et on écrit `x & mask` pour ne conserver que ceux-ci.'

L'opération `|` permet de réaliser efficacement une addition d'entiers dont les bits sont disjoints. En effet, comme il n'y a pas de retenue dans ce cas-là, il suffit de superposer les écritures avec un *ou*.

Remarque Ces opérations utilisent de simples portes logiques et sont très rapides à exécuter sur un processeur.

Notons que le compilateur les privilégie souvent. Par exemple, pour tester la parité `x % 2 == 0` sera traduit par un test de non-nullité de `x & 1` et les divisions/multiplications par 2^k sont remplacées par des décalages.

II.3 Popcount

On va avoir besoin de compter les murs ou les cases libres, pour cela on utilise une opération appelée `popcount` (pour *population count*) qui permet, étant donné un entier non signé `x` sur 64bits de renvoyer le nombre $p(x) \in [\![0, 64]\!]$ de 1 dans son écriture binaire.

Dans la suite, on utilisera cette fonction sur un plateau pour compter les murs ou sur sa négation pour compter les cases libres.

II.3.i Calcul naïf

Pour réaliser un calcul naïf, on va juste itérer sur les bits en faisant des tests de parité et des division par 2.

Question II.1 Écrire une fonction de signature

```c
int popcount_slow(uint64_t x);
```

qui effectue ce calcul.

On pourrait penser que ce code est long en raison de la boucle, mais en fait, celle-ci sera sûrement déroulée par le compilateur et le problème vient des branchements induits par les tests de parité.

II.3.ii Calcul par masquage et additions

Soit $p > 0$ et x, y deux entiers tels que $x, y \leq p < 2^p$. On a alors $x + y \leq 2p < 2^{p+1}$ ainsi, si $z = x2^p + y$ en considérant le masque $m = 2^p - 1$ on peut calculer $x + y$ ainsi :

```c
(z & m) + ((z & (m << p)) >> p)
```

Cette remarque a priori anodine permet de réaliser un calcul efficace de `popcount`. En effet, on souhaite effectuer en place la somme de tous les bits d'un entier.

Si `x` est un entier sur 8 bits, on peut se servir du procédé de masquage pour réaliser plusieurs additions d'un

coup à l'aide des masques

$$m_1 = \overline{01010101}^2 \quad m_2 = \overline{00110011}^2 \quad m_3 = \overline{00001111}^2$$

Par exemple, si $x = 173 = \overline{10101101}^2$, on va avoir :

z	m	p	$z\&m$	$(z\&(m << p)) >> p$	$z\&m + (z\&(m << p)) >> p$
10101101	01010101	1	00000101	01010100	01011001
01011001	00110011	2	00010001	00010010	00100011
00100011	00001111	4	00000011	00000010	00000101

Ici, les couleurs servent à identifier les sous-mots sur lesquels portent les additions.
Le dernier résultat est alors le nombre de 1 présent dans x : $\overline{00000101}^2 = 5$.
On en déduit le programme C suivant :

```c
unsigned char popcount_byte(unsigned char x)
{
    unsigned char m1 = 0x55; // 0b01010101
    unsigned char m2 = 0x33; // 0b00110011
    unsigned char m3 = 0x0f; // 0b00001111

    x = (x&m1) + ((x&(m1<<1))>>1);
    x = (x&m2) + ((x&(m2<<2))>>2);
    x = (x&m3) + ((x&(m3<<4))>>4);

    return x;
}
```

Question II.2 En déduire une fonction de signature

```c
int popcount(uint64_t x);
```

qui calcule le popcount d'un entier sur 64bits en utilisant cette méthode. On peut se contenter de renvoyer un int car on sait que la valeur renvoyée est dans $[\![0, 64]\!]$.

Question II.3 Combien cette fonction fait-elle d'opérations élémentaires ?

II.3.iii Instruction assembleur

Il existe une fonction `__builtin_popcountll` permettant de réaliser un popcount grâce à une instruction assembleur quand elle est disponible sur l'architecture visée, ou avec le code qu'on vient de voir quand ce n'est pas le cas.

