Robotique
CRC
Robotics

CRC

COMPÉTITION DE PROGRAMMATION

TAK TIK 2025

LIVRET DE PROBLÈMES

BLOC B

QUELQUES NOTES

- Les règles complètes sont dans la section 4 du livret des règlements
- Vous avez jusqu'au **15h59** pour remettre vos solutions
- N'hésitez pas à nous poser des questions et à communiquer avec les membres de votre équipe de programmation. Ce n'est pas un examen!
- **On vous donne des fichiers modèles faciles à utiliser pour votre code. Vous devez les utiliser.**

UTILISATION DU FICHIER MODÈLE

- Tous les fichiers possèdent une fonction solve() dans lequel vous devez résoudre le problème. **NE CHANGEZ PAS LE NOM DE CETTE FONCTION!**
- Pour rouler vos tests, dans un terminal (nous vous conseillons l'utilisation de vscode) écrivez la ligne de commande: `pytest` pour rouler les tests. Si vous faites la commande `pytest` dans le répertoire contenant tous les fichiers, tous les tests seront exécutés.
- Pour rouler les tests d'un problème spécifique, naviguez dans le répertoire de la question et exécutez la commande `pytest` pour rouler les tests du problème. Vous pouvez faire la même chose pour une section entière!

STRUCTURE

Chaque problème contient une petite mise en situation comme celle-ci expliquant les fondements du problème et donnant les bases nécessaires pour résoudre celui-ci. Chaque problème préparatoire contient aussi la répartition des points pour le problème.

Spécifications d'entrée et de sortie:

Dans cette section, on retrouve les caractéristiques des entrées qui peuvent être fournies au code en question ainsi que les critères attendus pour les sorties du programme.

Exemple d'entrée et de sortie:

Dans cette section se trouve un exemple d'entrée (parfois constitué lui-même de plusieurs sous-exemples) pour que vous puissiez tester votre code. L'exemple de sortie donne donc la réponse attendue pour cette entrée.

Explication de la première sortie:

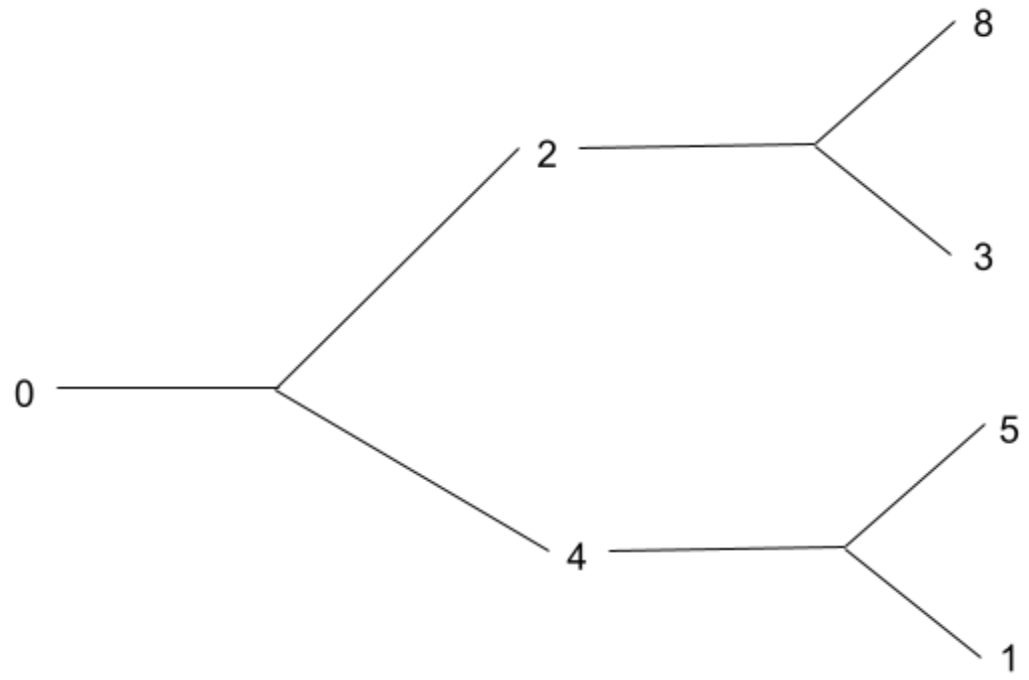
Si le problème n'est toujours pas clair après la mise en situation, l'explication de la première sortie sert parfois à démêler le tout en expliquant comment la première entrée est traitée et en montrant le chemin menant à cette réponse.

Table des matières

MODES DE TRANSPORT (MT)

MT1: Choisis ton chemin! (30 points)

Vous êtes un opérateur de train et vous souhaitez faire plaisir au plus de passagers en un seul trajet. Vos choix de chemin sont une série d'embranchements ou vous pouvez aller à droite ou à gauche. C'est votre seule décision, aller vers la gauche ou la droite et ces embranchements décideront par quelles gares vous allez passer. Vous devez donc trouver le chemin qui amène le plus de passagers possible à leur destination. Vous recevrez le nombre de passagers qui descendent à chaque station dans une liste qui représente un arbre. Voici une représentation rapide des choix que vous pouvez faire et la liste de passagers associée.
[0, 2, 4, 8, 3, 5, 1]



Vous devrez trouver le chemin qui apporte le plus grand nombre de passagers à leur destination. Dans ce cas, le meilleur chemin est gauche suivi de gauche encore pour pouvoir apporter 10 passagers à leur destination. La sortie serait ['L', 'L']. Les direction sont L pour gauche et R pour droite.

Spécifications d'entrée:

En entrée vous recevrez:

- ***passengers:*** le nombre de passagers qui veulent aller à chaque destination.

