TÉCNICO LISBOA

Fundamentos da Programação
Ano lectivo 2019-20
Primeiro Projecto
17 de Outubro de 2019

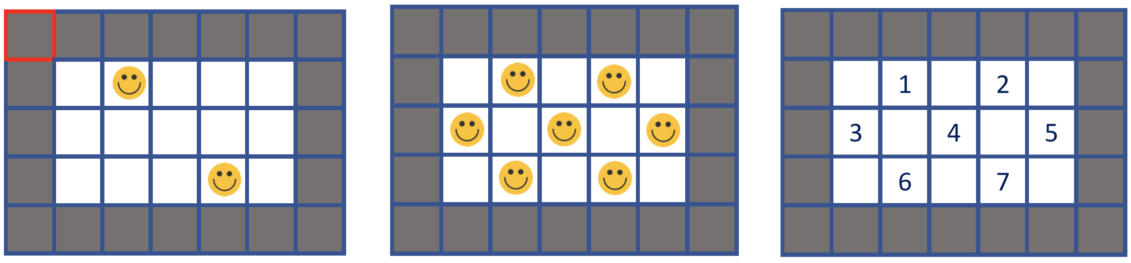
The Maze

Neste primeiro projecto de Fundamentos da Programação os alunos irão desenvolver as funções que permitam implementar um programa para simular o movimento de uma unidade através de um labirinto 2D contendo obstáculos e outras unidades. Esta unidade movimenta-se sempre para um espaço adjacente de forma a aproximar-se de uma das unidades alcançáveis mais próximas restantes no labirinto.

1 Movimento num labirinto

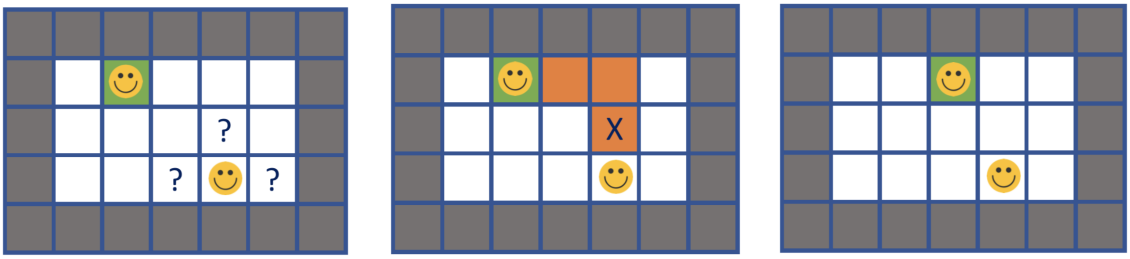
1.1 O labirinto e as unidades

O **labirinto** é uma estrutura rectangular de tamanho $N_x \times N_y$, sendo N_x o tamanho máximo do eixo de abcissas e N_y o tamanho máximo do eixo de ordenadas. Cada posição (x, y) do labirinto é indexada a partir da posição de origem $(0, 0)$ que corresponde ao canto superior esquerdo do labirinto. Num labirinto, todas as posições do limite exterior são *paredes*, ou seja, correspondem a posições que não podem ser ocupadas. As restantes posições podem corresponder a paredes ou a *corredores*. Corredores (ou espaços vazios) são posições passíveis de serem ocupadas por unidades que se movimentam no labirinto. Cada unidade apenas ocupa uma única posição no labirinto em cada instante de tempo. O exemplo da Figura 1a) mostra um labirinto de tamanho 7×5, com paredes a toda a volta, e com duas unidades situadas nas posições $(2, 1)$ e $(4, 3)$.
A **ordem de leitura** das posições do labirinto é sempre feita da esquerda para a direita seguida de cima para baixo. Assim sendo, a ordem de leitura das unidades existentes no labirinto da Figura 1b) é a indicada pela numeração do labirinto da Figura 1c).



a) b) c)

Figura 1: a) Labirinto com duas unidades situadas nas posições $(2, 1)$ e $(4, 3)$. As posições cinzentas correspondem a paredes enquanto que as brancas correspondem a corredores. A posição destacada a vermelho indica o origem das coordenadas. b, c) Ordem de leitura das posições do labirinto.



a) Possíveis objetivos b) Caminho mínimo c) Movimento

Figura 2: Processo de movimento da unidade situada na posição (2,1).

