TP C#1 : Input/Output

Consignes de rendu

A la fin de ce TP, votre rendu devra respecter l'architecture suivante :

```
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- TP1/
        |-- TP1.sln
        |-- Exercise1/
            |-- Everything except bin/ and obj/
        |-- Maze
            |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

Dans ce TP, nous allons étudier l'interaction entre notre programme et des fichiers stockés sur votre disque.

Nous utiliserons le namespace System.IO pour pouvoir manipuler les fichiers en C. I/O est le diminutif de Input/Output et se traduit en Français par Entrée/Sortie. Il réfère à toute communication entre un programme et le monde extérieur. D'où le nom System.IO.

Ce namespace contient un certain nombre de classes qui nous donnent accès à de nombreuses fonctionnalités que nous verrons dans la partie cours.

Le but de ce TP va être dans un premier temps de découvrir certaines des fonctions les plus importantes de System.IO. Dans un second temps, de les utiliser dans des cas d'application concrètes.

2 Outils et rappels

2.1 Rappels

Pour commencer en douceur, nous vous proposons de voir ou revoir l'utilisation de deux outils que vous allez utiliser tout au long de votre scolarité à EPITA, i3 et git. Ces outils sont par ailleurs utilisés lors des examens machines, leur maitrise est dont fortemment recommandée.

2.1.1 i3, Le « TILING WINDOW MANAGER »

Voici quelques combinaisons de touches par défaut, notez que la touche **mod** est celle que vous avez configurée à votre première connexion :

Combinaisson	Action
mod+Return	ouvre un terminal
mod+d	ouvre d-menu
mod+[numero]	se déplacer au workspace [numero]
mod+Shift+[numero]	déplace la fenêtre active au workspace [numero]
mod+f	passe en mode plein écran
mod+Shift+q	kill l'application active
mod+flêches	navigue entre les différentes fenêtres du workspace
mod+Shift+flêches	déplace la fênetre active sur le workspace
mod+Shift+e	quitter i3 et donc se déconnecter

Pour aller plus loin

Globalement la configuration de i3 est très bien documentée et claire. Elle est donc accessible aux utilisateurs souhaitant bidouiller leur configuration facilement. N'hésitez pas à faire des recherches et toucher à votre "i3/config".

2.1.2 GIT

Git est un logiciel faisant partie de la famille des systèmes de contrôle de version. Il en existe plusieurs, mais il est aujourd'hui l'un des plus connus, et surtout celui utilisé au sein d'EPITA. L'usage de **git** que vous aurez au cours du semestre est relativement simple par rapport aux fonctionnalités qu'il possède. Nous vous encourageons sa bonne maitrise pour vous permettre de mieux gérer votre travail, notament être capable de revenir en arrière ou collaborer avec efficacité sur vos projets.

La command **git clone** va récupérer votre *repository git* c'est-à-dire votre répertoire de travail géré par **Git**, vous pourrez donc travailler à l'intérieur ! Le lien du **git** vous sera donné pour chaque TP, ce n'est pas celui là !

```
git clone git@acdc.epita.fr:/chemin/vers/dépôt
```

La commande **git add** ajoute des fichiers à votre **git**. Par exemple, la commande suivante ajoutera un fichier nommé **temp.txt**.

```
git add temp.txt
```

La commande **git commit** permet de valider les modifications apportées au **repository git**. Notez que **commit** ne signifie pas envoyer les informations au serveur.

```
git commit -m "Description du commit"
```

La commande **git status** affiche la liste des fichiers modifiés ainsi que les fichiers qui doivent encore être ajoutés ou validés.

```
git status
```

Git push est une autre commandes **GIT** de base. Un simple **push** envoie les modifications locales apportées grace à vos **commits**, au serveur.

```
git push origin master
```

Vous avez ajouté un fichier bin ou obj sur le git ? Enlevez le grace à **git rm** avant que vos assistants ne le remarque. Cela ne va pas supprimer le fichier de votre ordinateur.

