Ciências ULisboa | Informática

Introdução à Programação

Licenciatura em Engenharia Informática

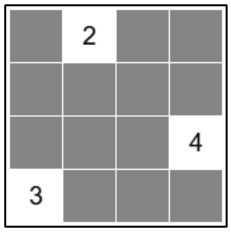
Trabalho: Iª parte

2025/2026

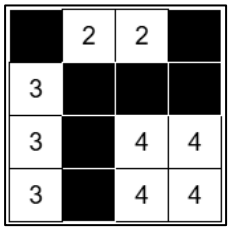
Logic IPuzzle

O trabalho de programação que vos é proposto em IP é sobre puzzles lógicos com números, que para efeitos do trabalho vamos chamar *Logic IPuzzle*.

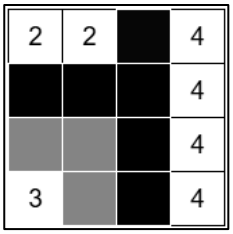
Um Logic IPuzzle consiste numa grelha retangular (n x m) onde algumas posições contêm números e algumas estão pintadas de preto — as pistas do puzzle. Uma pista que consista em ter uma posição com um número **k** indica que essa posição é branca e faz parte de uma forma retangular com exatamente **k** posições brancas. Resolver o puzzle consiste em deixar pintadas de preto todas as posições pretas dadas nas pistas e pintar todas as restantes posições da grelha de preto ou branco, sendo que as posições pintadas de branco devem formar formas retangulares que estejam de acordo com pistas fornecidas. Por exemplo, se uma posição do puzzle contiver o número 2 significa que essa posição é da cor branca, e que faz parte de uma forma retangular composta por exatamente 2 posições brancas. Neste caso há apenas duas possibilidades para essa forma retangular: 2 posições contiguas na mesma linha ou na mesma coluna. No entanto, por exemplo, se a pista numa posição do puzzle for o número 6 já há mais possibilidades para a forma retangular a que essa posição pertence: pode ser ocupando 1 linha e 6 colunas (1 x 6), ou 2 linhas e 3 colunas (2 x 3), ou 3 linhas e 2 colunas (3 x 2), ou 6 linhas e 1 coluna (6 x 1).

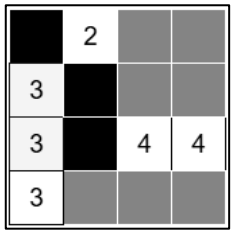


Num *Logic IPuzzle*, as pistas numéricas usam apenas os números de 1 a 9 e cada número é usado no máximo uma vez. Além disso duas formas retangulares brancas diferentes não se podem tocar horizontal ou verticalmente. Também não pode haver formas retangulares brancas sem pista associada. As formas das posições brancas têm sempre de ser retangulares (e notem que as formas quadradas também são retangulares). Não existem restrições para as posições pintadas a preto.



Ao lado, em cima, apresenta-se um exemplo de um puzzle, de tamanho 4 x 4 (4 linhas e 4 colunas), o qual tem três pistas (todas numéricas). No meio, apresenta-se a respetiva solução, que é única. Em baixo apresentam-se dois exemplos em que a solução do puzzle ainda está em construção. Num dos exemplos (o da direita) todas as posições já preenchidas estão corretas enquanto no outro exemplo isso não acontece. No que se segue, usa-se o termo *solução-puzzle-em-construção* para designar qualquer um destes casos.





Em que consiste o trabalho?

A vossa tarefa é desenvolver código Java que permita ler e validar puzzles do jogo *Logic IPuzzle*, bem como jogar o jogo, com uma interface textual básica. A vossa implementação está sujeita a um conjunto de restrições descritas de seguida, que tem como objetivo vos guiar no desenvolvimento da solução e ao mesmo tempo garantir que exercitam os conceitos já lecionados.