Spécifications de sortie:

En sortie vous devrez fournir: un array des directions à prendre aux embranchements

Exemple d’entrée:

[0, 2, 4, 8, 3, 5, 1]
[0, 2, 4, 8, 3, 5, 1, 1, 2, 2, 3, 9, 8, 7, 5]
[0, 6, 4, 2, 5, 8, 1]

Exemple de sortie:

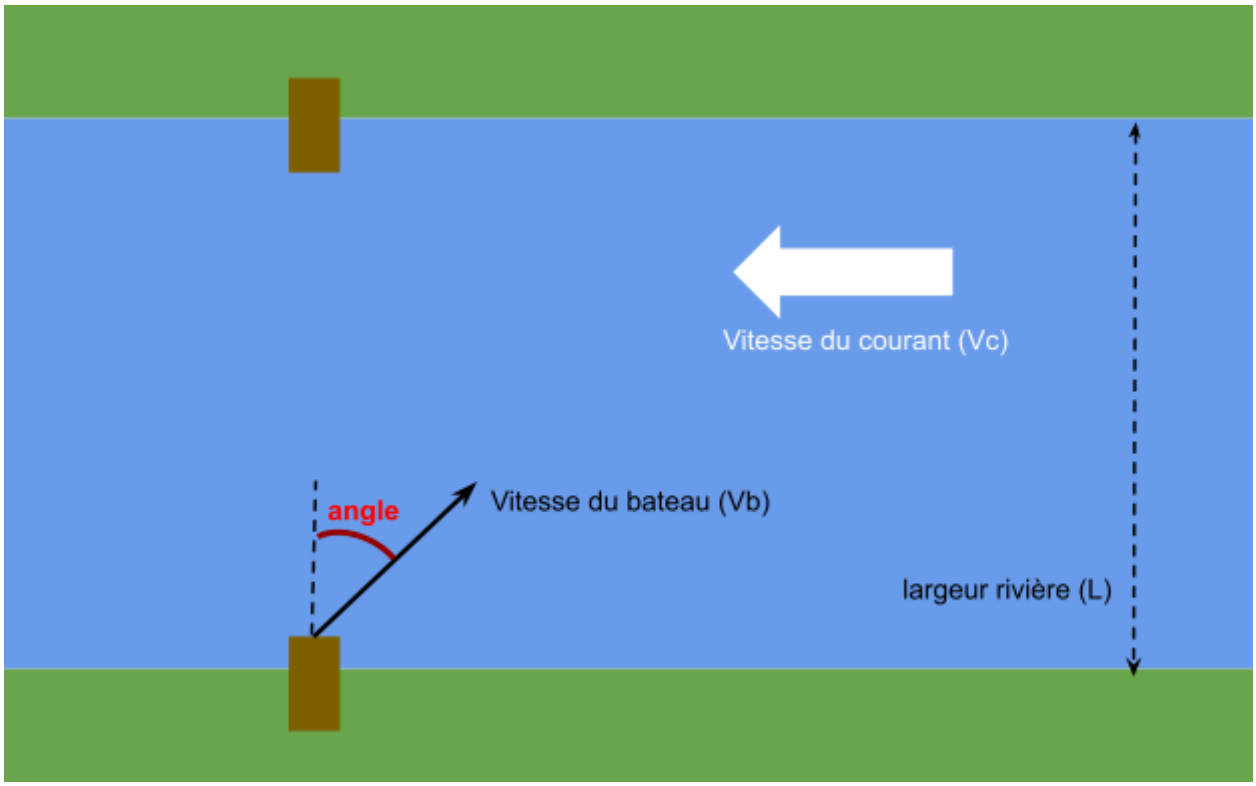
```
['L', 'L']
['R', 'L', 'L']
['R', 'L']
```

Exemple de la première sortie:

Voir l’explication dans la description.

MT2: Traverser la rivière (15 points)

Vous conduisez une petite barque qui fait traverser des touristes de bord en bord du fleuve St-Laurent. Pour vous rendre au quai de l'autre côté de la rive en face de vous, vous devez vous assurer de prendre en compte le courant. Ainsi, nous allons vous donner la vitesse du bateau et la vitesse du courant et vous devrez trouver le temps nécessaire pour effectuer la traversée.



Voici les formules dont vous aurez besoin pour trouver le temps de la traversée. Tout d'abord, comme les deux quais sont alignés, vous devrez trouver l'angle du bateau pour contrer le courant. Ceci nous permet de trouver la composante de la vitesse qui va contre le courant et donc l'angle que le bateau doit prendre.

$$angle = arcsin(\frac{V_c}{V_b})$$

Nous pouvons maintenant trouver la composante de la vitesse qui est dans le sens de la traversée.

$$V_\perp = V_b \cdot cos(angle)$$

Et maintenant avec la vitesse et la distance nous pouvons trouver le temps nécessaire à la traversée.

$$t = \frac{L}{V_\perp}$$

Assurez-vous de donner le temps de la traversée en minutes avec le nombre de secondes arrondies à la seconde près.

Spécifications d'entrée:

En entrée vous recevrez:

- ***Vb:*** un *float* de la vitesse du bateau en m/s
- ***Vc***: un *float* de la vitesse du courant en m/s
- **L**: un *int* de la largeur du fleuve à traverser en mètres

Spécifications de sortie:

En sortie vous devrez fournir:

- un string du temps nécessaire à la traversée avec le format XXmXXs

Exemple d'entrée:

```
Vb = 4.6 Vc = 3.9 L = 1350
Vb = 5.1 Vc = 4.7 L = 2250
Vb = 3.1 Vc = 1.7 L = 800
```

Exemple de sortie:

```
09m13s
18m56s
05m09s
```

Explication de la première sortie:

Pour le premier exemple avec `Vb = 4.6, Vc = 3.9 et L = 1350.` Nous pouvons commencer par calculer l'angle que le bateau devra avoir pour contrer le courant.

$$angle = arcsin(\frac{V_c}{V_b})$$

$$angle = 1.0118721573923892 radians$$

Avec l'angle, on peut maintenant aller chercher la composante de la vitesse dans le sens de la traversée.

$$V_\perp = V_b \cdot cos(angle)$$

$$V_\perp = 4.6 m/s \cdot cos(1.0118721573923892 radians)$$

$$V_\perp = 2.4392621835300927 m/s$$

Maintenant que nous avons la vitesse en direction de l'autre rive, nous pouvons calculer le temps de la traversée.