```
git rm file.txt
```

Pour aller plus loin

Git est très puissant quand il s'agit de travailler à plusieurs ! Nous vous conseillons fortement de regarder l'utilité et le fonctionement des branches, cela vous sera très utile pour vos projets.

3 Cours System.IO

Dans cette partie cours, les notions nécessaires pour compléter le TP vont vous être expliquées. Vous devrez aussi vous rendre sur MSDN pour étudier le fonctionnement des différentes fonctions de **System.IO**. Tout les champs qui ont **Exception** dans leur nom peuvent être ignorés, ils ne vous intéressent pas pour le moment.

3.1 File

Il y a deux parties importantes dans un fichier. Des informations sur lui-même et son contenu. La classe **FileInfo** permet d'obtenir et de modifier des informations sur un fichier comme :

— Sa date de création
— Sa longueur
— Son dossier parent
— Son chemin absolu
— ...

La classe **File** permet quant à elle de manipuler le contenu du fichier. Il y a de nombreuses fonctions qui permettent de lire, écrire et ajouter dans un fichier. Allez lire la MSDN pour obtenir plus d'informations.

Voici une liste non-exhaustive des fonctions qui pourraient vous être utile :

— **ReadAllText** lis la totalité du contenu du fichier et le retourne sous forme de chaîne de caractères
— **ReadAllLines** lis la totalité du contenu du fichier ligne par ligne et le retourne sous forme de liste de chaîne de caractères. Chaque élément de la liste représente une ligne.
— **WriteAllText** efface la totalité du contenu du fichier et écrit le texte passé en paramètre à la place.
— **AppendAllText** ajoute à la fin du fichier le texte passé en paramètre.
— **Delete** supprime le fichier.

3.2 Directory

De la même manière qu'un fichier, un dossier (directory en Anglais) possède une partie information. La classe **DirectoryInfo** permet d'obtenir et modifier à peu prêt les mêmes informations qu'un fichier mais pour un dossier.

La classe **Directory** permet de manipuler les dossiers. Elle sert surtout à obtenir des informations sur son contenu (les fichiers et dossiers qu'il contient). Elle permet aussi de supprimer, créer et déplacer des dossiers avec les fonctions respectivement CreateDirectory, Delete et Move. La fonction Move est aussi capable de déplacer des fichiers.

3.3 Path

Les fichiers et les dossiers sont stockés sous forme d'arbre. Un dossier peut avoir zéro ou plusieurs fils qui sont des fichiers et des dossiers également. Ainsi, on appelle fils d'un dossier, tous les fichiers et dossiers contenus dans ce dossier. On appelle parent d'un fichier, le dossier dans lequel est contenu le fichier. Il y a un dossier spécial, la racine. Tous les dossiers et fichiers

du disque sont des descendants de ce dossier.

Par convention, on ajoute un '/' à la fin du nom des dossiers pour les différencier des fichiers. Les chemins (ou path en Anglais) sont des chaînes de caractères qui permettent de se déplacer dans cette arborescence de fichiers et dossiers.

— / désigne la racine.
— ./ désigne le dossier actuel.
— ../ désigne le dossier parent.
— nom désigne le <nom> d'un fichier ou dossier.
— ../test désigne, dans le dossier parent, le fichier ou dossier test.

Il existe deux types de chemins : relatif et absolu.

Un chemin relatif est défini par rapport à l'endroit où on se trouve actuellement dans l'arborescence. Par exemple, si on exécute du code dans un projet C, le dossier courant est là où se trouve l'exécutable (soit dans <PojectPath>/bin/Debug/, soit dans <PojectPath>/bin/Release/). Les fichiers de code se trouvent donc avec le chemin : ../../*.cs (*.cs désigne les fichiers de code du projet).

Un chemin absolu est un chemin préfixé de / et qui part donc de la racine. Son avantage est qu'il ne dépend pas de l'endroit où l'on se trouve actuellement.

Voici quelques exemples de correspondances entre les chemins absolus et relatifs :