A implementação pretendida:

- Usa **`String`** na representação dos puzzles, soluções e, mais genericamente, das solução-puzzle-em-construção. Mais precisamente, uma solução-puzzle-em-construção é representada por uma **`String`** em que cada carácter corresponde a cada uma das posições da grelha, percorrida da esquerda para a direita e de cima para baixo. Uma **`String`** que representa uma solução-puzzle-em-construção pode apenas conter:
    - Dígitos (`'1'`–`'9'`), indicando que a respetiva posição é branca e faz parte de uma forma retangular cujo tamanho (número de posições) é dado pelo dígito;
    - Pontos (`'.'`), indicando uma posição ainda não preenchida;
    - A letra `'P'` maiúscula indicando que a posição é preta.

  Por exemplo, o puzzle mostrado anteriormente é representado por **"**`.2.........43...`**"**, a respetiva solução é representada por **"**`P22P3PPP3P443P44`**"**, a primeira solução-puzzle-em-construção é representada por **"**`22P4PPP4..P43.P4`**"** e a segunda solução-puzzle-em-construção é representada por **"**`P2..3P..3P443...`**"**.

- Usa os números inteiros entre `0` e (**`n`** x **`m`**) – `1` para representar as posições das solução-puzzle-em-construção sobre uma grelha de dimensão **`n`** x **`m`** (onde **`n`** representa o número de linhas e **`m`** o número de colunas). A numeração começa no canto superior esquerdo e é feita da esquerda para a direita, e depois de cima para baixo, terminando no canto inferior direito.

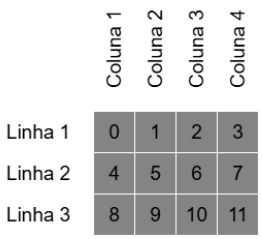
- Durante a resolução de um puzzle, enquanto este não estiver resolvido, permite ao utilizador selecionar uma posição da grelha a preencher (que não pode ser nenhuma para as quais foram dadas pistas) e indicar como quer preencher essa posição. É usado um número inteiro para a posição, enquanto para o preenchimento é usado:
    - Um dígito **d** para indicar que a posição é branca e pertence a uma forma retangular de tamanho **d**, sendo que **d** é uma pista do puzzle;
    - Um ponto para limpar uma posição;
    - Uma letra maiúscula P para indicar que a posição é preta.

- Só usa o que está disponível na versão 8 do Java e que tenha sido lecionado até à aula teórica 12 (consultar os sumários e os docentes em caso de dúvida).

Mais concretamente, devem implementar uma classe **`LogicIPuzzle`** que inclua os procedimentos listados abaixo. **Devem notar que a ideia será fazer estes procedimentos pela ordem listada**, pois em muitos casos a implementação ou especificação de um procedimento é suposto utilizar alguns dos procedimentos definidos anteriormente.

- Funções **`int rowOfPosition(int pos, int numColumns)`** e **`int columnOfPosition(int pos, int numColumns)`** que calculam, respetivamente, a linha e a coluna de uma posição de uma grelha, dado o número de colunas dessa grelha. A função deve assumir que **`pos`** não é negativo e **`numColumns`** é positivo. As linhas são numeradas de `1` a `numRows`, onde `numRows` é o número de linhas da grelha, e as colunas são numeradas de `1` a `numColumns`, onde `numColumns` é o número de colunas da grelha. As posições da grelha

são numeradas de `0` a `numRows x numColumns - 1`. Ao lado apresenta-se um exemplo de como as linhas, colunas e posições de uma grelha de uma solução-puzzle-em-construção, são numeradas.



- Função **`int positionNumber(int row, int column, int numColumns)`** que calcula a posição da grelha dada a linha `row` e a coluna `column`. A função deve assumir que **`row`**, **`column`** e **`numColumns`** são positivos. As posições, linhas e colunas da grelha são numeradas como indicado no ponto anterior.

- Uma função **`isIntFactor`** que, dados números inteiros **`number`** e **`factor`** que se assumem positivos, verifica se **`factor`** é um fator inteiro de **`number`**, isto é, se existe um número inteiro que multiplicado por **`factor`** resulte no valor de **`number`**. A função retorna **`true`** se o número existir e **`false`** caso contrário.
- Uma função **`getIntFactor`** que, dados números inteiros **`number`** e **`factor`** que se assumem positivos, retorna o número inteiro que multiplicado por **`factor`** resulte no número **`number`**. Esta função assume ainda que **`factor`** é um fator inteiro de **`number`**.