$$t = \frac{1350m}{2.4392621835300927 m/s}$$

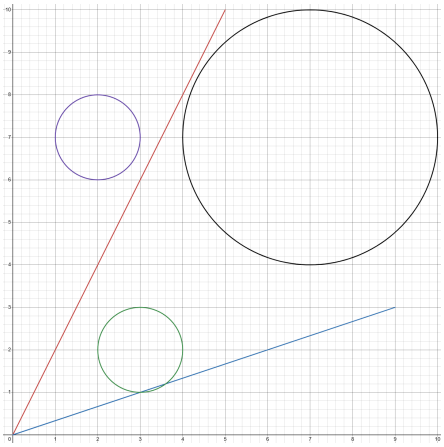
$$t = 553.4460416412819 s$$

t = 09m13s

MT3: On passe en hyperespace! (90 points)

Vous êtes en pleine bataille pour libérer la galaxie lorsque la retraite est sonnée à l'amorce d'une défaite écrasante. Transportant un passager de plus haute importance, vous devez passer en hyperespace pour espérer vous en sortir, mais votre ordinateur de navigation vous permettant d'accéder aux chemins sécurisés normalement employés (sécurisés, c'est-à-dire qui ne font pas rentrer directement dans un astre) est brisé. Vous devrez donc transmettre un algorithme de secours pour calculer comment vous rendre à un certain point par hyperespace suffisamment rapidement.

Votre vaisseau et votre destination seront placés sur un plan cartésien qui va de 0 à 10 à la fois en x et en y. Le point d'origine (0,0) est en bas à gauche du plan et le point (10,10) est en haut à droite complètement du plan. **Votre vaisseau peut se déplacer entre n'importe quelle combinaison de deux points en ligne droite tant qu'il n'entre pas en collision avec un astre.** Par exemple,



Dans le plan ci-dessus, le déplacement en rouge entre les points (0,0) et (5,10) est permis comme il est en ligne droite et ne rencontre aucun astre alors que le déplacement en bleu des points (0,0) à (9,3) n'est pas permis comme il entre en collision avec l'astre vert. Une droite parfaitement tangente à un des cercles (c'est-à-dire qu'elle n'a qu'un seul point d'intersection avec cet astre, par exemple (2,1) à (4,1) est tangente au cercle vert) est considérée comme invalide et collisionne avec l'astre.

Les astres seront tous des cercles et vous seront donnés sous la forme (point central, rayon). Par exemple, le cercle vert est décrit par ((3,2),1). Pour former un cercle (faites une série de points!) à partir de ces données, il suffit d'utiliser l'équation d'un cercle. En utilisant le point central (a,b) et le rayon r, l'équation devient:

$$(x-a)^2+(y-b)^2=r^2$$

Vous devrez donner une série de points commençant par les coordonnées du vaisseau et finissant par les coordonnées de la destination, contenant tous les points où le vaisseau change de trajectoire entre, c'est-à-dire que le vaisseau traverse en ligne droite entre un point et son suivant dans cette liste, puis le suivant devient le point de départ du prochain déplacement en ligne droite jusqu'au 3e point dans la liste, et ainsi de suite jusqu'à destination. **Vous devrez trouver un itinéraire valide qui est à moins de 5% de la distance optimale,**

c'est-à-dire que si le meilleur chemin est 8 unités de long, votre solution ne pourra pas dépasser 8.4 unités. Il est utile de rappeler la formule de distance:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

Où Δ représente une différence entre deux coordonnées, ici les différences entre les coordonnées en x et en y. **Il est possible que le trajet optimal ne prenne que le point de départ du vaisseau et sa destination si c'est une ligne droite valide. Le trajet optimal ne fera jamais plus de 5 déplacements (ou changements de trajectoire), donc votre série de points n'aura jamais besoin d'être plus de 6 points de long, mais sera tout de même valide avec plus de points si les contraintes de distance et de collision sont respectées.**

Note finale: Il est possible que, par erreur de notre part, vous trouviez un meilleur chemin que celui que nous avons utilisé pour calculer la distance optimale d'un exemple. Votre itinéraire sera évidemment bon, mais nous vous recommandons de vérifier qu'il ne rencontre aucun astre rapidement sur papier ou Desmos pour vous assurer qu'il soit valide.

Spécifications d'entrée:

En entrée vous recevrez:

- ***vaisseau:*** un tuple de deux entiers, soit les coordonnées initiales du vaisseau
- ***destination:*** un tuple de deux entiers, soit les coordonnées du point à atteindre
- ***astres:*** un tuple en 3D d'entiers contenant les astres à éviter de la forme ((a,b),r) chacun

Spécifications de sortie:

En sortie vous devrez fournir:

- une liste de tuples d'**entiers**, soit la liste de coordonnées décrivant l'itinéraire à suivre

Exemple d'entrée:

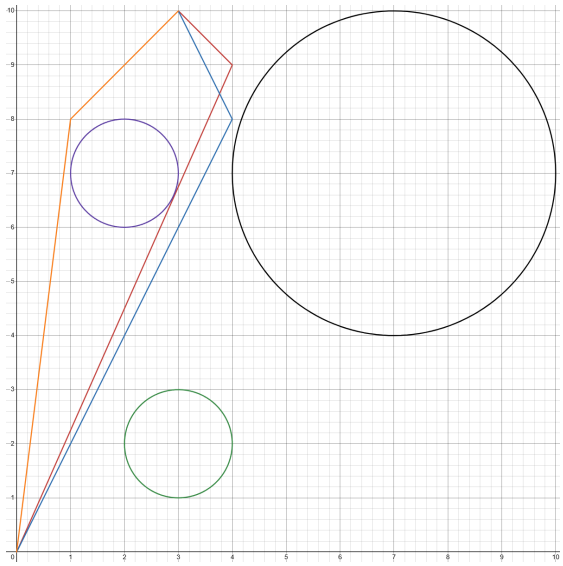
```
(0,0) (3,10) (((3,2),1),((2,7),1),((7,7),3))
(2,0) (9,4) (((6,2),2),((3,7),3),((8,6),1))
(6,9) (9,6) (((4,5),4),((9,8),1))
(2,9) (10,1) (((7,2),2),((4,7),2),((7,7),1),((9,6),1),((2,2),2))
```

Exemple de sortie:

[(0,0),(1,8),(3,10)] (distance = ~10.89)
[(2,0),(4,3),(6,5),(9,4)] (distance = ~9.60)
[(6,9),(9,6)] (distance = ~4.24)
[(2,9),(1,5),(8,4),(9,3),(10,1)] (distance = ~14.84)

Explication de la première sortie:

Comme le chemin direct de (0,0) à (3,10) passe quasiment en plein milieu de l'astre violet, les options pour le meilleur chemin sont pas mal de le contourner de justesse sans y toucher. Ici, seulement 3 très bons chemins sont illustrés mais il y en a plus de possibles, bien évidemment.



