TÉCNICO LISBOA

Fundamentos da Programação
Ano lectivo 2019-20
Segundo Projecto
22 de Novembro de 2019

Técnico Battle Simulator

O segundo projecto de Fundamentos da Programação consiste em escrever um programa em Python de simulação de batalhas entre dois exércitos formados por unidades que se movimentam e atacam dentro de um labirinto 2D, podendo conter obstáculos.

1 Simulação de batalhas

1.1 Labirinto, unidades e exércitos

O **labirinto** é definido de forma idêntica à do primeiro projeto, ou seja, é uma estrutura rectangular com posições indexadas a partir do canto superior esquerdo do labirinto, com *paredes* nas posições do limite exterior, onde as restantes posições podem corresponder a paredes ou a *corredores*. Os corredores podem ou não ser ocupados por **unidades**.

Neste segundo projeto, cada unidade pertence a um de dois possíveis **exércitos**. As unidades, para além de se movimentarem no labirinto, atacam as unidades do exército contrário. Cada unidade, para além da posição no labirinto e o seu exército, é caraterizada pelos seus **pontos de vida** e pela sua **força de ataque**.

A **ordem de leitura** das posições do labirinto é definida como no primeiro projeto: da esquerda para a direita seguida de cima para baixo.

1.2 Turno, movimento e ataque das unidades

A simulação de uma batalha consiste na execução de múltiplos **turnos de batalha**, até que um dos dois exércitos ganhe, isto é, até todas as unidades de um exército terem sido eliminadas, ou até não existirem mais movimentos possíveis. Em cada turno de batalha, cada unidade –seguindo a ordem de leitura do labirinto– realiza um movimento e um ataque.

O **movimento** consiste num único passo para uma posição adjacente seguindo as regras de movimento das unidades descritas no primeiro projeto, com a única diferença de que apenas são consideradas como **possíveis posições objetivo** as posições adjacentes livres de cada uma das unidades *inimigas* (isto é, do exército contrário). Tal como no primeiro projeto, uma unidade fica parada se já se encontra numa posição adjacente a um inimigo ou se não existir nenhuma posição objetivo alcançável.

Após ter completado um movimento, se a unidade se encontra adjacente a pelo menos uma unidade inimiga, então realiza um **ataque**. Se existir mais de uma unidade inimiga adjacente, o alvo do ataque é a primeira de acordo com a ordem de leitura do mapa. O efeito de um ataque consiste em subtrair os pontos de força de ataque da unidade atacante aos pontos de vida da unidade atacada. Uma unidade atacada que fique sem pontos de vida é eliminada, deixando de existir no turno de batalha corrente bem como nos subsequentes.

2 Trabalho a realizar

O objetivo deste segundo projecto é definir um conjunto de Tipos Abstratos de Dados (TAD) que deverão ser utilizados para representar a informação necessária, bem como um conjunto de funções adicionais que permitirão executar corretamente o simulador de batalhas.

2.1 Tipos Abstratos de Dados

Atenção:

- Apenas os construtores e as funções para as quais a verificação da correção dos argumentos é explicitamente pedida devem verificar a validade dos argumentos.
- Os modificadores, e as funções de alto nível que os utilizam, alteram de modo destrutivo o seu argumento.
- Todas as funções de alto nível (ou seja, que não correspondem a operações básicas) devem respeitar as barreiras de abstração.

2.1.1 TAD *posicao* (1.5 valores)

O TAD *posicao* é usado para representar uma posição (x, y) de um labirinto arbitrariamente grande, sendo x e y dois valores inteiros não negativos. As operações básicas associadas a este TAD são:

- Construtor
  - *cria_posicao*: $\mathbb{N}^2 \mapsto$ *posicao*
    cria_posicao(x,y) recebe os valores correspondentes às coordenadas de uma posição e devolve a posição correspondente. O construtor verifica a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_posicao: argumentos invalidos'` caso os seus argumentos não sejam válidos.
  - *cria_copia_posicao*: *posicao* $\mapsto$ *posicao*
    cria_copia_posicao(p) recebe uma posição e devolve uma cópia nova da posição.
- Seletores
  - *obter_pos_x*: *posicao* $\mapsto \mathbb{N}$
    obter_pos_x(p) devolve a componente x da posição p.
  - *obter_pos_y*: *posicao* $\mapsto \mathbb{N}$
    obter_pos_y(p) devolve a componente y da posição p.
- Reconhecedor
  - *eh_posicao*: *universal* $\mapsto$ *booleano*
    eh_posicao(arg) devolve `True` caso o seu argumento seja um TAD *posicao* e `False` caso contrário.