Uma função **`sumDigits`** que, dada uma **`String s`** que se assume que não é **`null`**, retorna a soma de todos os dígitos que **`s`** contém.

- Uma função **`countCharOccurrence`** que, dada uma **`String s`** que se assume que não é **`null`** e um carácter **`c`**, retorna quantas vezes **`c`** aparece em **`s`**.
- Uma função **`countDigitOccurrence`** que, dada uma **`String s`** que se assume que não é **`null`** e um número inteiro **`d`**, que se assume ser um dígito, retorna quantas vezes **`d`** aparece em **`s`**.
- Uma função **`hasDigit`** que, dada uma **`String s`** que se assume que não é **`null`** e dado um número inteiro **`d`**, que se assume ser um dígito, verifica se **`s`** contém **`d`**, retornando **`true`** se isso acontecer e **`false`** no caso contrário.

- Uma função **`isFillingChar`** que, dado um carácter verifica se é `'P'` ou um dos dígitos `'1'-'9'`.
- Uma função **`isValidPuzzleInConstruction`** que, dada uma **`String spc`**, verifica se **`spc`** representa uma solução-puzzle-em-construção, isto é, **`spc`** não é **`null`** e contém apenas os caracteres `'.'`, `'P'` e os dígitos `'1'-'9'`. A função retorna **`true`** se **`spc`** for válido e **`false`** caso contrário. Notem que esta função não valida se **`spc`** é uma solução de um puzzle, apenas que se trata de uma **`String`** que representa uma solução-puzzle-em-construção para algum puzzle.
- Uma função **`isFilled`** que, dada uma **`String spc`** que se assume que representa uma solução-puzzle-em-construção, verifica se todas as posições estão preenchidas, isto é, se **`spc`** não contém pontos (`'.'`) e retorna **`true`** se isso acontecer e **`false`** caso contrário.
- Uma função **`isPuzzle`** que dados dois números inteiros **`rows`** e **`cols`** que se assumem positivos e uma **`String spc`** que se assume que representa uma solução-puzzle-em-construção, verifica várias condições necessárias para **`spc`** ser um puzzle sobre uma grelha com **`rows`** linhas e **`cols`** colunas:
    - Se o tamanho de **`spc`** é **`rows`** `x` **`cols`**,

- Se não existem pistas com números repetidos,
- Se existem posições suficientes no puzzle, tendo em conta as pistas fornecidas.

A função retorna **`true`** se todas estas condições forem verdadeiras, e **`false`** no caso contrário.

- Uma função **`hasAdjacentPositionsWithDifferentDigits`** que, dados dois números inteiros **`rows`** e **`cols`** que se assumem positivos e uma **`String spc`** que se assume que tem tamanho **`rows`**x**`cols`** e representa uma solução-puzzle-em-construção, verifica que não existem duas posições adjacentes na grelha de **`spc`** com dois dígitos diferentes.
- Uma função **`isSolution`** que, dados dois números inteiros **`rows`** e **`cols`** que se assumem positivos, uma **`String puzzle`** que representa um puzzle sobre uma grelha de dimensão **`rows`** x **`cols`** e uma **`String spc`** que representa uma solução-puzzle-em-construção, verifica se **`spc`** é uma solução do **`puzzle`**, isto é:
    1. Se **`spc`** tem o mesmo tamanho de **`puzzle`** e tem todas as posições preenchidas;
    2. As pistas do **`puzzle`** (sejam posições com números ou pintadas de preto) também estão em **`spc`**, nas mesmas posições;
    3. Cada dígito **`d`** que é uma pista do **`puzzle`**, ocorre exatamente **`d`** vezes em **`spc`** e cada dígito **`d`** que não é uma pista do **`puzzle`**, não ocorre em **`spc`**.
    4. Para cada dígito **`d`** que é uma pista do **`puzzle`**, existe uma forma retangular em **`spc`**, composta por exatamente **`d`** posições marcadas com esse dígito, formado um retângulo válido dentro da grelha de tamanho **`rows`** x **`cols`**. Note que se esse retângulo existir, a primeira ocorrência de **`d`** em **`spc`** corresponde ao seu canto superior esquerdo. Note ainda que esta ocorrência de **`d`** em **`spc`** juntamente com todas ocorrências consecutivas de **`d`** à sua direita em **`s`** que estejam na mesma linha da grelha, formam um dos lados do retângulo, se este existir.
    5. Não existem retângulos em posições adjacentes, isto é, não existem duas posições adjacentes do puzzle com dois dígitos diferentes.