Le chemin bleu fait une distance de ~11.18 unités, le rouge fait ~11.26 unités et le jaune fait ~10.89 unités. Le jaune est donc le chemin le plus court et la meilleure solution est [(0,0),(1,8),(3,10)]. Les deux autres chemins auraient été acceptés, ceci dit, comme leur distance est inférieure à 10.89*1.05=~11.43 unités.

MT4: Comment sortir du stationnement (55 points)

Vous habitez à Montréal et vous avez malheureusement besoin de prendre votre voiture tous les jours pour travailler dans le centre-ville. Vous avez un stationnement dans lequel vous pouvez mettre votre voiture, mais sortir votre votre voiture est toujours un défi de taille donc vous souhaitez coder un programme pour vous aider à sortir du stationnement sans accrocher d'autres véhicules.
Vous aurez donc une grille dans laquelle, votre véhicule sera marqué par des @ alors que les autres véhicules seront marqués par les lettres A, B, C, … Les véhicules ont au moins 2 unités de long et se déplacent toujours dans le sens de leur longueur. Ainsi, certains véhicules pourront seulement se déplacer de gauche (L) à droite (R) alors que d'autres pourront seulement se déplacer de haut (U) en bas (D). Pour indiquer un déplacement, vous donnez le nom du véhicule et sa direction (ex: BU) et chaque déplacement est d'une seule case dans la direction choisie. Pour sortir du stationnement, il faut que votre véhicule ait sa portion avant qui soit entre les deux murs de la sortie. Aucun véhicule ne peut aller sur une case occupée par un autre véhicule sinon vous aurez une collision. Vous devez donc trouver une séquence de déplacements pour faire sortir votre véhicule qui ne dépasse pas 100 opérations.

N.B. Comme plusieurs solutions sont possible, votre code sera évalué directement

Spécifications d'entrée:

En entrée vous recevrez:

- ***grid:*** un *array* de *string* qui représente le stationnement avec tous les véhicules

Spécifications de sortie:

En sortie vous devrez fournir:

- un *array* de *string* des opérations à effectuer pour sortir du stationnement

Exemple d'entrée:

```
'########',
'#AA   B#',
'#C  D B#',
'#C@@D B ',
'#C  D  #',
'#E   FF#',
'#E GGG #',
'########'
```

```
'########',
'#A  BBB#',
'#A  C D#',
'#@@ CED ',
'#FFF ED#',
'#  G HH#',
'#IIGJJ #',
'########'

'########',
'#      #',
'#      #',
'# @@A   ',
'# BBA C#',
'# D A C#',
'# DEE C#',
'########'
```

Exemple de sortie:

```
Vous pouvez avoir d'autres sorties valides que celles indiquées
['AR', 'CU', 'EU', 'GL', 'GL', 'FL', 'FL', 'FL', 'DD', 'DD',
'BD', 'BD', 'BD', '@R', '@R', '@R', '@R']

['@R', 'FR', 'AD', 'AD', 'AD', '@L', 'BL', 'BL', 'BL', 'EU',
'DU', 'FR', 'FR', 'GU', 'GU', 'GU', 'IR', 'AD', 'HL', 'HL',
'HL', 'FL', 'FL', 'FL', 'CD', 'CD', 'ED', 'ED', 'DD', 'DD',
'DD', 'BR', 'BR', 'BR', 'GU', '@R', '@R', '@R', '@R', '@R']

['AU', 'AU', 'CU', 'CU', 'CU', 'ER', 'ER', 'BR', 'BR', 'BR',
'AD', 'AD', 'AD', '@R', 'DU', 'DU', 'DU', 'DU', '@L', 'AU',
'AU', 'AU', 'BL', 'BL', 'BL', 'EL', 'EL', 'EL', 'AD', 'AD',
'AD', 'CD', 'CD', 'CD', '@R', '@R', '@R', '@R']
```

Explication de la première sortie:

Dans la première grille, on commence par tasser l'auto en haut (A) pour faire monter les deux autos à gauche chacune d'une case (C et E). Ainsi, on peut mettre complètement à gauche l'auto d'en bas (G) et tasser aussi vers la gauche celle d'au-dessus (F). Maintenant, on peut descendre l'auto D jusqu'en bas et l'auto B aussi jusqu'en bas pour libérer le passage pour que notre auto puisse sortir du stationnement! **D'autres solutions peuvent être valides que celles montrées dans les exemples.**

MT5: Atterrissage d'avion (25 points)

Vous faites un système de sécurité pour aider à l' atterrissage des avions. Vous calculez l'altitude et l'angle d'approche pour vous assurer d'être toujours sur la bonne trajectoire. L'angle d'approche idéal est de 3 degrés par rapport au point de contact avec la piste. Nous allons fournir votre distance projetée sur le sol entre l'avion et le point d'atterrissage ainsi que votre vitesse. Malheureusement, le vrai monde n'est pas parfait, donc après quelques secondes ininterrompues de descente, vous aurez un coup de vent qui va remonter votre avion. Vous devez déterminer si la nouvelle trajectoire est encore parfaite (entre 3 et 4 degrés), correcte (entre 4 et 5 degrés) ou dangereuse (plus de 5 degrés).