```
Exemple: on est dans /tmp/tests/
../ => /tmp/ # Le parent de tests/ est tmp/
./../../ => / # Le parent du parent de tests/ est la racine /
../../../ => / # Le parent de la racine est la racine
```

3.4 Les projets en C#

3.4.1 Solution

Une solution est une structure qui permet d'organiser les projets. Elle peut donc contenir plusieurs projets. Si vous avez attentivement regardé l'architecture de rendu, vous avez remarqué que vous devez remettre une solution avec deux projets. La solution s'appelle TP6, et possède deux projets : Exercise1 et Maze.

3.4.2 Projets

Un projet est un dossier contenant

— Deux dossiers bin/ et obj/ qui sont recréés à chaque exécution du projet. Le code compilé est mis dans ces deux dossiers. Ils peuvent donc être supprimés sans poser de problème.
— Un fichier .csproj qui définit le projet.
— Un nombre indéfini de sous-dossiers et de fichiers de code. Les .cs par exemple sont les fichiers qui contiennent le code en C.

Chaque projet peut se compiler indépendamment. Pour lancer facilement un projet : le sélectionner dans l'explorateur de solution, clic droit, run <project>.

Voici l'architecture du projet Exercise1 :

```
Exercise1 /
| - - bin /
| - - obj /
| - - Exercise1 . csproj
| - - Architecture . cs
| - - CopyFile . cs
| - - PrintFile . cs
| - - Test . cs
```

4 Exercices

C'est maintenant à vous de jouer avec les fichiers.

4.1 Exercise1 : Bases

Cet exercice va vous faire utiliser les différentes fonctions des classes :

— **File**
— **Directory**
— **Path**

Vous devrez modifier les fonctions dans le projet nommé **Exercise1**. Pour tester une fonction, il faut l'appeler dans la fonction **Main** qui est dans le fichier **Test.cs**. Aussi, des fichiers pour tester votre code sont fournis dans l'archive, dans le dossier **tests**.

Le fichier **Test.cs** sera supprimé lors de la correction, donc vous ne devez rien coder d'important dans ce fichier. Le dossier **tests** sera, de la même manière, supprimé.

N'oubliez pas d'ajouter au début de chaque fichier :

```
1 using System.IO
```

Ainsi vous n'aurez pas à taper **System.IO.<class>.<function>()** à chaque utilisation d'une fonctionnalité du **namespace**. Vous pourrez ainsi directement écrire **<class>.<function>()**.

4.1.1 Afficher un fichier

Vous devez trouver la manière la plus simple d'afficher la totalité du contenu d'un fichier dans la console.

Il faut compléter la fonction suivante dans le fichier **PrintFile.cs :**

```
1 public static void PrintAllFile(string path)
2 {
3     //TODO
4 }
```

Attention ! Il faut gérer le cas où le fichier passé en paramètre n'existe pas. Il faut alors afficher un message d'erreur personnalisé. Vous pouvez par exemple afficher :

```
could not open file: <file given as parameter>
```

4.1.2 Je garde la moitié

Cette fois il faut afficher uniquement une ligne sur deux du fichier passé en argument. Il faut donc que la première ligne du fichier soit affichée, mais pas la deuxième.
Les lignes numéro : 0, 2, 4, 6, 8, 10, . . . doivent s'afficher.
Les lignes numéro : 1, 3, 5, 7, 9, 11, . . . ne doivent PAS s'afficher.

Il faut compléter la fonction suivante dans le fichier **PrintFile** :

```
1  public static void PrintHalfFile(string path)
2  {
3      //TODO
4  }
```

Comme la question précédente, n'oubliez pas de gérer le cas où le fichier passé en paramètre n'existe pas.

4.1.3 Copy-Paste

De la même manière que la partie 1, il faut maintenant récupérer la totalité du contenu du fichier source, puis le copier dans le fichier destination.
Si le fichier destination n'existe pas, il faut le créer.
Si le fichier destination existe déjà, il faut écrire par-dessus.

Vous devez compléter la fonction suivante dans le fichier **CopyFile.cs** :

```
1  public static void CopyAllFile(string source, string destination)
2  {
3      //TODO
4  }
```

Comme la question précédente, n'oubliez pas de gérer le cas où le fichier passé en paramètre n'existe pas.

4.1.4 Copy-Flemme

Pour cette fonction, il faut copier la moitié du fichier source dans le fichier destination. Si le fichier destination n'existe pas, il faut le créer. Si le fichier destination existe déjà, il faut écrire par-dessus.

Si le fichier fait 11 lignes par exemple : Les lignes de 0 à 4 seront copiées dans le fichier destination (les 5 premières lignes). Les lignes de 5 à 10 seront ignorées (les 6 dernières lignes).

Si le fichier fait 10 lignes par exemple : Les lignes de 0 à 4 seront copiées dans le fichier destination (les 5 premières lignes). Les lignes de 5 à 9 seront ignorées (les 5 dernières lignes).

Il faut compléter la fonction suivante dans le fichier **CopyFile.cs**.