1.2 Regras de movimento das unidades

O movimento de uma unidade é calculado através da aplicação das seguintes regras:

- Uma unidade movimenta-se dando passos. Um **passo** é definido como sendo um único movimento realizado para uma posição adjacente. O conjunto de **posições adjacentes** a uma unidade é definido como sendo as posições situadas imediatamente acima, abaixo, à direita ou à esquerda dela. As unidades não podem atravessar paredes nem outras unidades.

- Para escolher a posição seguinte de uma dada unidade, começa-se por identificar o conjunto de **possíveis posições objetivo**, sendo estas as posições adjacentes livres de cada uma das restantes unidades. Se inicialmente a unidade já se encontra adjacente a uma outra unidade, fica parada. Por exemplo, no labirinto da Figura 2a) as posições marcadas com o símbolo **?**, são as possíveis posições objetivo da unidade destacada com fundo verde e situada na posição $(2, 1)$.

- De seguida, determina-se o caminho de número **mínimo de passos** desde a unidade até a posição objetivo. A posição objetivo é aquela (dentre as possíveis posições objetivo) que se encontra a número mínimo de passos. Se mais do que uma posição objetivo estiver à mesma distância (número de passos) mínima, então a posição objetivo é a primeira seguindo a **ordem de leitura** do labirinto. Se não existe nenhum caminho possível entre a unidade e as possíveis posições objetivo, a unidade não se move. Por exemplo, no labirinto da Figura 2b) mostra-se a posição objetivo assinalada com o símbolo **X**.

- Na processo de determinação do caminho de número mínimo de passos, se múltiplas posições adjacentes colocam a unidade a exactamente a mesma distância do objetivo (múltiplos caminhos mínimos), então a posição seguinte escolhida é a primeira de acordo com a **ordem de leitura** do labirinto. Por exemplo, no labirinto da Figura 2b) mostra-se o caminho de número mínimo de passos destacado a laranja.

- Finalmente, a unidade avança uma posição seguindo o caminho de número mínimo de passos encontrado, como mostrado no labirinto da Figura 2c).

1.3 Procura do caminho de número mínimo de passos

Existem diversos algoritmos que permitem resolver o problema para a obtenção do caminho de número mínimo de passos entre duas posições num labirinto. Uma possível abordagem é o algoritmo de Lee[1] que se baseia no Breadth-First Search (BFS)[2], um algoritmo para atravessar ou procurar em grafos.

O algoritmo BFS baseia-se em garantir a exploração, ou visita, em primeiro lugar de todas as posições atingíveis com igual número de passos, antes de passar a explorar posições atingíveis com um número de passos maior. Para isso, recorre a uma estrutura de dados linear conhecida como *fila* onde as novas posições a serem exploradas são acrescentadas no final da fila. Chamamos a esta estrutura a **fila de exploração**. Inicialmente, a fila de exploração contém apenas a posição inicial. De seguida, o algoritmo processa em ciclo as posições encontradas na fila de exploração retirando sempre aquela que se encontra na primeira posição até atingir a condição de terminação (por exemplo, a posição retirada é a de destino) ou até que a fila de exploração se encontrar vazia (neste caso, não se conseguiu atingir a condição de terminação). Para cada posição da fila de exploração, o ciclo de processamento começa por verificar se a posição já foi *visitada* previamente. Se sim, a posição é retirada da fila e passa-se à próxima posição. Se não, explora-se a posição e assinala-se como visitada. Para que isto seja possível, é preciso guardar as posições já visitadas numa estrutura chamada **posições visitadas**. A exploração de uma posição consiste em acrescentar à fila de exploração as posições adjacentes vazias. De forma a poder recuperar o caminho quando atinjamos a posição destino, é necessário guardar a **sequência de posições** adjacentes que nos levou até cada posição explorada.

Neste primeiro projecto de FP, o objetivo é encontrar o **caminho** de número mínimo de passos desde uma posição dada até uma das possíveis posições objetivo, de acordo com as regras de desempate e de movimento descritas na secção anterior. O algoritmo BFS garante que a primeira posição visitada que corresponder a uma das possíveis posições objetivo encontra-se a número mínimo de passos. Por outro lado, para garantir que se cumprem as regras de desempate na escolha da posição seguinte e da posição objetivo, basta que na fase de exploração sejam acrescentadas as posições à fila de exploração pela ordem adequada (de menor a maior ordem de leitura). O pseudo-código correspondente é descrito no Algoritmo 1.