Si l'angle après le coup de vent est:

- entre 3 et 4 degrés, vous devez retourner: "Perfect"
- entre 4 et 5 degrés. vous devez retourner: "Okay"
- plus de 5 degrés, vous devez retourner: "Danger"

Spécifications d'entrée:

En entrée vous recevrez:

- ***distance:*** la distance en *int* de l'avion projetée sur le sol entre sa position actuelle et le point d'atterrissage.
- ***speed:*** la vitesse en *int* en m/s de l'avion en direction du point d'atterrissage.
- ***time:*** un *int* du temps en seconde entre la position initiale et la turbulence.
- ***offset:*** un *int* du nombre de mètres par lequel l'avion est soulevé lors de la turbulence.

Spécifications de sortie:

En sortie vous devrez fournir:

- un *string* du statut de l'avion pour l'atterrissage juste après la turbulence.

Exemple d'entrée:

```
distance=850, speed=60, time=8, offset=10
distance=720, speed=65, time=9, offset=5
distance=880, speed=55, time=4, offset=11
```

Exemple de sortie:

```
"Okay"
"Danger"
"Perfect"
```

Explication de la première sortie:

Pour le premier exemple, nous avons une distance au sol de 850 mètres ce qui nous donne avec un angle de 3 degrés un distance directe dans les airs de 851.1664940982328 mètres. Nous pouvons donc commencer la descente pour 8 secondes avec une vitesse de 60 m/s ce qui nous amène à une distance en vol de 371.1664940982328 mètres. Cette distance directe peut aussi être exprimée comme 370.65782331780446 mètres en distance au sol et 19.42535339397198 mètres d'altitude parce qu'avant la turbulence, nous gardons une approche de 3 degrés. Avec notre turbulence, on monte de 10 mètres de plus et maintenant, notre angle est rendu à 4.539011506371929 degrés ce qui est considéré comme "Okay".

4 ÉLÉMENTS (EL)

EL1: Bourrasques de vent (45 points)

Vous êtes responsable d'une ferme d'éoliennes et devez vous assurer qu'aucune d'entre elles ne sont coincées. Normalement, lorsque le vent souffle sur les pales de l'éolienne, celles-ci devraient tourner dans la direction que le vent les entraîne. Pour simplifier la tâche, nous allons représenter les éoliennes comme étant des coins de 90 degrés sur une grille de n par n et les bourrasques de vent qui vous seront données viendront uniquement des 4 directions cardinales (vers le haut, la droite, le bas ou la gauche). Au début de la journée, toutes les éoliennes commencent avec leurs pales pointant vers la droite et vers le bas. Leur point de rotation sera fixé aux coordonnées cartésiennes de nombres entiers (enter image pour clarifier). lorsqu'une bourrasque de vent passe, l'éolienne doit tourner de 90 degrés si une de ses pales est dans la trajectoire du vent. Les bourrasques seront données sous la forme +/- n x/y. le signe et l'axe déterminent la direction du vent (+x = vers la droite, +y = vers le haut) tandis que "n" représente la rangée ou colonne ou le vent souffle. Si le vent souffle en X, la bourrasque passera entre la rangée "n" et "n-1". Si le vent souffle en y, la bourrasque passera entre la colonne "n" et "n-1".

(coin (0, 0) est en bas à gauche)

Voici les valeurs ascii des caractères à utiliser pour les éoliennes: ASCII 191, 192, 217, 218

Spécifications d'entrée:

En entrée vous recevrez:

- ***n:*** le nombre de turbines par rangée
- ***gusts:*** un *array* de *strings* des bourrasques de vents

Spécifications de sortie:

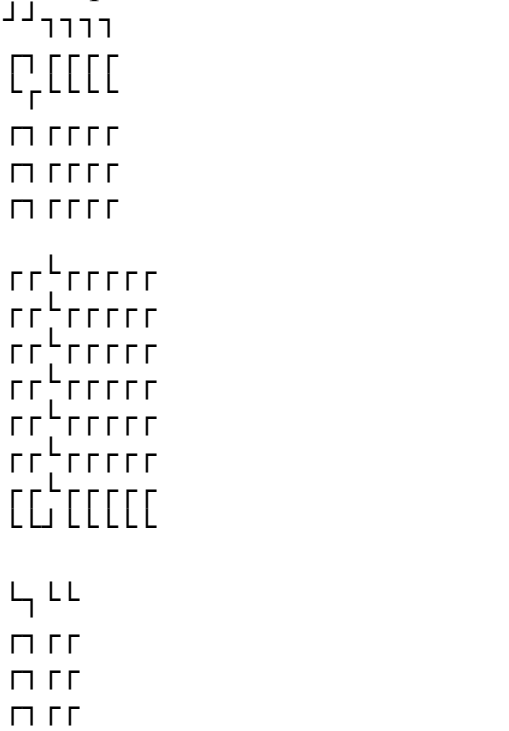
En sortie vous devrez fournir:

- un array de strings qui correspond à l'état final des turbines

Exemple d'entrée:

```
6, (3x, -2y, -5x, 0y)

8, (0x, -0y, 8x, 3y)

4, (-2y, 3x, -2y)
```

Exemple de sortie:



Explication de la première sortie:

```
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

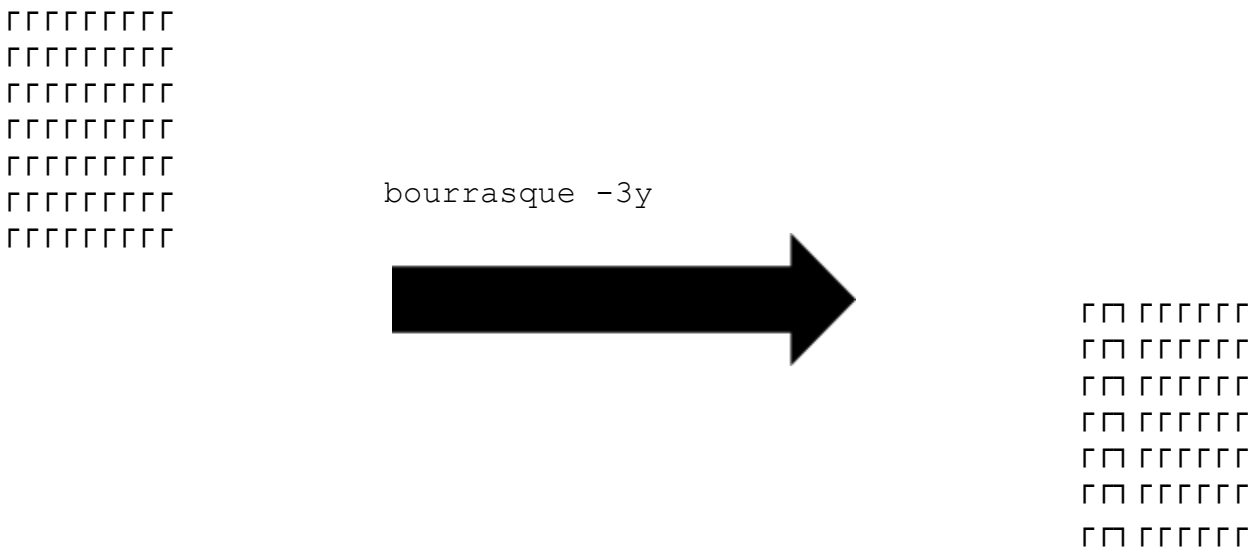
bourrasque 5x