- Teste
  - *posicoes_iguais*: *posicao* $\times$*posicao* $\mapsto$ *booleano*
    posicoes_iguais(p1, p2) devolve `True` apenas se $p1$ e $p2$ são posições iguais.
- Transformador
  - *posicao_para_str*: *posicao* $\mapsto$ *str*
    posicao_para_str(p) devolve a cadeia de caracteres `'(x, y)'` que representa o seu argumento, sendo os valores x e y as coordenadas de p.

As funções de alto nível associadas a este TAD são:

- *obter_posicoes_adjacentes*: *posicao* $\mapsto$ *tuplo de posicoes*
  obter_posicoes_adjacentes(p) devolve um *tuplo* com as posições adjacentes à posição p de acordo com a ordem de leitura de um labirinto.

Exemplos de interacção:

```
>>> p1 = cria_posicao(-1, 2)
Traceback (most recent call last): <...>
ValueError: cria_posicao: argumentos invalidos
>>> p1 = cria_posicao(2, 3)
>>> p2 = cria_posicao(7, 0)
>>> posicoes_iguais(p1, p2)
False
>>> posicao_para_str(p1)
'(2, 3)'
>>> tuple(posicao_para_str(p) for p in obter_posicoes_adjacentes(p2))
('(6, 0)', '(8, 0)', '(7, 1)')
```

2.1.2 TAD *unidade* (3 valores)

O TAD *unidade* é usado para representar as unidades de combate no simulador de batalhas presentes num labirinto. Cada unidade é caracterizada pela sua posição, força de ataque, pontos de vida e exército. A força de ataque e os pontos de vida são valores inteiros positivos e o exército é qualquer cadeia de caracteres não vazia. As operações básicas associadas a este tipo são:

- Construtor
  - *cria_unidade*: *posicao* $\times\mathbb{N}\times\mathbb{N}\times$*str* $\mapsto$ *unidade*
    cria_unidade(p, v, f, str) recebe uma *posicao* p, dois valores inteiros maiores que 0 correspondentes à vida e força da unidade, e uma cadeia de caracteres não vazia correspondente ao exército da unidade; e devolve a *unidade* correspondente. O construtor verifica a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_unidade: argumentos invalidos'` caso os seus argumentos não sejam válidos.

  - *cria_copia_unidade*: *unidade* $\mapsto$ *unidade*
    cria_copia_unidade(u) recebe uma *unidade* u e devolve uma nova cópia da *unidade*.

- Seletores
  - *obter_posicao*: *unidade* $\mapsto$ *posicao*
    obter_posicao(u) devolve a *posição* da *unidade* u.
  - *obter_exercito*: *unidade* $\mapsto$ *str*
    obter_exercito(u) devolve a cadeia de carateres correspondente ao exército da *unidade*.
  - *obter_forca*: *unidade* $\mapsto \mathbb{N}$
    obter_forca(u) devolve o valor corresponde à força de ataque da *unidade*.
  - *obter_vida*: *unidade* $\mapsto \mathbb{N}$
    obter_vida(u) devolve o valor corresponde aos pontos de vida da *unidade*.

- Modificadores
  - *muda_posicao*: *unidade* $\times$*posicao* $\mapsto$ *unidade*
    muda_posicao(u, p) modifica destrutivamente a *unidade* u alterando a sua posição com o novo valor p, e devolve a própria *unidade*.
  - *remove_vida*: *unidade* $\times\mathbb{N} \mapsto$ *unidade*
    remove_vida(u, v) modifica destrutivamente a *unidade* u alterando os seus pontos de vida subtraindo o valor v, e devolve a própria *unidade*.

- Reconhecedor
  - *eh_unidade*: *universal* $\mapsto$ *booleano*
    eh_unidade(arg) devolve `True` caso o seu argumento seja um TAD *unidade* e `False` caso contrário.