2 Trabalho a realizar

O objetivo do primeiro projecto é escrever um programa em Python, correspondendo às funções descritas nesta secção, que permita movimentar uma unidade num labirinto conforme descrito anteriormente. Para isso, deverá definir o conjunto de funções solicitadas, assim como eventualmente algumas funções auxiliares adicionais.

[1] `https://en.wikipedia.org/wiki/Lee_algorithm`
[2] `https://en.wikipedia.org/wiki/Breadth-first_search`

Algoritmo 1: Caminho de número mínimo de passos

```
Entrada: Labirinto com unidades. Uma unidade
Saída  : Caminho de número mínimo de passos
inicialização estruturas;
obter objetivos possíveis;
enquanto a lista de exploração não for vazia repete:
    posição_atual, caminho_atual ↤ primeira posição e caminho da lista de
     exploração;
    se posição_atual não foi visitada anteriormente :
        atualiza posições visitadas;
        atualiza caminho_atual;
        se posição_atual é um dos objetivos :
            resultado ↤ caminho_atual;
            sair do ciclo;
        se não
            por cada posição_adjacente à posição_atual repete:
                se posição_adjacente é livre :
                    acrescenta posição_adjacente e caminho_atual à lista de
                     exploração;
                fim
            fim
        fim
    fim
fim
```

2.1 Representação do labirinto e das unidades

Considere que um *labirinto* é representado internamente (ou seja, no seu programa) por um tuplo com N_x tuplos, cada um deles contendo N_y valores inteiros. Os valores inteiros representam cada uma das posições do labirinto podendo tomar valores igual a 1 ou 0 dependendo se a posição corresponder a uma parede ou não. Assim, sem ter em conta as unidades, o labirinto da Figura 1a) é definido pelo tuplo `((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))`.

Considere também que as posições ocupadas pelas unidades presentes num labirinto são representadas internamente por um *conjunto de posições* correspondente a um tuplo de posições. Uma *posição* é representada internamente como um tuplo de dois inteiros correspondendo o primeiro valor à posição no eixo de abcissas x e o segundo valor à posição no eixo de ordenadas y. Assim, o *conjunto de posições* que corresponde às unidades presentes na Figura 1a) será dado pelo tuplo `((2,1),(4,3))`.

As funções a escrever durante o projecto são as seguintes:

2.1.1 eh_labirinto: universal → booleano (2 valores)

Esta função recebe um argumento de qualquer tipo e devolve `True` se o seu argumento corresponde a um labirinto e `False` caso contrário, sem nunca gerar erros. Nesta parte do projecto, considere que um labirinto corresponde a um tuplo contendo N_x tuplos, cada um deles contendo N_y inteiros, com valores 0 ou 1. O labirinto de tamanho mínimo é de dimensão 3×3. Num labirinto, as posições externas correspondem sempre a paredes.

```
>>> maze = ((1,1,1,1),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1,1))
>>> eh_labirinto(maze)
True
>>> maze = ((1,1,1,1),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1))
>>> eh_labirinto(maze)
False
>>> maze = ((0,0,0,0),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1,1))
>>> eh_labirinto(maze)
False
```

2.1.2 eh_posicao: universal → booleano (1 valores)

Esta função recebe um argumento de qualquer tipo e devolve `True` se o seu argumento corresponde a uma posição e `False` caso contrário, sem nunca gerar erros. Nesta parte do projecto, considere que uma posição corresponde a um tuplo contendo 2 valores inteiros não negativos correspondendo a uma posição (x, y) num labirinto.

```
>>> eh_posicao((0,2))
True
>>> eh_posicao((1,-2))
False
>>> eh_posicao((1,1,2))
False
```

2.1.3 eh_conj_posicoes: universal → booleano (1 valores)

