TÉCNICO LISBOA

Fundamentos da Programação
Ano lectivo 2020-21
Segundo Projecto
11 de Dezembro de 2020

Jogo do Moinho

Neste segundo projecto de Fundamentos da Programação os alunos irão desenvolver as funções de forma a implementar um programa em Python que permita a um jogador humano jogar o Jogo do Moinho contra o computador.

1 Descrição do jogo

O jogo do moinho é um antigo jogo tradicional de tabuleiro para dois jogadores com uma multitude de variantes dependendo do número de peças disponíveis para cada jogador e a disposição do tabuleiro[1].

1.1 O tabuleiro de jogo

O **tabuleiro** de jogo considerado é uma estrutura rectangular de tamanho 3x3. Cada **posição** do tabuleiro é indexada pela coluna e a linha que ocupa. Num tabuleiro, uma posição pode estar livre ou ocupada pela **peça** de um jogador. O exemplo da Figura 1 mostra um tabuleiro com duas peças diferentes nas posições $a1$ e $b2$.

```
   a   b   c
1 [X]-[ ]-[ ]
   | \ | / |
2 [ ]-[O]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
```

Figura 1: Tabuleiro do jogo do moinho com peças diferentes nas posições *a1* e *b2*.

A **ordem de leitura** das posições do tabuleiro é definida da esquerda para a direita seguida de cima para baixo.

[1] O jogo do moinho é conhecido com vários nomes, em Inglês, conhece-se como Nine men's Morris.

1.2 Regras do jogo

Na variante do jogo do moinho considerada, cada jogador tem três peças e o vencedor é o primeiro jogador a alinhar suas três peças numa linha vertical ou horizontal. O jogo desenvolve-se em duas fases: colocação e movimento.

- Na fase de colocação, tal como no jogo do galo, o tabuleiro começa inicialmente vazio e os jogadores colocam de forma alternada uma das suas peças no tabuleiro. Após ter colocado todas as peças, e se nenhum jogador conseguiu ganhar entretanto, começa a fase de movimento.
- Durante a fase de movimento, os jogadores continuam a alternar turnos, neste caso podendo movimentar qualquer uma das peças próprias a qualquer um dos espaços livres imediatamente adjacentes conectados por uma linha horizontal, vertical ou diagonal.

O jogo continua até que um dos jogadores consegue ganhar.

1.3 Estratégia de jogo automático

Neste projecto consideraremos estratégias de jogo diferentes dependendo da fase de jogo.

1.3.1 Fase de colocação

Na fase de colocação, o jogador computador escolherá a primeira acção disponível da lista a seguir:

1. **Vitória:**
   Se o jogador tiver duas das suas peças em linha e uma posição livre
   então deve marcar na posição livre (ganhando o jogo);
2. **Bloqueio:**
   Se o adversário tiver duas das suas peças em linha e uma posição livre
   então deve marcar na posição livre (para bloquear o adversário);
3. **Centro:**
   Se a posição central estiver livre
   então jogar na posição central;
4. **Canto vazio:**
   Se um canto for uma posição livre
   então jogar nesse canto;
5. **Lateral vazio:**
   Se uma posição lateral (que nem é o centro, nem um canto) for livre
   então jogar nesse lateral.

1.3.2 Fase de movimento

Na fase de movimento, o jogador computador utilizará o algoritmo minimax[2] para escolher o seu seguinte movimento. O minimax é um algoritmo recursivo muito utilizado em teoria de jogos que pode sumarizar-se como a escolha do melhor movimento para um próprio assumindo que o adversário irá a escolher o pior possível.