Si votre processeur dispose de cette instruction, vous pouvez compiler avec le flag `-mpopcnt` pour l'utiliser. Cela permettra d'améliorer les performances.

III Squelette du backtracking

La fonction récursive `solve` réalisant le backtracking a la structure suivante :

```c
void solve(problem *p, int row, board b)
{
    // teste si b est une impasse et sort

    // b est une feuille de l'arbre
    if(row == 8)
    {
        // teste et sort si ce n'est pas une solution

        printf("Solution %lx\n", b);
        print_board(p, b);
        return;
```

```
    }

    // pour chaque mouvement move valide dans
    // la rangée row
        solve(p, row+1, b+move);
}
```

Question III.1 Écrire une fonction de signature

```
void print_board(problem *p, board b);
```

qui permet d'afficher le plateau avec un affichage comme celui-ci :

```
###...##
#...#...
M.#.#...
#...#T..
###.####
.....#.M
M#M#.#.#
####...#
```

où `#` est un mur, `M` est un monstre, `T` un trésor et `.` est une case libre.

Afin de réaliser la boucle sur les mouvements, on peut itérer sur les 256 placements possibles de murs, ne garder que ceux qui ont le bon nombre de murs et qui ne se superposent pas aux monstres et aux trésors.

Question III.2 Écrire le test permettant de s'assurer que `move` n'est pas en conflit avec les monstres et les trésors du problème `p`.

Question III.3 Écrire la boucle en utilisant `popcount` pour tester qu'il y a le bon nombre de murs.

La fonction précédente va refaire continuellement les mêmes calculs de `popcount`.

Question III.4 Dans le programme donné, il y a une variable globale

```
int enum_popcount_byte[256];
```

Rajouter les instructions dans `main` pour que `enum_popcount_byte[i]` contienne la valeur `popcount(i)` et en déduire une version plus efficace de la boucle précédente.

Remarque On peut réduire les branchements en précalculant les tableaux de placements pour chaque nombre de murs.

IV Implémentation des différentes vérifications

IV.1 Tests pour les colonnes

Il n'est pas nécessaire de tester si les lignes ont le bon nombre de murs, car on a restreint le backtracking pour l'assurer.

Question IV.1 Déterminer un masque correspondant à la première colonne.
En déduire, à l'aide de décalages, une fonction de signature

```
uint64_t mask_column(int i)
```

renvoyant un masque correspondant à la colonne `i`.

Question IV.2 En déduire des fonctions de signature

```c
bool test_columns_overload(problem *p, board b);
bool test_columns_underload(problem *p, board b);
```

Testant respectivement si une colonne à trop ou pas assez de murs dans le plateau.

IV.2 Impasses

Question IV.3 Écrire une fonction de signature

```c
bool empty(problem *p, board b, int pos);
```

indiquant s'il y a une case libre dans le plateau `b` à la position `pos`.
Attention, une case contenant un monstre ou un trésor n'est pas libre.

Question IV.4 Écrire des fonctions de signature

```c
bool treasure(problem *p, int pos);
bool monster(problem *p, int pos);
```

déterminant respectivement s'il y a un trésor ou un monstre à la position `pos`.

Question IV.5 Écrire une fonction de signature

```c
uint64_t neighbours(int pos);
```

qui renvoie le masque correspondant aux cases voisines de la position `pos`. Ce masque contiendra ainsi de 2 à 4 bits à 1 selon les positions (2 aux coins, 3 sur les bords stricts et 4 à l'intérieur du plateau).

Question IV.6 En déduire une fonction de signature

```c
int count_non_wall(problem *p, board b, int pos);
```

qui compte le nombre de cases qui ne sont pas des murs dans le voisinage de `pos`.
On pensera bien à prendre en compte la négation du plateau pour ne pas avoir besoin de faire de cas selon qu'on soit à l'intérieur ou sur un bord.

Maintenant qu'on peut compter le degré d'une position dans le graphe, on peut réaliser les tests présentés plus haut.