    A função retorna **`true`** se **`spc`** for uma solução do **`puzzle`**, e **`false`** caso contrário.

    Dica: É fortemente aconselhado definirem outras funções que vos ajudem a implementar esta função de forma que o seu corpo não ultrapasse 40 linhas de código.

- Um procedimento **`printPuzzleInConstruction`** que, dados dois números inteiros **`rows`** e **`cols`** que se assumem positivos e uma **`String spc`** que se assume que tem tamanho **`rows`** x **`cols`** e representa uma solução-puzzle-em-construção, imprime **`spc`** em forma de quadrícula (ou seja cada linha do puzzle aparece numa linha diferente, com as colunas alinhadas).
- Uma função **`isValidMove`** que dada uma **`String spc`** que se assume que representa uma solução-puzzle-em-construção, dada uma **`String puzzle`** que se assume que representa um puzzle (para alguma grelha) e tem o mesmo tamanho que **`spc`**, dado um inteiro **`pos`** que representa a posição de jogada, e dado um **`char c`** que representa a ação da jogada, valida se é uma jogada válida. Uma jogada é válida se:
    1. A posição de jogada corresponde a uma posição da grelha e para essa posição o **`puzzle`** não dá nenhuma pista.

2. A jogada a fazer **c** é um dos seguintes caracteres: `'.'`, `'P'` ou os dígitos `'1'–'9'`. Caso a jogada corresponda a um dígito, esse dígito tem de ser uma das pistas numéricas do **puzzle**.

A função retorna **true** se a jogada for válida e **false** caso contrário.

- Uma função **playMove** que, que dada uma **String spc** que se assume que representa uma solução-puzzle-em-construção, dada uma **String puzzle** que se assume que representa um puzzle (para alguma grelha), dado um inteiro **pos** e um carácter **c** que representa uma jogada e que se assume ser uma jogada válida, retorna uma **String** que representa a solução-puzzle-em-construção que resulta de efetuar a jogada em **spc**.
- Um procedimento **void play(Scanner sc, int rows, int cols, String puzzle)** que, dado um canal de leitura **sc** que se assume aberto para ler do *standart input*, dois números inteiros **rows** e **cols** que se assumem positivos, **puzzle** que se assume que representa um puzzle para uma grelha **rows** x **cols**, faz o seguinte:
    1. Declara uma variável **spc** do tipo **String** e inicializa-a com o **puzzle**;
    2. Imprime **puzzle**;
    3. Obtém do utilizador uma jogada – uma posição da grelha do puzzle e um carácter para colocar nessa posição – e caso a jogada seja válida, efetua a jogada, atualizando a solução-puzzle-em-construção **spc**. O jogador pode ainda indicar que quer terminar fornecendo o número -1 como posição de jogada. Caso a jogada lida não seja a de terminar e não seja válida, deve ser apresentada uma mensagem a indicar que a jogada é inválida;
    4. Caso a jogada lida não seja a de terminar, Imprime a solução-puzzle-em-construção **spc**;
    5. Repete os passos 3 e 4 até **spc** ser uma solução de **puzzle** ou o jogador indicar que quer terminar, imprimindo uma mensagem apropriada.
- O método **main**, caso sejam passados três argumentos na linha de comandos, deve inicializar uma variável com o número de linhas do puzzle com o primeiro argumento, uma variável com o número de colunas do puzzle com o segundo argumento, e uma variável do tipo **String** com o puzzle com o terceiro argumento.
  Caso não sejam fornecidos quaisquer argumentos, então o programa começa por ler do input primeiro o número de linhas do puzzle, depois o número de colunas, e por fim a **String** com o puzzle, e guarda a informação em variáveis de tipo apropriado. Em qualquer dos casos deve ser verificado que o número de linhas e colunas está entre 1 e 30 e os dados fornecidos definem efetivamente um puzzle. Neste caso, deve permitir ao utilizador jogar (**play**).