Na prática, o algoritmo minimax pode ser implementado como uma função recursiva que recebe um **tabuleiro** e o **jogador** com o turno atual. A função explora todos os movimentos legais desse jogador chamando à função recursiva com o tabuleiro modificado com um dos movimentos e o jogador adversário como novos parâmetros. No caso geral, o algoritmo escolherá/devolverá o movimento que mais favoreça o jogador do turno atual. A recursão finaliza quando existe um ganhador ou quando se atinge um nível máximo de **profundidade** da recursão. O valor que devolve a função é o **valor** do estado do tabuleiro para cada jogador, sendo positivo para estados de tabuleiro que favoreçam ao jogador 'X' e negativo se favorecem ao jogador 'O'. No projeto definimos uma função simples para o valor dum tabuleiro: +1 se o ganhador é o joqador 'X', -1 se o ganhador é jogador 'O', ou 0 se não há ganhador. Assim, no caso geral, quando é o jogador 'X' a escolher movimento, escolherá/devolverá o primeiro movimento de valor máximo, e quando é o jogador 'O' a escolher movimento, escolherá/devolverá o primeiro movimento de valor mínimo. Adicionalmente, a função recursiva pode ter como argumento uma estrutura que é utilizada para registar a sequência de **movimentos** realizados e que é atualizada na chamada à função recursiva. O pseudo-código correspondente é descrito no Algoritmo 1.

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 jogo do moinho.

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] https://en.wikipedia.org/wiki/Minimax

Algoritmo 1: Algoritmo minimax

```
Função Recusiva minimax(tabuleiro, jogador, profundidade, seq_movimentos):
    se existe um ganhador ou a profundidade é 0 :
        devolve valor_tabuleiro, seq_movimentos;
    fim
    se não
        melhor_resultado ↤ ganha o outro_jogador;
        para cada peça do jogador atual repete:
            para cada posição adjacente à peça repete:
                se posição é livre :
                    cria nova cópia do tabuleiro e atualiza com novo_movimento;
                    novo_resultado, nova_seq_movimentos ↤
                     minimax(novo_tabuleiro, outro_jogador, profundidade-1,
                     seq_movimentos + novo_movimento);
                    se (melhor_seq_movimentos não está definida) ou
                    (turno do jogador 'X' e novo_resultado > melhor_resultado) ou
                    (turno do jogador 'O' e novo_resultado < melhor_resultado) :
                        melhor_resultado, melhor_seq_movimentos ↤
                         novo_resultado, nova_seq_movimentos;
                    fim
                fim
            fim
        fim
        devolve melhor_resultado, melhor_seq_movimentos;
    fim
fim
```

2.1.1 TAD *posicao* (1.5 valores)

O TAD *posicao* é usado para representar uma posição do tabuleiro de jogo. Cada posição é caraterizada pela coluna e linha que ocupa no tabuleiro. As operações básicas associadas a este TAD são:

- Construtor
  - *cria_posicao*: $str \times str \mapsto posicao$
    cria_posicao(c,l) recebe duas cadeias de carateres correspondentes à coluna c e à linha l 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_c*: *posicao* $\mapsto$ *str*
    obter_pos_c(p) devolve a componente coluna c da posição p.
  - *obter_pos_l*: *posicao* $\mapsto$ *str*
    obter_pos_l(p) devolve a componente linha l 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 e são iguais.
- Transformador
  - *posicao_para_str*: *posicao* $\mapsto$ *str*
    posicao_para_str(p) devolve a cadeia de caracteres `'cl'` que representa o seu argumento, sendo os valores c e l as componentes coluna e linha 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 do tabuleiro.

Exemplos de interacção:

```
>>> p1 = cria_posicao('a', '4')
Traceback (most recent call last): <...>
ValueError: cria_posicao: argumentos invalidos
>>> p1 = cria_posicao('a', '2')
>>> p2 = cria_posicao('b', '3')
>>> posicoes_iguais(p1, p2)
False
>>> posicao_para_str(p1)
'a2'
>>> tuple(posicao_para_str(p) for p in obter_posicoes_adjacentes(p2))
('b2', 'a3', 'c3')
```

2.1.2 TAD *peca* (1.5 valores)

O TAD *peca* é usado para representar as peças do jogo. Cada peça é caracterizada pelo jogador a quem pertencem, podendo ser peças do jogador 'X' ou do jogador 'O'. Por conveniência, é também definido o conceito peça *livre*, que é uma peça que não pertence a nenhum jogador. As operações básicas associadas a este TAD são:

- Construtor
  - *cria_peca*: *str* $\mapsto$ *peca*
    cria_peca(s) recebe uma cadeia de carateres correspondente ao identificador de um dos dois jogadores ('X' ou 'O') ou a uma peça livre (' ') e devolve a peça correspondente. O construtor verifica a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_peca: argumento invalido'` caso o seu argumento não seja válido.
  - *cria_copia_peca*: *peca* $\mapsto$ *peca*
    cria_copia_peca(j) recebe uma peça e devolve uma cópia nova da peça.
- Reconhecedor
  - *eh_peca*: *universal* $\mapsto$ *booleano*
    eh_peca(arg) devolve `True` caso o seu argumento seja um TAD *peca* e `False` caso contrário.
- Teste
  - *pecas_iguais*: *peca* $\times$*peca* $\mapsto$ *booleano*
    pecas_iguais(j1, j2) devolve `True` apenas se $p1$ e $p2$ são peças e são iguais.
- Transformador
  - *peca_para_str*: *peca* $\mapsto$ *str*
    peca_para_str(j) devolve a cadeia de caracteres que representa o jogador dono da peça, isto é, '[X]', '[O]' ou '[ ]'.

As funções de alto nível associadas a este TAD são:

- *peca_para_inteiro*: *peca* $\mapsto$ **N**
  peca_para_inteiro(j) devolve um *inteiro* valor 1, -1 ou 0, dependendo se a peça é do jogador 'X', 'O' ou livre, respetivamente.

Exemplos de interacção:

```
>>> j1 = cria_peca('x')
Traceback (most recent call last): <...>
ValueError: cria_peca: argumento invalido
>>> j1 = cria_peca('X')
>>> j2 = cria_peca('O')
>>> pecas_iguais(j1, j2)
False
>>> peca_para_str(j1)
'[X]'
>>> peca_para_inteiro(cria_peca(' '))
0
```

2.1.3 TAD *tabuleiro* (3 valores)

O TAD *tabuleiro* é usado para representar um tabuleiro do jogo do moinho de 3x3 posições e as peças dos jogadores que nele são colocadas. As operações básicas associadas a este TAD são:

- Construtor
  - *cria_tabuleiro*: $\{\} \mapsto$ *tabuleiro*
    cria_tabuleiro() devolve um tabuleiro de jogo do moinho de 3x3 sem posições ocupadas por peças de jogador.
  - *cria_copia_tabuleiro*: *tabuleiro* $\mapsto$ *tabuleiro*
    cria_copia_tabuleiro(t) recebe um tabuleiro e devolve uma cópia nova do tabuleiro.
- Seletores
  - *obter_peca*: *tabuleiro* $\times$*posicao* $\mapsto$ *peca*
    obter_peca(t, p) devolve a peça na posição p do tabuleiro. Se a posição não estiver ocupada, devolve uma peça *livre*.
  - *obter_vetor*: *tabuleiro* $\times$*str* $\mapsto$ *tuplo de pecas*
    obter_vetor(t, s) devolve todas as peças da linha ou coluna especificada pelo seu argumento.
- Modificadores
  - *coloca_peca*: *tabuleiro* $\times$*peca* $\times$*posicao* $\mapsto$ *tabuleiro*
    coloca_peca(t, j, p) modifica destrutivamente o tabuleiro t colocando a peça j na posição p, e devolve o próprio *tabuleiro*.
  - *remove_peca*: *tabuleiro* $\times$*posicao* $\mapsto$ *tabuleiro*
    remove_peca(t, p) modifica destrutivamente o tabuleiro t removendo a peça da posição p, e devolve o próprio *tabuleiro*.

  - *move_peca*: *tabuleiro* $\times$*posicao* $\times$*posicao* $\mapsto$ *tabuleiro*
    move_peca(t, p1, p2) modifica destrutivamente o tabuleiro t movendo a peça que se encontra na posição *p1* para a posição *p2*, e devolve o próprio *tabuleiro*.

- Reconhecedor
  - *eh_tabuleiro*: *universal* $\mapsto$ *booleano*
    eh_tabuleiro(arg) devolve `True` caso o seu argumento seja um TAD *tabuleiro* e `False` caso contrário. Um tabuleiro válido pode ter um máximo de 3 peças de cada jogador, não pode conter mais de 1 peça mais de um jogador que do contrario, e apenas pode haver um ganhador em simultâneo.
  - *eh_posicao_livre*: *tabuleiro* $\times$*posicao* $\mapsto$ *booleano*
    eh_posicao_livre(t, p) devolve `True` apenas no caso da posição p do tabuleiro corresponder a uma posição livre.