- Teste
  - *unidades_iguais*: *unidade* $\times$*unidade* $\mapsto$ *booleano*
    unidades_iguais(u1, u2) devolve `True` apenas se $u1$ e $u2$ são unidades iguais.

- Transformadores
  - *unidade_para_char*: *unidade* $\mapsto$ *str*
    unidade_para_char(u) devolve a cadeia de caracteres dum único elemento, correspondente ao primeiro carácter em maiúscula do exército da *unidade* passada por argumento.
  - *unidade_para_str*: *unidade* $\mapsto$ *str*
    unidade_para_str(u) devolve a cadeia de caracteres que representa a *unidade* como mostrado nos exemplos a seguir.

As funções de alto nível associadas a este TAD são:

- *unidade_ataca*: *unidade* $\times$*unidade* $\mapsto$ *booleano*

  unidade_ataca(u1, u2) modifica destrutivamente a unidade $u2$ retirando o valor de pontos de vida correspondente à força de ataque da unidade $u1$. A função devolve `True` se a unidade $u2$ for destruída ou `False` caso contrário.

- *ordenar_unidades*: *tuplo unidades* $\mapsto$ *tuplo unidades*

  ordenar_unidades(t) devolve um tuplo contendo as mesmas unidades do tuplo fornecido como argumento, ordenadas de acordo com a ordem de leitura do labirinto.

Exemplos de interacção:

```
>>> p = cria_posicao(2, 3)
>>> u1 = cria_unidade(p, 30, -5, 'elfos')
Traceback (most recent call last): <...>
ValueError: cria_unidade: argumentos invalidos
>>> u1 = cria_unidade(p, 30, 4, 'elfos')
>>> unidade_para_str(u1)
'E[30, 4]@(2, 3)'
>>> unidade_para_char(u1)
'E'
>>> u2 = cria_copia_unidade(u1)
>>> unidades_iguais(u1, u2)
True
>>> u1 = muda_posicao(u1, cria_posicao(3, 3))
>>> unidade_para_str(u1)
'E[30, 4]@(3, 3)'
>>> unidade_para_str(u2)
'E[30, 4]@(2, 3)'
>>> unidades_iguais(u1, u2)
False
>>> tuple(unidade_para_str(u) for u in ordenar_unidades((u1, u2)))
('E[30, 4]@(2, 3)', 'E[30, 4]@(3, 3)')
>>> u2 = remove_vida(u2, 25)
>>> unidade_ataca(u1, u2)
False
>>> unidade_para_str(u2)
'E[1, 4]@(2, 3)'
>>> unidade_ataca(u1, u2)
True
```

2.1.3 TAD *mapa* (4 valores)

O TAD *mapa* é usado para representar um labirinto e as unidades que se encontram dentro do labirinto. As operações básicas associadas a este TAD são:

- Construtor
  - *cria_mapa*: *tuplo* $\times$*tuplo* $\times$*tuplo* $\times$*tuplo* $\mapsto$ *mapa*
    cria_mapa(d, w, e1, e2) recebe um tuplo d de 2 valores inteiros correspondentes às dimensões N_x e N_y do labirinto seguindo as restrições do 1º projecto, um tuplo w de 0 ou mais posições correspondentes às paredes que não são dos limites exteriores do labirinto, um tuplo $e1$ de 1 ou mais unidades do mesmo exército, e um tuplo $e2$ de um ou mais unidades de um outro exército; e devolve o mapa que representa internamente o labirinto e as unidades presentes. O construtor verifica a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_mapa: argumentos invalidos'` caso os seus argumentos não sejam válidos.
    Nota: pode assumir que os dois tuplos representam exércitos distintos, que cada tuplo só contém unidades do mesmo exército e que as unidades dos dois exércitos estão todas em posições distintas e livres dentro do labirinto.
  - *cria_copia_mapa*: *mapa* $\mapsto$ *mapa*
    cria_copia_mapa(m) recebe um *mapa* e devolve uma nova cópia do *mapa*.
- Seletores
  - *obter_tamanho*: *mapa* $\mapsto$ *tuplo*
    obter_tamanho(m) devolve um tuplo de dois valores inteiros correspondendo o primeiro deles à dimensão N_x e o segundo à dimensão N_y do mapa.
  - *obter_nome_exercitos*: *mapa* $\mapsto$ *tuplo*
    obter_nome_exercitos(m) devolve um tuplo ordenado com duas cadeias de caracteres correspondendo aos nomes dos exércitos do mapa.
  - *obter_unidades_exercito*: *mapa* $\times$*str* $\mapsto$ *tuplo unidades*
    obter_unidades_exercito(m, e) devolve um tuplo contendo as unidades do mapa pertencentes ao exército indicado pela cadeia de caracteres e, ordenadas em ordem de leitura do labirinto.
  - *obter_todas_unidades*: *mapa* $\mapsto$ *tuplo*
    obter_todas_unidades(m) devolve um tuplo contendo todas as unidades do mapa, ordenadas em ordem de leitura do labirinto.
  - *obter_unidade*: *mapa* $\times$*posicao* $\mapsto$ *unidade*
    obter_unidade(m, p) devolve a unidade do mapa que se encontra na posição p.