Esta função recebe um argumento de qualquer tipo e devolve `True` se o seu argumento corresponde a um conjunto de posições únicas e `False` caso contrário, sem nunca gerar erros. Nesta parte do projecto, considere que um conjunto de posições corresponde a um tuplo contendo 0 ou mais posições diferentes.

```
>>> eh_conj_posicoes(((1,1),(2,2)))
True
>>> eh_conj_posicoes(((1,1),(-5,5)))
False
```

```
>>> eh_conj_posicoes(((1,1),(2,2),(2,2)))
False
```

2.1.4 tamanho_labirinto: labirinto → tuplo (0.5 valores)

Esta função recebe um labirinto e devolve um tuplo de dois valores inteiros correspondendo o primeiro deles à dimensão N_x e o segundo à dimensão N_y do labirinto. Se o argumento dado for inválido, a função deve gerar um erro com a mensagem `'tamanho_labirinto: argumento invalido'`.

```
>>> maze = ((1,1,1,1),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1,1))
>>> tamanho_labirinto(maze)
(5, 4)
>>> maze = ((1,1,1,1),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1))
>>> tamanho_labirinto(maze)
Traceback (most recent call last): <...>
ValueError: tamanho_labirinto: argumento invalido
```

2.1.5 eh_mapa_valido: labirinto ×conj_posicoes → booleano (1.5 valores)

Esta função recebe um labirinto e um conjunto de posições correspondente às unidades presentes no labirinto, e devolve `True` se o segundo argumento corresponde a um conjunto de posições compatíveis (não ocupadas por paredes) dentro do labirinto e `False` caso contrário. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'eh_mapa_valido: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1),(1,0,0,1),(1,0,0,1),(1,0,0,1),(1,1,1,1))
>>> eh_mapa_valido(maze, ((1,1),(2,2)))
True
>>> eh_mapa_valido(maze, ((1,1),(5,5)))
False
>>> eh_mapa_valido(maze, ((1,1),(-5,5)))
Traceback (most recent call last): <...>
ValueError: eh_mapa_valido: algum dos argumentos e
invalido
```

2.1.6 eh_posicao_livre: labirinto ×conj_posicoes ×posicao → booleano (1 valor)

Esta função recebe um labirinto, um conjunto de posições correspondente a unidades presentes no labirinto e uma posição, e devolve `True` se a posição corresponde a uma

posição livre (não ocupada nem por paredes, nem por unidades) dentro do labirinto e `False` caso contrário. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'eh_posicao_livre: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),
            (1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))
>>> unidades = ((2,1),(4,3))
>>> eh_posicao_livre(maze, unidades, (2,2)
True
>>> eh_posicao_livre(maze, unidades, (2,0))
False
>>> eh_posicao_livre(maze, unidades, (4,3))
False
>>> eh_posicao_livre(maze, unidades, (2,-1))
Traceback (most recent call last): <...>
ValueError: eh_posicao_livre: algum dos argumentos e invalido
>>> unidades = ((2,0), (4,3))
>>> eh_posicao_livre(maze, unidades, (2,2))
Traceback (most recent call last): <...>
ValueError: eh_posicao_livre: algum dos argumentos e invalido
```

2.1.7 posicoes_adjacentes: posicao → conj_posicoes (1 valor)

Esta função recebe uma posição e devolve o conjunto de posições adjacentes da posição em ordem de leitura de um labirinto. Se o argumento dado for inválido, a função deve gerar um erro com a mensagem `'posicoes_adjacentes: argumento invalido'`.

```
>>> posicoes_adjacentes((2,1))
((2, 0), (1, 1), (3, 1), (2, 2))
>>> posicoes_adjacentes((3,2))
((3, 1), (2, 2), (4, 2), (3, 3))
>>> posicoes_adjacentes((0,0))
((1, 0), (0, 1))
>>> posicoes_adjacentes((-1,2))
Traceback (most recent call last): <...>
ValueError: posicoes_adjacentes: argumento invalido
```

2.1.8 mapa_str: labirinto ×conj_posicoes → cad. carateres (2 valores)