Question IV.7 Écrire une fonction de signature

```c
bool test_deadends(problem *p, board b);
```

qui vérifie que dans le plateau `b` : * tous les monstres sont des impasses * aucune case libre n'est dans une impasse.

Question IV.8 Écrire une fonction de signature

```c
bool test_no_deadend_upto_row(problem *p, board b, int row);
```

qui vérifie qu'aucune case libre n'est dans une impasse dans les rangées d'indice `< row`.

Question IV.9 Écrire une fonction de signature

```c
bool test_trapped_monster(problem *p, board b);
```

qui vérifie qu'aucun monstre n'est piégé par des murs.

IV.2.i Carrés libres

Question IV.10 Déterminer un masque correspondant à un carré de quatre cases commençant à la position 0.

Question IV.11 En déduire une fonction de signature

```
bool test_empty_squares(problem *p, board b);
```

qui teste qu'il n'y a pas de carré de quatre cases libres dans le plateau b.

Remarque À partir d'ici, vous pouvez résoudre tous les problèmes ne comportant pas de trésors.

IV.2.ii Salles au trésor

Pour chaque trésor, on va appliquer la méthode suivante :

- à l'aide d'un masque correspondant à un carré 3x3, identifier une possible salle au trésor autour du trésor (pas la peine de toute les chercher, il ne peut il y en avoir qu'une)
- à l'aide d'un masque correspondant à la bordure de la salle, en prenant garde au fait qu'elle n'est pas nécessairement à l'intérieur du plateau, déterminer l'unique ouverture de la salle
- remplir la salle au trésor de mur

Si jamais on n'arrive pas à trouver une salle valide, on sait que le plateau est insoluble.

Question IV.12 Écrire une fonction de signature

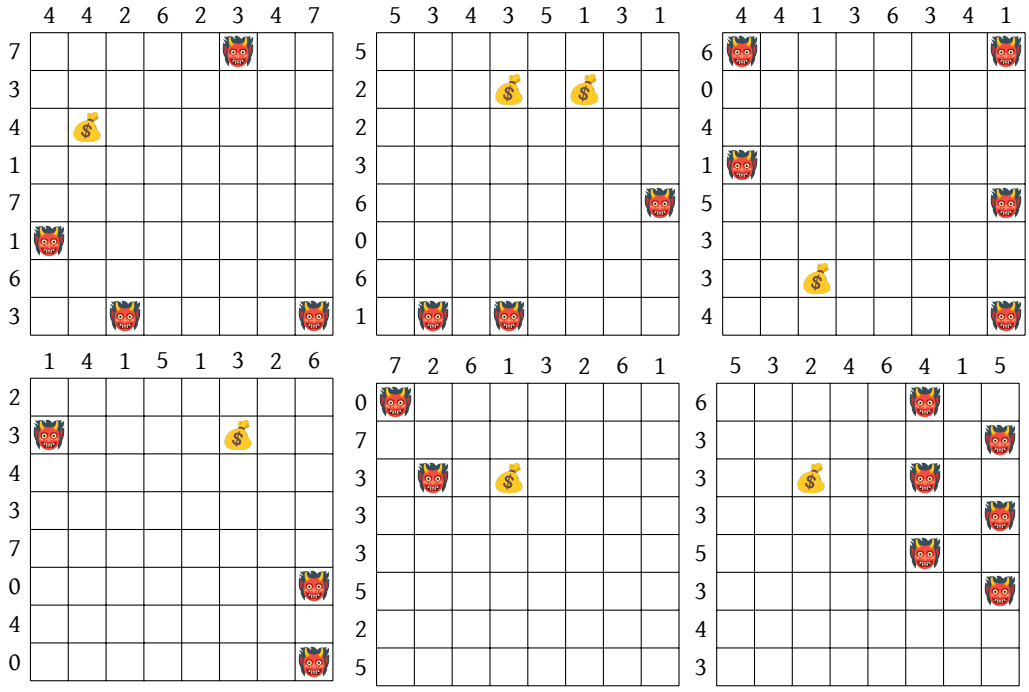
```
bool test_treasure_rooms(problem *p, board *b);
```