```
[[[[[[[[[
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

```
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
┌┌┌┌┌┌┌┌┌
```

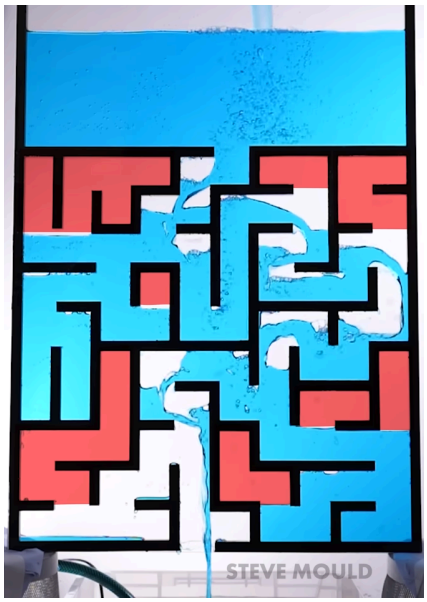
bourrasque 3y

```
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
┌┌└┌┌┌┌┌┌
```


bourrasque -3y

EL2: Labyrinthe d'eau (80 points)

Vous avez vu la vidéo de Steve Mould sur youtube (https://youtu.be/81ebWToAnvA?si=VSnPgD3V1bsac7XE&t=252) qui parle de résoudre un labyrinthe avec de l'eau. Par contre, contrairement aux simulations sans air, la simulation réelle montre un phénomène différent. Bien que le labyrinthe soit résolu, plusieurs poches d'air ne sont pas remplies par l'eau! C'est un phénomène bien intéressant, on remarque que l'eau ne peut pas déloger l'air pris dans ces sections surélevées, sauf si elles amènent vers la sortie ou l'air peut s'échapper. Vous devrez donc prendre un labyrinthe en paramètre et indiquer tous les endroits ou l'eau pourrait se retrouver. Ce sont donc tous les endroits qui ne sont PAS en rouge dans la simulation. Les murs du labyrinthe sont des # et vous devez mettre des points (.) à tous les endroits qui peuvent contenir de l'eau.



Spécifications d'entrée:

En entrée vous recevrez:

- ***labyrinth:*** un *array* de *strings* qui composent le labyrinthe

Spécifications de sortie:

En sortie vous devrez fournir:

- un array de strings qui correspond aux labyrinthe avec les endroits qui peuvent contenir de l'eau qui sont indiqués

Exemple d'entrée:

```
'########## #########',
'# #      #   #   #   #',
'# # # ### ##### # ###',
'# # # #   #     #   #',
'# # ### ### ### ### #',
'#         # #   #   #',
'##### ### # # ### # #',
'#     # # # #     # #',
'# ### # # # ####### #',
'#   #   #   #   #   #',
'# # ######### # # ###',
'# # # #     # #   # #',
'# # # # ### # ##### #',
'# # # #   #   #   # #',
'##### ### ### # ### #',
'#     #   # #       #',
'# ##### # # ####### #',
'#   #   # #   #     #',
'### # ### ### # ### #',
'#       # ###   #   #',
'######### ##########'


'########## ########',
'# ### ##   ##   ###',
'# # # ## #### # ###',
'#             #   #',
'################ #'


'### #######',
'#   #   # #',
'# ### # # #',
'#     #   #',
'######### #',
'#    #### #',
'# ##      #',
'# #########'
```

Exemple de sortie:

```
'###########.#########',
'# #      #...#    #    #',
'# # # ###.##### # ###',
'# # # #...#.....#    #',
'# # ###.###.###.### #',
'#.........#.#...#...#',
'#####.###.#.#.###.#.#',
'#.....# #.#.#.....#.#',
'#.###.# #.#.#######.#',
'#...#...#...#...#...#',
'#.#.#########.#.#.###',
'#.#.# #.....#.#...# #',
'#.#.# #.###.#.##### #',
'#.#.# #...#...#    # #',
'##### ###.###.# ### #',
'#      #...# #.......#',
'# #####.#.# #######.#',
'#    #...#.#    #.....#',
'### #.###.### #.###.#',
'#.......#.###...#...#',
'#########.###########'
```

```
'##########.########',
'# ### ##...##...###',
'# # # ##.####.#.###',
'#............#...#',
'##################.#'
```

```
'###.#######',
'#...#...# #',
'#.###.#.# #',
'#.....#...#',
'#########.#',
'#....####.#',
'#.##......#',
'#.#########'
```

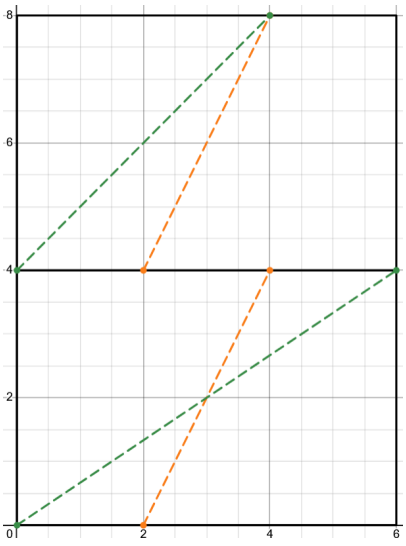
Explication de la première sortie:

Le premier labyrinthe est celui de la vidéo et donc de l’image!