Esta função recebe um labirinto e um conjunto de posições correspondente às unidades presentes no labirinto e devolve a cadeia de caracteres que as representa (a representação externa ou representação "para os nossos olhos"), de acordo com o exemplo na seguinte interacção. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'mapa_str: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),
            (1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))
>>> unidades = ((2,1),(4,3))
>>> mapa_str(maze, unidades)
'#######\n#.O...#\n#.....#\n#...O.#\n#######'
>>> print(mapa_str(maze, unidades))
#######
#.O...#
#.....#
#...O.#
#######
>>> mapa_str(maze, 3)
Traceback (most recent call last): <...>
ValueError: mapa_str: algum dos argumentos e invalido
>>> unidades = ((2, 0), (4, 3))
>>> mapa_str(maze, unidades)
Traceback (most recent call last): <...>
ValueError: mapa_str: algum dos argumentos e invalido
```

2.2 Funções de movimento

2.2.1 obter_objetivos: labirinto ×conj_posicoes ×posicao → conj_posicoes (2 valores)

Esta função recebe um labirinto, um conjunto de posições correspondente às unidades presentes no labirinto e uma posição correspondente a uma das unidades, e devolve o conjunto de posições (em qualquer ordem) não ocupadas dentro do labirinto correspondente a todos os possíveis objetivos da unidade correspondente à posição dada. Isto é, as posições livres dentro do labirinto adjacentes às restantes unidades como representado na Figura 2a). A sua função deve utilizar as funções desenvolvidas nas secções anteriores sempre que possível. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'obter_objetivos: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),
            (1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))
>>> unidades = ((2,1),(4,3))
>>> obter_objetivos(maze, unidades, (2,1))
((4, 2), (3, 3), (5, 3))
>>> obter_objetivos(maze, unidades[:1], (2,1))
()
>>> unidades = ((2,1),(5,2),(4,3))
>>> obter_objetivos(maze, unidades, (2,1))
((5, 1), (4, 2), (5, 3), (3, 3))
```

```
>>> unidades = ((4,2),(4,3))
>>> obter_objetivos(maze, unidades, (4,2))
((3, 3), (5, 3))
>>> obter_objetivos(maze, unidades, (2,2))
Traceback (most recent call last): <...>
ValueError: obter_objetivos: algum dos argumentos e invalido
```

2.2.2 obter_caminho: labirinto ×conj_posicoes ×posicao → conj_posicoes (2 valores)

Esta função recebe um labirinto, um conjunto de posições correspondente às unidades presentes no labirinto, e uma posição correspondente a uma das unidades, e devolve um conjunto de posições (potencialmente vazio caso não exista nenhuma unidade alcançável) correspondente ao caminho de número mínimo de passos desde a posição dada até à posição objetivo (ou seja, a posição mais próxima de acordo com a ordem de leitura do labirinto que se encontra ao número mínimo de passos). Um caminho é um conjunto de posições adjacentes, como o representado na Figura 2b), que começa na posição da unidade dada e termina na posição objetivo. Para encontrar o caminho deve concretizar o Algoritmo 1 descrito na secção 1.3. A sua função deve utilizar as funções desenvolvidas nas secções anteriores sempre que possível. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'obter_caminho: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),
            (1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))
>>> unidades = ((2,1),(4,3))
>>> obter_caminho(maze, unidades, (2,1))
((2, 1), (3, 1), (4, 1), (4, 2))
>>> obter_caminho(maze, unidades[:1], (2,1))
()
>>> unidades = ((2,1),(5,2),(4,3))
>>> obter_caminho(maze, unidades, (2,1))
((2, 1), (3, 1), (4, 1), (5, 1))
>>> obter_caminho(maze, unidades, (2,2))
Traceback (most recent call last): <...>
ValueError: obter_caminho: algum dos argumentos e invalido
```

2.2.3 mover_unidade: labirinto ×conj_posicoes ×posicao → conj_posicoes (2 valores)

Esta função recebe um labirinto, um conjunto de posições correspondente às unidades presentes no labirinto, e uma posição correspondente a uma das unidades, e devolve o conjunto de posições actualizado correspondente às unidades presentes no labirinto

após a unidade dada ter realizado um único movimento. A ordem das unidades deve ser mantida, isto é, o conjunto de posições de saída deve ser idêntico ao de entrada, com a excepção da posição correspondente à unidade dada, se esta se mover. Como descrito na secção 1.2, a unidade não se move se já se encontrar adjacente a uma outra unidade ou se não existir nenhuma unidade atingível. Caso contrário, a unidade avança um passo no caminho de número mínimo de passos como representado na Figura 2c). A sua função deve utilizar as funções desenvolvidas nas secções anteriores sempre que possível. Se algum dos argumentos dado for inválido, a função deve gerar um erro com a mensagem `'mover_unidade: algum dos argumentos e invalido'`.