- Modificadores
  - *eliminar_unidade*: *mapa* $\times$*unidade* $\mapsto$ *mapa*
    eliminar_unidade(m, u) modifica destrutivamente o mapa m eliminando a unidade u do mapa e deixando livre a posição onde se encontrava a unidade. Devolve o próprio mapa.
  - *mover_unidade*: *mapa* $\times$*unidade* $\times$*posicao* $\mapsto$ *mapa*
    mover_unidade(m, u, p) modifica destrutivamente o mapa m e a unidade u alterando a posição da unidade no mapa para a nova posição p e deixando livre a posição onde se encontrava. Devolve o próprio mapa.
- Reconhecedores
  - *eh_posicao_unidade*: *mapa* $\times$*posicao* $\mapsto$ *booleano*
    eh_posicao_unidade(m, p) devolve `True` apenas no caso da posição p do mapa estar ocupada por uma unidade.
  - *eh_posicao_corredor*: *mapa* $\times$*posicao* $\mapsto$ *booleano*
    eh_posicao_corredor(m, p) devolve `True` apenas no caso da posição p do mapa corresponder a um corredor no labirinto (independentemente de estar ou não ocupado por uma unidade).
  - *eh_posicao_parede*: *mapa* $\times$*posicao* $\mapsto$ *booleano*
    eh_posicao_parede(m, p) devolve `True` apenas no caso da posição p do mapa corresponder a uma parede do labirinto.
- Teste
  - *mapas_iguais*: *mapa* $\times$*mapa* $\mapsto$ *booleano*
    mapas_iguais(m1, m2) devolve `True` apenas se $m1$ e $m2$ forem mapas iguais.
- Transformador
  - *mapa_para_str*: *mapa* $\mapsto$ *str*
    mapa_para_str(u) devolve uma cadeia de caracteres que representa o mapa como descrito no primeiro projeto, neste caso, com as unidades representadas pela sua representação externa.

As funções de alto nível associadas a este TAD são:

- obter_inimigos_adjacentes: *mapa* $\times$*unidade* $\mapsto$ *tuplo unidades*
  obter_inimigos_adjacentes(m, u) devolve um tuplo contendo as unidades inimigas adjacentes à unidade u de acordo com a ordem de leitura do labirinto.
- obter_movimento: *mapa* $\times$*unidade* $\mapsto$ *posicao*
  obter_movimento(m, u) devolve a posição seguinte da unidade u de acordo com as regras de movimento das unidades no labirinto. **Esta função é fornecida pelo corpo docente, e deve ser copiada para dentro do vosso programa.**