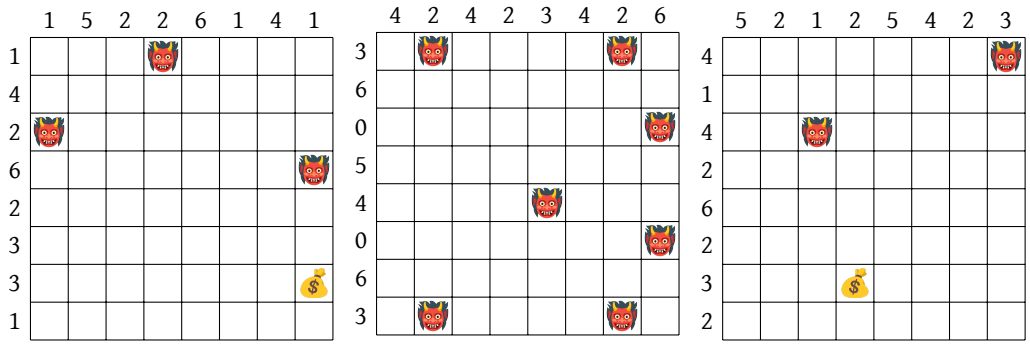
qui teste si toutes les trésors sont dans des salles valides et les remplit par des murs à l'aide du pointeur b.

Remarque Vous pouvez maintenant résoudre tous les problèmes. En tenant compte de toutes les optimisations dans les remarques et en compilant en `-O3`, il est possible de résoudre les 64 problèmes en moins de 500ms !

V Exemples

Les exemples de problème présents dans le fichier d'en-tête sont reproduits ici dans le même ordre.





V.1 Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef uint64_t board;
typedef bool map[64];
typedef char constraint[8];

struct problem {
    board monsters;
    board treasures;
    constraint rows;
    constraint columns;
};
typedef struct problem problem;

problem p1 = {
    .monsters = 0x8400010000000020,
    .treasures = 0x20000,
    .rows = { 7,3,4,1,7,1,6,3 },
    .columns = { 4,4,2,6,2,3,4,7 }
};

problem p2 = {
    .monsters = 0xa00008000000000,
    .treasures = 0x2800,
    .rows = { 5,2,2,3,6,0,6,1 },
    .columns = { 5,3,4,3,5,1,3,1 }
};

problem p3 = {
    .monsters = 0x8000008001000081,
    .treasures = 0x4000000000000,
    .rows = { 6,0,4,1,5,3,3,4 },
    .columns = { 4,4,1,3,6,3,4,1 }
};

problem p4 = {
    .monsters = 0x8000800000000100,
    .treasures = 0x2000,
    .rows = { 2,3,4,3,7,0,4,0 },
    .columns = { 1,4,1,5,1,3,2,6 }
};

problem p5 = {
    .monsters = 0x20001,
    .treasures = 0x80000,
    .rows = { 0,7,3,3,3,5,2,5 },
    .columns = { 7,2,6,1,3,2,6,1 }
};
```

```
problem p6 = {
    .monsters = 0x802080208020,
    .treasures = 0x40000,
    .rows = { 6,3,3,3,5,3,4,3 },
    .columns = { 5,3,2,4,6,4,1,5 }
};

problem p7 = {
    .monsters = 0x80010008,
    .treasures = 0x800000000000000,
    .rows = { 1,4,2,6,2,3,3,1 },
    .columns = { 1,5,2,2,6,1,4,1 }
};

problem p8 = {
    .monsters = 0x4200801000800042,
    .treasures = 0x0,
    .rows = { 3,6,0,5,4,0,6,3 },
    .columns = { 4,2,4,2,3,4,2,6 }
};

problem p9 = {
    .monsters = 0x40080,
    .treasures = 0x2000000000000000,
    .rows = { 4,1,4,2,6,2,3,2 },
    .columns = { 5,2,1,2,5,4,2,3 }
};