```
>>> maze = ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),
            (1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1))
>>> unidades = ((2,1),(4,3))
>>> print(mapa_str(maze, unidades))
#######
#.O...#
#.....#
#...O.#
#######
>>> unidades = mover_unidade(maze, unidades, unidades[0])
>>> print(mapa_str(maze, unidades))
#######
#..O..#
#.....#
#...O.#
#######
>>> unidades = mover_unidade(maze, unidades, unidades[1])
>>> print(mapa_str(maze, unidades))
#######
#..O..#
#...O.#
#.....#
#######
```

```
>>> unidades = mover_unidade(maze, unidades, unidades[0])
>>> print(mapa_str(maze, unidades))
#######
#...O.#
#...O.#
#.....#
#######
>>> unidades = mover_unidade(maze, unidades, unidades[1])
>>> print(mapa_str(maze, unidades))
#######
#...O.#
#...O.#
#.....#
#######
>>> mover_unidade(maze, unidades, (2,2))
Traceback (most recent call last): <...>
ValueError: mover_unidade: algum dos argumentos e invalido
```

3 Condições de Realização e Prazos

- A entrega do 1º projecto será efetuada exclusivamente por via eletrónica. Deverá submeter o seu projecto através do sistema Mooshak, até às **17:00 do dia 31 de Outubro de 2019**. Depois desta hora, não serão aceites projectos sob pretexto algum.

- Deverá submeter um único ficheiro com extensão *.py* contendo todo o código do seu projecto. **O ficheiro de código deverá conter em comentário, na primeira linha, o número e o nome do aluno**.

- No seu ficheiro de código não devem ser utilizados caracteres acentuados ou qualquer outro carácter não pertencente à tabela ASCII. Todos os testes automáticos falharão, mesmo que os caracteres não ASCII sejam utilizados dentro de comentários ou cadeias de caracteres. Programas que não cumpram este requisito serão penalizados em três valores.

- Não é permitida a utilização de qualquer módulo ou função não disponível built-in no Python 3.

- Pode, ou não, haver uma discussão oral do trabalho e/ou uma demonstração do funcionamento do programa (será decidido caso a caso).

- Lembre-se que no Técnico, a fraude académica é levada muito a sério e que a cópia numa prova (projectos incluídos) leva à reprovação na disciplina. O corpo docente da cadeira será o único juiz do que se considera ou não copiar num projecto.

4 Submissão

A submissão e avaliação da execução do projecto de FP é feita utilizando o sistema Mooshak [3]. Para obter as necessárias credenciais de acesso e poder usar o sistema deverá:

- Obter uma password para acesso ao sistema, seguindo as instruções na página: `http://acm.tecnico.ulisboa.pt/~fpshak/cgi-bin/getpass`. A password ser-lhe-á enviada para o email que tem configurado no Fenix. Se a password não lhe chegar de imediato, aguarde.

- Após ter recebido a sua password por email, deve efetuar o login no sistema através da página: `http://acm.tecnico.ulisboa.pt/~fpshak/`. Preencha os campos com a informação fornecida no email.

- Utilize o botão *"Browse..."*, selecione o ficheiro com extensão *.py* contendo todo o código do seu projecto. O seu ficheiro *.py* deve conter a implementação das funções pedidas no enunciado. De seguida clique no botão *"Submit"* para efetuar a submissão.
  Aguarde (20-30 seg) para que o sistema processe a sua submissão!!!

- Quando a submissão tiver sido processada, poderá visualizar na tabela o resultado correspondente. Receberá no seu email um relatório de execução com os detalhes da avaliação automática do seu projecto podendo ver o número de testes passados/falhados.

- Para sair do sistema utilize o botão *"Logout"*.

Submeta o seu projecto atempadamente, dado que as restrições seguintes podem não lhe permitir fazê-lo no último momento:

- Só poderá efetuar uma nova submissão pelo menos 15 minutos depois da submissão anterior.

- Só são permitidas 10 submissões em simultâneo no sistema, pelo que uma submissão poderá ser recusada se este limite for excedido [4].

- Não pode ter submissões duplicadas, ou seja submissão igual à anterior.