Exemplos de interacção:

```
>>> d = (7, 5)
>>> w = (cria_posicao(4,2), cria_posicao(5,2))
>>> e1 = tuple(cria_unidade(cria_posicao(p[0], p[1]), 20, 4, 'humans')
         for p in ((3, 2),(1, 1)))
>>> e2 = tuple(cria_unidade(cria_posicao(p[0], p[1]), 10, 2, 'cylons')
         for p in ((3, 1), (1, 3), (3, 3), (5, 3)))
>>> m1 = cria_mapa(d, w, e1, e2)
>>> print(mapa_para_str(m1))
#######
#H.C..#
#..H###
#C.C.C#
#######
>>> obter_nome_exercitos(m1)
('cylons', 'humans')
>>> u1 = obter_unidade(m1, cria_posicao(1,1))
>>> unidade_para_str(u1)
'H[20, 4]@(1, 1)'
>>> tuple(unidade_para_str(u) for u in \
    obter_unidades_exercito(m1, 'humans'))
('H[20, 4]@(1, 1)', 'H[20, 4]@(3, 2)')
>>> print(mapa_para_str(mover_unidade(m1, u1, cria_posicao(2,1))))
#######
#.HC..#
#..H###
#C.C.C#
#######
>>> u2 = obter_unidade(m1, cria_posicao(5,3))
>>> print(mapa_para_str(eliminar_unidade(m1, u2)))
#######
#.HC..#
#..H###
#C.C..#
#######
>>> u3 = obter_unidade(m1, cria_posicao(3,2))
>>> tuple(unidade_para_str(u) for u in obter_inimigos_adjacentes(m1,u3))
('C[10, 2]@(3, 1)', 'C[10, 2]@(3, 3)')
>>> obter_movimento(m1, u3)
(3, 2)
>>> u4 = obter_unidade(m1, cria_posicao(1,3))
>>> obter_movimento(m1, u4)
(1, 2)
```

2.2 Funções adicionais

2.2.1 calcula_pontos: *mapa* $\times$ *str* $\mapsto$ *int* (0.5 valores)

Função auxiliar que recebe um mapa e uma cadeia de caracteres correspondente ao nome de um dos exércitos do mapa e devolve a sua pontuação. A pontuação dum exército é o total dos pontos de vida de todas as unidades do exército.

```
>>> d = (7, 6)
>>> w = (cria_posicao(2,3), cria_posicao(4,4))
>>> e1 = tuple(cria_unidade(cria_posicao(p[0], p[1]), 30, 5, 'elfos')
for p in ((4, 2), (5, 4)))
>>> e2 = tuple(cria_unidade(cria_posicao(p[0], p[1]), 20, 5, 'orcos')
for p in ((2, 1), (3, 4), (5, 2), (5, 3)))
>>> m1 = cria_mapa(d, w, e1, e2)
>>> print(mapa_para_str(m1))
#######
#.O...#
#...EO#
#.#..O#
#..O#E#
#######
>>> calcula_pontos(m1, 'elfos'), calcula_pontos(m1, 'orcos')
(60, 80)
```

2.2.2 simula_turno: *mapa* $\mapsto$ *mapa* (1 valor)

Função auxiliar que modifica o mapa fornecido como argumento de acordo com a simulação de um turno de batalha completo, e devolve o próprio mapa. Isto é, seguindo a ordem de leitura do labirinto, cada unidade (viva) realiza um único movimento e (eventualmente) um ataque de acordo com as regras descritas.

```
>>> print(mapa_para_str(simula_turno(m1)))
#######
#..O..#
#...EO#
#.#O.O#
#...#E#
#######
>>> calcula_pontos(m1,'elfos'), calcula_pontos(m1, 'orcos')
(50, 70)
>>> print(mapa_para_str(simula_turno(m1)))
```

```
#######
#...O.#
#..OEO#
#.#..O#
#...#E#
#######
>>> calcula_pontos(m1,'elfos'), calcula_pontos(m1, 'orcos')
(30, 60)
>>> print(mapa_para_str(simula_turno(m1)))
#######
#...O.#
#..O.O#
#.#..O#
#...#E#
#######
>>> calcula_pontos(m1,'elfos'), calcula_pontos(m1, 'orcos')
(15, 55)
>>> print(mapa_para_str(simula_turno(m1)))
#######
#...O.#
#..O.O#
#.#...#
#...#E#
#######
>>> calcula_pontos(m1,'elfos'), calcula_pontos(m1, 'orcos')
(10, 50)
```

2.2.3 simula_batalha: *str* $\times$ *booleano* $\mapsto$ *str* (2 valores)