```
1  public static void CopyHalfFile(string source, string destination)
2  {
3      //TODO
4  }
```

Comme la question précédente, n'oubliez pas de gérer le cas où le fichier passé en paramètre n'existe pas.

4.1.5 Architecte de talent

Cette fonction devra créer une architecture de fichiers comme expliqué dans les étapes suivantes :

```
1. Créer un dossier avec le chemin (path) donné en paramètre.
2. Créer un fichier "AUTHORS" à l'intérieur du dossier qui vient d'être créé.
3. Remplir le fichier avec le texte "* prenom.nom\n"
('\n' est le caractère de saut de ligne)
4. Créer un fichier "README" à l'intérieur du dossier qui vient d'être créé.
5. Remplir le fichier avec le texte
"Everything in programming is magic... except for the programmer\n"
6. Créer un dossier "TP1" à l'intérieur du dossier qui vient d'être créé.
7. Créer un fichier "useless.txt" vide à l'intérieur du dossier "TP1".
```

Si on donne path égal à **"/tmp/archi"**, l'architecture obtenue devra donc être :

```
/tmp/archi/
| -- AUTHORS
| -- README
| -- TP1/
     | -- useless.txt
```

Il faut compléter la fonction suivante dans le fichier **Architecture.cs**.

```
1  public static void Architect(string path)
2  {
3      //TODO
4  }
```

Attention ! Il faut gérer ces deux cas :
— Si "path" passé en paramètre est un fichier déjà existant, le supprimer.
— Si "path" passé en paramètre est un dossier déjà existant, le supprimer ainsi que tout son contenu.

4.2 Exercise2 : Labyrinthe

Maintenant que vous avez vu les bases de l'utilisation de **System.IO**, passons à quelque chose d'un peu plus sérieux.

Dans cet exercice, vous devrez créer des fonctions dans le projet nommé **Maze**. Vous êtes libre de créer autant de fonctions que vous voulez. Vous pouvez même ajouter des fichiers .cs dans le projet. Il faut cependant que le fichier **Maze.cs** contienne la fonction **Main**.

4.2.1 Présentation

Le programme que vous allez créer se découpe en trois étapes :
Il faut d'abord charger un labyrinthe dans son programme. Il faut donc premièrement demander le nom du fichier à charger à l'utilisateur, puis le charger et mettre sa donnée dans une grille.

Une fois le labyrinthe chargé dans la grille, il faut le résoudre. Pour ce faire, il faut trouver un chemin entre le départ et la fin du labyrinthe. Vous devrez modifier la grille pour y insérer le chemin que vous avez trouvé.

Maintenant que le labyrinthe est résolu, il suffit de récupérer la grille et l'enregistrer dans un nouveau fichier.

4.2.2 Consignes

Il est temps de vous expliquer exactement ce que vous devez faire. Votre programme devra respecter strictement les consignes.

Demander un fichier

Au début de votre programme, vous devez demander à l'utilisateur quel labyrinthe il souhaite charger. Vous devez ensuite récupérer sa réponse. Cependant, puisque vous ne lui faites évidemment pas confiance, il faut vérifier qu'il ne vous ai pas donné n'importe quoi. Il ne faut jamais faire confiance à l'utilisateur, il peut toujours faire des bêtises (surtout vos ACDC qui testent votre code dans ses derniers retranchements).

Vous avez donc deux tests à faire, vérifier que le fichier existe et que celui-ci a bien **.maze** en extension. Pour réaliser ces deux vérifications allez chercher dans les classes **File** et **Path** de **System.IO**. Il y a des fonctions qui vous permettent de tout tester très facilement.

Exemples :