- Será considerada para avaliação a última submissão (mesmo que tenha pontuação inferior a submissões anteriores). Deverá, portanto, verificar cuidadosamente que a última entrega realizada corresponde à versão do projecto que pretende que seja avaliada. Não há excepções!

[3] A versão de Python utilizada nos testes automáticos é Python 3.5.3.

[4] Note que o limite de 10 submissões simultâneas no sistema Mooshak implica que, caso haja um número elevado de tentativas de submissão sobre o prazo de entrega, alguns alunos poderão não conseguir submeter nessa altura e verem-se, por isso, impossibilitados de submeter o código dentro do prazo.

- Cada aluno tem direito a **15 submissões sem penalização** no Mooshak. Por cada submissão adicional serão descontados 0.1 valores na componente de avaliação automática.

5 Classificação

A nota do projecto será baseada nos seguintes aspetos:

1. **Avaliação automática (80%).** A avaliação da correcta execução será feita através do sistema Mooshak. O tempo de execução de cada teste está limitado, bem como a memória utilizada.
   Existem 177 casos de teste configurados no sistema: 40 testes públicos (disponibilizados na página da disciplina) valendo 0 pontos cada e 137 testes privados (não disponibilizados). Como a avaliação automática vale 80% (equivalente a 16 valores) da nota, uma submissão obtém a nota máxima de 1600 pontos.
   O facto de um projecto completar com sucesso os testes públicos fornecidos não implica que esse projecto esteja totalmente correto, pois estes não são exaustivos. É da responsabilidade de cada aluno garantir que o código produzido está de acordo com a especificação do enunciado, para completar com sucesso os testes privados.

2. **Avaliação manual (20%).** Estilo de programação e facilidade de leitura[5]. Em particular, serão consideradas as seguintes componentes:

   - Boas práticas (1.5 valores): serão considerados entre outros a clareza do código, a integração de conhecimento adquirido durante a UC e a criatividade das soluções propostas.
   - Comentários (1 valor): deverão incluir a assinatura das funções definidas, comentários para o utilizador (docstring) e comentários para o programador.
   - Tamanho de funções, duplicação de código e abstração procedimental (1 valor)
   - Escolha de nomes (0.5 valores).

6 Recomendações e aspetos a evitar

As seguintes recomendações e aspetos correspondem a sugestões para evitar maus hábitos de trabalho (e, consequentemente, más notas no projecto):

- Leia todo o enunciado, procurando perceber o objetivo das várias funções pedidas. Em caso de dúvida de interpretação, utilize o horário de dúvidas para esclarecer as suas questões.

[5]Podem encontrar algumas boas práticas relacionadas em `https://gist.github.com/ruimaranhao/4e18cbe3dad6f68040c32ed6709090a3`

- No processo de desenvolvimento do projecto, comece por implementar as várias funções pela ordem apresentada no enunciado, seguindo as metodologias estudadas na disciplina. Ao desenvolver cada uma das funções pedidas, comece por perceber se pode usar alguma das anteriores.

- Para verificar a funcionalidade das suas funções, utilize os exemplos fornecidos como casos de teste. Tenha o cuidado de reproduzir fielmente as mensagens de erro e restantes *outputs*, conforme ilustrado nos vários exemplos.

- Não pense que o projecto se pode fazer nos últimos dias. Se apenas iniciar o seu trabalho neste período irá ver a Lei de Murphy em funcionamento (todos os problemas são mais difíceis do que parecem; tudo demora mais tempo do que nós pensamos; e se alguma coisa puder correr mal, ela vai correr mal, na pior das alturas possíveis);

- Não duplique código. Se duas funções são muito semelhantes é natural que estas possam ser fundidas numa única, eventualmente com mais argumentos;

- Não se esqueça que as funções excessivamente grandes são penalizadas no que respeita ao estilo de programação;

- A atitude "vou pôr agora o programa a correr de qualquer maneira e depois preocupo-me com o estilo" é totalmente errada;

- Quando o programa gerar um erro, preocupe-se em descobrir qual a causa do erro. As "marteladas" no código têm o efeito de distorcer cada vez mais o código.