uint64_t mask_column(int i)
{
    return 0x0101010101010101 << i;
}

board map_to_board(map m)
{
    board b = 0;
    for(int i = 0; i < 64; i++)
    {
        b = b * 2;
        if (m[i])
            b = b + 1;
    }
    return b;
}

void board_to_map(board b, map m)
{
    for(int i = 0; i < 64; i++)
    {
        m[i] = b % 2 == 1;
        b = b / 2;
    }
}


int popcount_slow(uint64_t v)
{
    int s = 0;
    while(v != 0)
    {
        if (v % 2 == 1)
            s = s+1;
        v = v/2;
```

```
    }
    return s;
}

int popcount(uint64_t v)
{
    uint64_t mask_1 = 0x5555555555555555;
    uint64_t mask_2 = 0x3333333333333333;
    uint64_t mask_3 = 0x0f0f0f0f0f0f0f0f;
    uint64_t mask_4 = 0x00ff00ff00ff00ff;
    uint64_t mask_5 = 0x0000ffff0000ffff;
    uint64_t mask_6 = 0x00000000ffffffff;

    v = (v & mask_1) + ((v & (mask_1 << 1)) >> 1);
    v = (v & mask_2) + ((v & (mask_2 << 2)) >> 2);
    v = (v & mask_3) + ((v & (mask_3 << 4)) >> 4);
    v = (v & mask_4) + ((v & (mask_4 << 8)) >> 8);
    v = (v & mask_5) + ((v & (mask_5 << 16)) >> 16);
    v = (v & mask_6) + ((v & (mask_6 << 32)) >> 32);

    return v;
}

#define popcount __builtin_popcountll

int enum_popcount_byte[256];

void print_board(problem *p, board b)
{
    for (int i = 0; i < 64; i++)
    {
        if (((b >> i) & 1) != 0)
            putchar('#');
        else if (((p->monsters >> i) & 1) != 0)
            putchar('M');
        else if (((p->treasures >> i) & 1) != 0)
            putchar('T');
        else putchar('.');

        if (i % 8 == 7) printf("\n");
    }
}

uint64_t bit(int pos)
{
    return (uint64_t) 1 << pos;
}

uint64_t neighbours(int pos)
{
    uint64_t mask = 0;
    if (pos % 8 != 0)
        mask |= bit(pos - 1);
    if (pos % 8 != 7)
        mask |= bit(pos + 1);
    if (pos > 7)
        mask |= bit(pos - 8);
    if (pos < 56)
        mask |= bit(pos + 8);
    return mask;
}

int count_non_wall(problem *p, board b, int pos)
{
```

```
    int count = 0;
    uint64_t mask = neighbours(pos);
    return popcount(~b & mask);
}

bool empty(problem *p, board b, int pos)
{
    board fb = b | p->monsters | p->treasures;

    return (fb & bit(pos)) == 0;
}

bool treasure(problem *p, int pos)
{
    return p->treasures & bit(pos);
}

bool monster(problem *p, int pos)
{
    return p->monsters & bit(pos);
}

int enum_byte[] = { 0, 1, 2, 4, 8, 16, 32, 64, 128, 3, 5, 6, 9, 10, 12, 17, 18, 20, 24, 33, 34, 36,
int enum_byte_idx[] = { 0, 1, 9, 37, 93, 163, 219, 247, 255, 256 };

bool test_columns_overload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) > p->columns[col])
            return true;
    return false;
}

bool test_columns_underload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) < p->columns[col])
            return true;
    return false;
}

bool test_no_deadend_upto_row(problem *p, board b, int row)
{
    for(int i = 0; i < 8*row; i++)
    {
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
    }
    return false;
}

bool test_trapped_monster(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
        if (monster(p,i) && count_non_wall(p, b, i) == 0)
            return true;
    return false;
}

bool test_deadends(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
    {
        // Empty cell must not be a dead-end
```