```
> Which file should be loaded ?
/tmp/tests/map1.txt # ce n'est pas un .maze donc on recommence
> Which file should be loaded ?
/tmp/tests/map1.maz # ce n'est pas un .maze donc on recommence
> Which file should be loaded ?
/tmp/tests/map1.maze # c'est un .maze et ce fichier existe bien
> Thank you, bye
```

Le fichier de sortie

Vous avez un fichier de labyrinthe valide, il faut, à partir de celui-ci, trouver le nom du fichier de sortie (où vous enregistrerez la solution).

Pour l'obtenir, vous devrez changer l'extension du fichier d'entrée **.maze** en une extension **.solved**.

Le format du .maze

Les fichiers **.maze** sont enregistrés sous la forme d'un tableau de caractères.

Voici un exemple de **.maze**

```
> cat -e tests/map1.maze
SBOOOBOOOBOOO$
OBOBOBOBOBOBO$
OBOBOBOBOBOBO$
OBOBOBOBOBOBO$
OBOBOBOBOBOBO$
OBOBOBOBOBOBO$
OOOBOOOBOOOBF$
```

Attention

Les **$** signifient *fin de ligne*, ils sont ajoutés par l'option **e** de cat. Vous ne devez pas les ajouter à votre **.maze** !

Un .maze est un donc un rectangle avec chaque caractère qui représente un type de case. Voici la signification de chaque caractère :

— S est la case START, le début du labyrinthe.
— F est la case FINISH, la fin du labyrinthe.
— B est une case BLOCK, un mur qu'on ne peut pas traverser.
— O est une case EMPTY, un chemin (on peut passer).
Le but est donc de parcourir les cases O, pour aller du départ S à l'arrivée F.

Le format du .solved

Dans le labyrinthe, il est possible de se déplacer uniquement en vertical et horizontal. Les mouvements diagonaux sont interdits. Si un chemin à été trouvé, toutes les cases du chemin doivent désormais être des P pour PATH.

```
> cat -e tests/map3.maze
SOBOBOBOBOBOBOBOBF$
OOOOOOOOOOOOOOOOOO$
BOBOBOBOBOBOBOBOBB$
BBBBBBBBBBBBBBBBBB$
> cat -e tests/map3.solved
PPBOBOBOBOBOBOBOBF$
OPPPPPPPPPPPPPPPPP$
BOBOBOBOBOBOBOBOBB$
BBBBBBBBBBBBBBBBBB$
```

La classe Point

Dans le fichier **Maze.cs**, en plus de la classe **Maze**, où vous allez coder l'exercice, il y a une seconde classe nommée Point.

Cette classe permet de stocker les coordonnées d'une case dans la grille du labyrinthe. Plutôt que de balader deux variables dans chaque fonction (une pour les abscisses, une pour les ordonnées), le point permet de n'avoir plus qu'une seule variable.

Voici des exemples d'utilisation :

```
public static void Main(string[] args)
{
    // Créer un nouveau point avec X = 0 et Y = 1.
    Point p = new Point(0, 1);
    Console.Writeline(p.X); // affiche 0
    Console.Writeline(p.Y); // affiche 1
    p.X = 2;
    p.Y = 10;

    Console.Writeline(p.X); // affiche 2
    Console.Writeline(p.Y); // affiche 10
}
```

4.2.3 Suivez le guide (ou pas)

Tant que vous respectez les consignes, vous n'êtes pas obligé de suivre le guide. **Vous pouvez implémenter cet exercice de la manière que vous voulez.**

Si vous utilisez les fonctions décrites ci-dessous, elles devront être ajoutées dans le fichier **Maze.cs** dans la classe **Maze**.

Demander le .maze

Demander le fichier **.maze** à l'utilisateur et vérifier sa validité. Pour cela créer une fonction qui renvoie le chemin vers un fichier valide :

```
private static string AskMazeFile()
{
    // TODO
}
```

— Demander un fichier à l'utilisateur.
— Attendre une entrée de sa part.
— Vérifier si le fichier existe et est un **.maze**.
— Si ce n'est pas le cas recommencer depuis le début.

Le nom du .solved

Trouver le nom du fichier **.solved** à partir du nom du fichier **.maze**. Pour cela, faire une fonction qui prend en paramètre le nom du fichier **.maze** et retourne le nom du fichier **.solved** :

```
private static string GetOutputFile(string fileIn)
{
    // TODO
}
```

Il faut trouver la fonction dans **System.IO** qui permet de faire ça en une ligne.

Récupérer le labyrinth du .maze

Lire le **.maze** passé en paramètre. Créer un tableau à deux dimension à partir de ça puis le retourner.