Função principal que permite simular uma batalha completa. A batalha termina quando um dos exércitos vence ou, se após completar um turno de batalha, não ocorreu nenhuma alteração ao mapa e às unidades. A função *simula_batalha* recebe uma cadeia de caracteres e um valor booleano e devolve o nome do exército ganhador. Em caso de empate, a função deve devolver a cadeia de caracteres `'EMPATE'`. A cadeia de caracteres passada por argumento corresponde ao ficheiro de configuração do simulador. O argumento booleano ativa o modo *verboso* (`True`) ou o modo *quiet* (`False`). No modo *quiet* mostra-se pela saída *standard* o mapa e a pontuação no início da simulação e após do último turno de batalha. No modo *verboso*, mostra-se também o mapa e a pontuação após de cada turno de batalha. O exemplo seguinte mostra o conteúdo de um ficheiro de configuração:

Exemplo ficheiro configuração `'config.txt'`:

```
(7,5)
('empire', 30, 4)
('rebellion', 10, 2)
((4, 2), (4, 3))
((1, 1),)
((4, 1), (1, 3), (2, 3), (5, 3))
```

A primeira linha contêm a dimensão do mapa; a segunda o nome, pontos de vida e força de ataque das unidades do primeiro exército; a terceira o nome, pontos de vida e força de ataque das unidades do segundo exército; a quarta a representação externa das posições das paredes do mapa; a quinta a representação externa das posições das unidades do primeiro exército; e a sexta a representação externa das posições das unidades do segundo exército. Pode assumir que o ficheiro de configuração está sempre bem formado.

Exemplo

```
>>> simula_batalha('config.txt', True)
#######
#E..R.#
#...#.#
#RR.#R#
#######
[ empire:30 rebellion:40 ]
#######
#..R..#
#ER.#.#
#R..#R#
#######
[ empire:26 rebellion:36 ]
#######
#.R...#
#ER.#.#
#R..#R#
#######
[ empire:22 rebellion:32 ]
#######
#R....#
#ER.#.#
#R..#R#
#######
```

```
[ empire:16 rebellion:28 ]
#######
#R....#
#ER.#.#
#R..#R#
#######
[ empire:10 rebellion:24 ]
#######
#.....#
#ER.#R#
#R..#.#
#######
[ empire:4 rebellion:22 ]
#######
#....R#
#.R.#.#
#R..#.#
#######
[ empire:0 rebellion:18 ]
'rebellion'
>>> simula_batalha('config.txt', False)
#######
#E..R.#
#...#.#
#RR.#R#
#######
[ empire:30 rebellion:40 ]
#######
#....R#
#.R.#.#
#R..#.#
#######
[ empire:0 rebellion:18 ]
'rebellion'
```

3 Condições de Realização e Prazos

- A entrega do 2º projecto será efetuada exclusivamente por via eletrónica. Deverá submeter o seu projecto através do sistema Mooshak, até às **17:00 do dia 5 de Dezembro 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, com exceção da função `reduce` do `functools`.

- 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 [1]. 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"*.

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

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 [2].
- 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!
- 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. **Execução correta (60%).** 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 vários casos de teste configurados no sistema: testes públicos (disponibilizados na página da disciplina) valendo 0 pontos cada e testes privados (não disponibilizados). Como a avaliação automática vale 60% (equivalente a 12 valores) da nota, uma submissão obtém a nota máxima de 1200 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. **Respeito pelas barreiras de abstração (20%).** Esta componente da avaliação é feita automaticamente, recorrendo a um conjunto de *scripts* (não Mooshak) que testam o respeito pelas barreiras de abstração do código desenvolvido pelos alunos.
3. **Avaliação manual (20%).** Estilo de programação e facilidade de leitura[3]. Em particular, serão consideradas as seguintes componentes:

[2]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.

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

- Boas práticas (1.5 valores): serão considerados entre outros a clareza do código, elementos de programação funcional, integração de conhecimento adquirido durante a UC, a criatividade das soluções propostas e a escolha da representação adotada nos TADs.
- Comentários (1 valor): deverão incluir a assinatura dos TADs (incluindo representação interna adotada e assinatura das operações básicas), assim como a assinatura de cada função definida, 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.
- 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.