```
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
        // Monsters must be a dead-end
        if (monster(p,i) && count_non_wall(p, b, i) != 1)
            return true;
    }

    return false;
}

bool test_treasure_rooms(problem *p, board *b)
{
    for(int i = 0; i < 64; i++)
    {
        if(treasure(p,i))
        {
            int x = i % 8;
            int y = i / 8;
            int minx = x - 2;
            if (minx < 0) minx = 0;
            int miny = y - 2;
            if (miny < 0) miny = 0;
            // starting point of the treasure room
            //    xxx
            //    xTx
            //    xxx
            bool found = false;
            uint64_t room = 0;
            int sx, sy;

            for(sy = miny; sy <= y; sy++)
            {
                for(sx = minx; sx <= x; sx++)
                {
                    room = (uint64_t)0x070707  << (sx + 8*sy);
                    if (popcount(~(*b) & room) == 9)
                    {
                        found = true;
                        break;
                    }
                }
                if(found) break;
            }

            if (!found) return true;
            uint64_t border = 0;
            if (sy > 0)
                border |= (uint64_t)0x07 << (sx+8*(sy-1));
            if (sy+2 < 7)
                border |= (uint64_t)0x07 << (sx+8*(sy+3));
            if (sx > 0)
                border |= (uint64_t)0x010101 << (sx-1+8*sy);
            if (sx+2 < 7)
                border |= (uint64_t)0x010101 << (sx+3+8*sy);

            int exit = popcount(~(*b) & border);
            if (exit != 1)
                return true;

            *b = *b | room;
        }
    }

    return false;
```

```
}

bool test_empty_squares(problem *p, board b)
{
    uint64_t mask_square = 0x0303;
    for(int i = 0; i < 7; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if((b & mask_square) == 0)
                return true;
            mask_square *= 2;
        }
        mask_square *= 2;
    }

    return false;
}

void solve(problem *p, int row, board b)
{
    // Unsolvable from this board
    if (test_columns_overload(p, b))
        return;
    if (test_no_deadend_upto_row(p, b, row))
        return;
    if (test_trapped_monster(p, b))
        return;

    // Board is filled
    if(row == 8)
    {
        // Validate columns count
        if (test_columns_underload(p, b))
            return;

        // Full board with monsters
        board fb = b | p->monsters;

        if (test_treasure_rooms(p, &fb))
            return;
        // Every treasure room is now filled in fb

        if (test_empty_squares(p, fb))
            return;

        printf("Solution %lx\n", b);
        print_board(p, b);

        return;
    }

    /*
    // Slower because of branching
    for(int i = 0; i < 256; i++)
        if (enum_popcount_byte[i] == p->rows[row])
        {
        uint64_t move = (uint64_t)i << (8*row);

        // ensure there's no overlap with walls/monsters/treasures
        if ((move & (p->monsters | p->treasures)) != 0)
            continue;

        solve(p, row+1, b + move);
```

```
        }
    */

    int tgt = p->rows[row];
    for(int v = enum_byte_idx[tgt]; v < enum_byte_idx[tgt+1]; v++)
    {
        uint64_t move = (uint64_t)enum_byte[v] << (8*row);

        // ensure there's no overlap with walls/monsters/treasures
        if ((move & (p->monsters | p->treasures)) != 0)
            continue;

        solve(p, row+1, b + move);
    }
}

int main(void)
{
    /*
    map monsters = {
        0, 0, 1, 0, 1, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 1, 0, 0,
    };
    */

    map monsters = { [5] = 1 };
    problem test = {
       .rows = { 6, 2, 5, 3, 2, 5, 2, 6 },
       .columns = { 6, 2, 4, 3, 4, 4, 2, 6 }
    };
    test.monsters = map_to_board(monsters);

    for(int i = 0; i < 256; i++)
        enum_popcount_byte[i] = popcount(i);

    problem *problems[] = {
        &p1, &p2, &p3,
        &p4, &p5, &p6,
        &p7, &p8, &p9
    };

    for (int i = 0; i < 9; i++)
    {
        printf("Solving problem %d\n", i);
        print_board(problems[i], 0);
        solve(problems[i], 0, 0);
    }
}
```