EL3: Craquage de terre (70 points)

En terrain très sec, la terre à tendance à craquer. Un bon jour, en se promenant en tel terrain, Elrik se demande combien de morceaux différents sont créés par les craques dans une aire donnée.

Nous vous donnerons donc les coordonnées des coins inférieur gauche et supérieur droit d'un rectangle dans un plan cartésien représentant l'aire à découper. Les craques seront approximées par des segments de droite reliant deux points placés sur le périmètre du rectangle. Une craque passant par un morceau, peu importe où, le découpe en deux plus petits morceaux. La première craque séparera donc la zone en deux morceaux. La craque suivant "créera" un nouveau morceau si elle ne croise pas la craque précédente, ou 2 morceaux si elle la croise cette dernière.



Dans le cas supérieur, la 2e craque en vert n'intersecte pas avec la 1re **(à l'intérieur du rectangle)**. Elle ne "crée" qu'un seul autre morceau pour un total de 3.

Dans le cas inférieur, les deux droites se croisent dans le rectangle, "créant" 2 nouveaux morceaux, amenant le total à 4.

La formule pour morceaux générés par une nouvelle craque est donc:

$$nouveaux\ morceaux = \#croisements + 1$$

Il est important de noter que le nombre de croisements ne correspond pas toujours au nombre de droites croisées. Dans le cas où une craque croise ce qui est déjà un point d'intersection de 2 autres craques ou plus, cela ne compte que pour un croisement même si plusieurs craques sont croisées à ce point. Vous devrez prendre compte de ceci dans votre code. Un visuel de ce cas particulier vous sera démontré dans l'explication de la première sortie.

N.B. Vous pouvez utiliser des librairies externes pour faire l'intersection des droites.

Spécifications d'entrée:

En entrée vous recevrez:

- ***coins:*** tuple 2D d'entiers contenant les deux paires de coordonnées désignant les coins du rectangle
- ***craques:*** liste de tuples 2D d'entiers où chaque tuple contient deux paires de coordonnées désignant le début et la fin de chaque craque

Spécifications de sortie:

En sortie vous devrez fournir:

- un entier correspondant aux nombres de zones distinctes contenues dans le rectangle après les craques

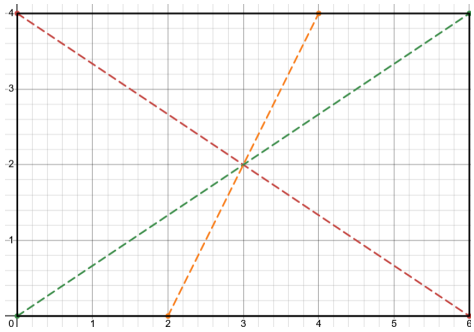
Exemple d'entrée:

```
((0,0),(6,4))
[((0,0),(6,4)),((2,0),(4,4)),((0,4),(6,0)),((0,3),(6,3))]

((0,0),(2,4))
[((0,4),(2,3)),((1,0),(2,3)),((0,4),(1,0)),((0,2),(1,4)),((0,3),
(2,0)),((0,1),(2,1))]

((0,0),(4,3))
[((4,1),(2,3)),((1,0),(2,3)),((0,3),(1,0)),((0,2),(4,1)),((0,3),
(2,0)),((0,1),(4,2)),((2,0),(4,2)),((1,0),(4,2))]
```

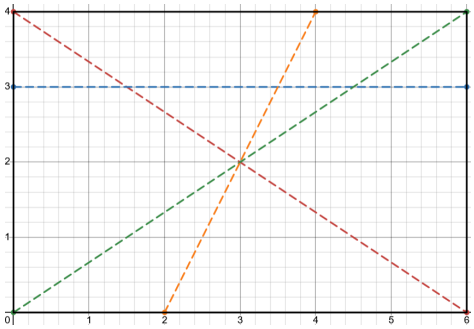
Exemple de sortie:

```
10
14
23
```

Explication de la première sortie:



En reprenant l'exemple commencé dans la description du problème, où on a 4 régions après les deux premières craques. La troisième craque (rouge) n'introduit qu'un autre croisement et ajoute donc deux régions.



La dernière craque (bleue) effectue trois croisements différents et ajoute 4 régions supplémentaires pour un total de 10.

EL4: Feu de forêt (40 points)

Pour mieux savoir comment arrêter des feux de forêts, le ministère de l'environnement vous confie la tâche de modéliser la propagation de feux de forêts afin de mieux prévoir comment arrêter un tel feu. Vous serez placés dans une grille carrée 20 x 20 représentant différentes sections d'une forêt où une case spécifique aura pris feu. Une case peut avoir plusieurs états:

- Vierge, cette case contient bien une portion de forêt mais cette portion se porte plutôt bien
- Allumée, cette case vient de se faire allumer par un feu adjacent
- Enflammée, cette case est maintenant assez enflammée pour allumer les cases vierges adjacentes
- Brûlée, tout est brûlé dans cette case, elle ne propage plus de feu

Vous devrez itérer sur la forêt après chaque heure. Après chaque heure, le feu se propage: les cases allumées deviennent enflammées, et les cases enflammées deviennent brûlées. Les règles de propagation entre les cases lorsqu'on change d'heure sont les suivantes:

- Si une case vierge est orthogonalement adjacente (c'est-à-dire partage un côté avec cette case) à exactement une case enflammée, elle devient allumée à la prochaine heure
- Si un case vierge ou allumée est orthogonalement adjacente à 2 ou plus cases enflammées, elle devient instantanément enflammée à la prochaine heure

Vous devrez déterminer quand il n'y aura plus aucun feu, tel que toutes les cases restantes soient soit vierges, soit brûlées. Une date et heure de début vous sera donnée et vous devrez donner l'heure de fin du feu dans le même format. Vous n'aurez jamais à changer de mois dans la date, seulement à changer l'heure et le jour selon votre résultat.