Podem, e devem, incluir na vossa classe outros procedimentos ou funções que considerem úteis. Não esquecer que tão importante como o programa funcionar é a legibilidade do código.

Recomenda-se que para a passagem de um dígito **d** enquanto número do tipo **int** para um carácter do tipo **char** usem **Character.forDigit(d,10)**. Inversamente, para obter o inteiro que representa um carácter **c** que é um dígito (ou seja, para o qual **Character.isDigit(c)** é verdadeiro), devem usar **Character.getNumericValue(c)**.

Exemplos ilustrativos de execuções do programa

Para facilitar a compreensão dos exemplos, apresenta-se o input a vermelho e omitiram-se algumas coisas (representado por (…)).

```
$ java LogicIPuzzle
Número de linhas (1..30): 4
Número de colunas (1..30): 4
Puzzle (string de tamanho 16 com '.', 'P', '1'..'9'): .2.........43...
Puzzle inicial:
.2..
....
...4
3...
Introduza uma jogada (ou -1 para sair).
Posição: 0
Preencher com: P
P2..
....
...4
3...
Introduza uma jogada (ou -1 para sair).
Posição: 3
Preencher com: P
P2.P
....
...4
3...
(…)
Introduza uma jogada (ou -1 para sair).
Posição: 1
Preencher com: .
Jogada inválida.
Introduza uma jogada (ou -1 para sair).
Posição: 4
Preencher com: 0
Jogada inválida.
Introduza uma jogada (ou -1 para sair).
Posição: 4
Preencher com: 2
P2.P
2...
..44
3P44
Introduza uma jogada (ou -1 para sair).
Posição: 4
Preencher com: 3
P2.P
3...
..44
3P44
(…)
Introduza uma jogada (ou -1 para sair).
Posição: 9
Preencher com: P
P22P
3PPP
3P44
3P44
Parabéns – o puzzle está resolvido!
```

```
$ java LogicIPuzzle
Número de linhas (1..30): 5
Número de colunas (1..30): 2
Puzzle (string de tamanho 10 com '.', 'P', '1'..'9'): ..2.....4
Os dados fornecidos não definem um puzzle válido. Verifique tamanho/pistas
repetidas/posições.
```

```
$ java LogicIPuzzle 5 2 ..2.....4
Os dados fornecidos não definem um puzzle válido. Verifique tamanho/pistas
repetidas/posições.
```

Para facilitar o teste do vosso programa podem colocar os dados de input num ficheiro e usar o redireccionamento do input como mostrado abaixo. É fornecido o ficheiro usado neste exemplo.

```
$ java LogicIPuzzle < LogicIPuzzle4x4.txt
Número de linhas (1..30):
Número de colunas (1..30):
Puzzle (string de tamanho16 com '.', 'P', '1'..'9'):
Puzzle inicial:
.2..
....
...4
3...
Introduza uma jogada (ou -1 para sair).
Posição:Preencher com:P2..
....
...4
3...
(…)
Introduza uma jogada (ou -1 para sair).
Posição:Preencher com:P22P
.PPP
3P44
3P44
Introduza uma jogada (ou -1 para sair).
Posição:Preencher com:P22P
3PPP
3P44
3P44
Parabéns – o puzzle está resolvido!
```

O que entrego? O ficheiro LogicIPuzzle.java com a solução, condignamente documentada. Devem incluir no início da classe um cabeçalho javadoc com @author (nome e número dos alunos que compõem o grupo). Para cada procedimento/função definidos há que preparar um cabeçalho incluindo a sua descrição, e, se for caso disso, @param, @requires, @ensures e @return. Apresentem código que siga as normas de codificação em Java, bem alinhado e com um número de colunas adequado.

Como entrego? Um dos dois alunos do grupo entrega o trabalho através da ligação que, para o efeito, existe na página da disciplina no moodle. O prazo de entrega é dia 9 de Novembro às 23h.

Quanto vale o trabalho? Esta primeira parte do trabalho é cotada para 10 valores e irá somar à nota da segunda parte.