```
1  private static char[][] ParseFile(string file)
2  {
3      // TODO
4  }
```

Trouver le départ

Pour pouvoir résoudre le labyrinthe, il faut peut-être commencer au début. On a donc besoin de trouver les coordonnées de la case START qui est représentée par un caractère S. Pour cela, on prend en paramètre la grille du labyrinthe et on cherche la caractère S. Lorsqu'on l'a trouvé, on créé un objet Point avec les coordonnées de la case que l'on vient de trouver. Enfin, on retourne cet objet Point. Le cas d'un fichier sans case START ne sera pas testé.

```
1  private static Point FindStart(char[][] grid)
2  {
3      // TODO
4  }
```

Résoudre le labyrinthe

La méthode de résolution proposée ici est le **backtracking**. C'est une méthode simple qui fonctionne. Il y a cependant d'autres méthodes qui peuvent donner de meilleurs résultats.

La fonction est une fonction récursive. Elle prend en paramètre :
— La grille du labyrinthe qui s'appelle **grid**.
— Une grille de la même taille que grid (processed) , mais remplie de zéros. Elle est à créer avant d'appeler la fonction pour la première fois. Elle permet de savoir sur quelles cases on est déjà passé. Si une case est à 0, on est jamais passé dessus, si elle est différente de 0, on est déjà passé dessus.
— Un point. On lance l'algorithme avec le point de départ (START).

```
1  private static bool SolveMazeBackTracking(char[][] grid, int[][] processed,
2                                            Point p)
3  {
4      // TODO
5  }
```

L'algorithme de **backtracking** consiste à suivre un chemin tant qu'on ne sait pas s'il est vrai ou faux. S'il est faux, on arrête de le suivre et on retourne à l'étape précédente en signalant que c'est un mauvais chemin. S'il est vrai, on arrête, on à fini et on retourne sur toutes les étapes précédentes en signalant qu'on est sur le bon chemin.

```
1. Si le point est en dehors de la grille, retourner FAUX.
2. Si on est déjà passé sur ce point, retourner FAUX.
3. On indique que maintenant on est passé sur ce point.
4. Si ce point est la case FINISH, retourner VRAI.
5. Si ce point est une case BLOCK, retourner FAUX.
6. Appeler récursivement l'algorithme sur la case au-dessus.
7. S'il a renvoyé vrai, mettre la case actuelle en PATH, retourner VRAI.
8. Appeler récursivement l'algorithme sur la case à gauche.
9. S'il a renvoyé vrai, mettre la case actuelle en PATH, retourner VRAI.
10. Appeler récursivement l'algorithme sur la case à droite.
11. S'il a renvoyé vrai, mettre la case actuelle en PATH, retourner VRAI.
12. Appeler récursivement l'algorithme sur la case en-dessous.
13. S'il a renvoyé vrai, mettre la case actuelle en PATH, retourner VRAI.
14. Retourner FAUX.
```

Sauvegarder le résultat

Il vous faut maintenant enregistrer cette grille résolu dans un nouveau fichier !

```
1  private static void SaveSolution(char[][] grid, string fileOut)
2  {
3      // TODO
4  }
```

4.2.4 Bonus

Bonus 1 : Améliorations
Améliorer la recherche du chemin. Le **backtracking** est lent et ne donne pas le meilleur chemin. Implémentez un algorithme capable de trouver le meilleur chemin possible rapidement. Vous pouvez vous aider de Wikipédia (recherchez Path Finding) ou d'autres sources si vous le souhaitez.

Bonus 2 : Affichages
Vous pouvez afficher le labyrinthe sous une forme plus élégante que des lettres. Les ordinateurs du PIE sont configurés avec un terminal (URxvt) capable d'afficher tous les caractères unicodes. Autant vous dire qu'il y a de quoi être très créatif. N'hésitez pas à fouiller la classe System.Console pour voir aussi comment modifier les couleurs.

Bonus n : Autres
Comme à chaque fois, plus vous faites de bonus, mieux c'est ! N'oubliez juste pas de les expliquer dans votre README.

If you knew time as well as I do, You wouldn't talk about wasting it