Spécifications d'entrée:

En entrée vous recevrez:

- ***feux:*** une liste de tuples contenant les paires de coordonnées (x,y) des feux initiaux (la case en bas à gauche étant (0,0) et la prochaine case à sa droite étant (1,0))
- ***debut:*** une string contenant la date et l'heure du début du feu

Spécifications de sortie:

En sortie vous devrez fournir:

- une string dans le même format que ***début*** correspondant à l'heure de fin du feu

Exemple d'entrée:

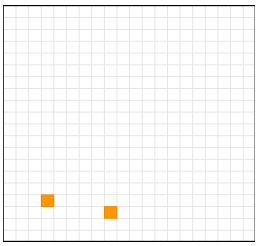
```
[(8,2),(3,3)] "20 janvier, 8:57"
[(0,19)] "5 mars, 11:11"
[(9,9)] "7 décembre, 17:00"
[(17,2),(8,11),(13,7)] "22 août, 9:23"
```

Exemple de sortie:

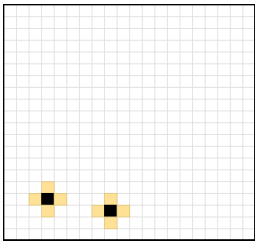
```
"22 janvier, 6:57"
"7 mars, 21:11"
"9 décembre, 0:00"
"23 août, 13:23"
```

Explication de la première sortie:

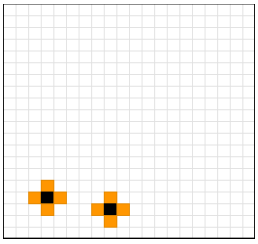
Montrons les premières itérations du feu de forêt. Après avoir placé nos deux feux initiaux, la forêt ressemble à ceci:



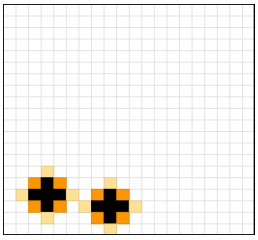
Nous utiliserons la couleur orange pour signifier une case enflammée, jaune pour une case allumée et noir pour une case brûlée. Ces deux feux font en sorte que chacune de leurs cases vierges adjacentes s'allumeront. Ces deux cases deviendront brûlées à la prochaine heure.



Après 1 heure. Il n'y a aucune case enflammée et il n'y aura donc pas de propagation d'une case à l'autre possible. Les cases allumées deviennent enflammées.



Après 2 heures. Les coins vierges du carré 3x3 centré sur un feu initial ont deux cases enflammées comme voisins, ce qui fait qu'ils deviendront enflammés à leur tour. Chaque case enflammée a un voisin vierge à l'opposé de la case brûlée qui n'a qu'un voisin enflammé. Ces cases deviennent allumées. Finalement, Les cases enflammées deviennent brûlées.



Après 3 heures. On répète le patron de propagation du feu jusqu'à la fin. Dans ce cas-ci, c'est jusqu'à ce que la forêt brûle en entier après 46 heures. 46 heures après le 20 janvier à 8h57min donne le 22 janvier à 6h57min.

Ragnarök (RK)

RK: La bataille des 6 armées (100 points)

Le Ragnarök approche et la guerre se dessine avec les affrontements à venir des 6 puissantes armées. D'un côté se trouve les armées de Odin, Thor et Freya contre les armées de Loki, Hel et Fenrir. Vous serez en charge de simuler les combats épiques ainsi que la grande bataille finale pour déterminer l'issue du Ragnarök. Vous aurez à simuler tour par tour les conflits entre deux armées opposées pour déterminer le vainqueur et ses unités restantes pour le grand combat final. Ainsi, vous simulerez les combats initiaux et toutes les unités ayant survécu à ces combats s'affronteront dans la grande bataille de la fin de l'humanité. Toutes les unités avec leur statistiques vous sont directement fournies dans le template et les armées sont déjà instanciées pour vous! Il vous reste donc à mettre en place les règles d'un combat et simuler les combats incluant la bataille finale.

Les règles d'un combat:

- L'armée qui commence le combat selon les affrontements donnés est **l'armée en premier** dans le tuple des combats. Exemple de combats: [("Odin", "Fenrir"), ("Thor", "Hel"), ("Loki", "Freya")] Ce qui veut dire que dans le combat de l'armée d'Odin contre celle de Fenrir, c'est l'armée d'Odin qui attaque en premier.
- Lors de l'affrontement final, l'armée qui attaque en premier est celle avec le **moins de vie restante** (en additionnant la vie de toutes les unités).
- Quand une armée attaque, ce sont **toutes les unités avec le score de priorité le plus élevé** qui sont encore en vie qui attaquent l'une après l'autre.
- Chaque unité d'une armée attaquent **une après l'autre**. La cible de l'attaque est l'unité avec le **score de priorité le plus élevé**. S'il y a plusieurs unités en vie avec le même score de priorité, celle avec le **plus de vie** est attaquée.
- Pour compter le dommage d'une unité, on prend le **attack_value - defense_value** et le dommage résultant est retiré de la vie de la cible.
- On échange les tours d'attaque jusqu'à ce qu'une armée n'ait plus d'unités en vie.
- Toutes les unités ayant survécu sont prises avec leur vie restante pour l'affrontement final.
- L'affrontement final contient d'un côté ce qu'il reste des armées de **Odin, Thor et Freya** contre les unités restantes des armées de **Loki, Hel et Fenrir**.

Spécifications d'entrée:

En entrée vous recevrez:

- ***combats:*** Un *array* de *tuples* des affrontements initiaux avec comme première armée celle qui attaque en premier.
- ***armies***: Un *dictionnaire* contenant des *listes* de toutes les unités de chacune des armées

Spécifications de sortie:

En sortie vous devrez fournir:

- un array de strings du nom des dieux ayant survécu à la bataille finale

Exemple d'entrée:

```
[("Odin", "Fenrir"), ("Thor", "Hel"), ("Loki", "Freya")]
```

Exemple de sortie:

```
Vous devez trouver par vous même mais ça serait du style
["Odin", "Freya"]
```

Explication de la première sortie:

Il n'y a pas d'explication supplémentaire, si vous avez la moindre question allez voir CRC.Frank