- Teste
  - *tabuleiros_iguais*: *tabuleiro* $\times$*tabuleiro* $\mapsto$ *booleano*
    tabuleiros_iguais(t1, t2) devolve `True` apenas se $t1$ e $t2$ são tabuleiros e são iguais.

- Transformador
  - *tabuleiro_para_str*: *tabuleiro* $\mapsto$ *str*
    tabuleiro_para_str(t) devolve a cadeia de caracteres que representa o *tabuleiro* como mostrado nos exemplos a seguir.
  - *tuplo_para_tabuleiro*: *tuplo* $\mapsto$ *tabuleiro*
    tuplo_para_tabuleiro(t) devolve o tabuleiro que é representado pelo tuplo t com 3 tuplos, cada um deles contendo 3 valores inteiros iguais a 1, -1 ou 0, tal como no primeiro projeto[3].

As funções de alto nível associadas a este TAD são:

- *obter_ganhador*: *tabuleiro* $\mapsto$ *peca*

  obter_ganhador(t) devolve uma peça do jogador que tenha as suas 3 peças em linha na vertical ou na horizontal no tabuleiro. Se não existir nenhum ganhador, devolve uma peça *livre*.

- *obter_posicoes_livres*: *tabuleiro* $\mapsto$ *tuplo de posicoes*

  obter_posicoes_livres(t) devolve um *tuplo* com as posições não ocupadas pelas peças de qualquer um dos dois jogadores na ordem de leitura do tabuleiro.

- *obter_posicoes_jogador*: *tabuleiro* $\times$*peca* $\mapsto$ *tuplo de posicoes*

  obter_posicoes_jogador(t, j) devolve um *tuplo* com as posições ocupadas pelas peças j de um dos dois jogadores na ordem de leitura do tabuleiro.

[3]Enunciado do primeiro projeto aqui.

Exemplos de interacção:

```
>>> t = cria_tabuleiro()
>>> tabuleiro_para_str(coloca_peca(t, cria_peca('X'),
    cria_posicao('a','1')))
'   a   b   c\n1 [X]-[ ]-[ ]\n   | \\ | / |\n
2 [ ]-[ ]-[ ]\n   | / | \\ |\n3 [ ]-[ ]-[ ]'
>>> print(tabuleiro_para_str(t))
   a   b   c
1 [X]-[ ]-[ ]
   | \ | / |
2 [ ]-[ ]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
>>> print(tabuleiro_para_str(coloca_peca(t, cria_peca('O'),
cria_posicao('b','2'))))
   a   b   c
1 [X]-[ ]-[ ]
   | \ | / |
2 [ ]-[O]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
>>> print(tabuleiro_para_str(move_peca(t, cria_posicao('a','1'),
cria_posicao('b','1'))))
   a   b   c
1 [ ]-[X]-[ ]
   | \ | / |
2 [ ]-[O]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
>>> t = tuplo_para_tabuleiro(((0,1,-1),(-0,1,-1),(1,0,-1)))
>>> print(tabuleiro_para_str(t))
   a   b   c
1 [ ]-[X]-[O]
   | \ | / |
2 [ ]-[X]-[O]
   | / | \ |
3 [X]-[ ]-[O]
>>> peca_para_str(obter_ganhador(t))
'[O]'
>>> tuple(posicao_para_str(p) for p in obter_posicoes_livres(t))
('a1', 'a2', 'b3')
>>> tuple(peca_para_str(peca) for peca in obter_vetor(t, 'a'))
('[ ]', '[ ]', '[X]')
```

```
>>> tuple(peca_para_str(peca) for peca in obter_vetor(t, '2'))
('[ ]', '[X]', '[O]')
```

2.2 Funções adicionais

2.2.1 obter_movimento_manual: *tabuleiro* $\times$*peca* $\mapsto$ *tuplo de posicoes* (1.5 valores)

Função auxiliar que recebe um tabuleiro e uma peca de um jogador, e devolve um tuplo com uma ou duas posições que representam uma posição ou um movimento introduzido manualmente pelo jogador. Na fase de colocação, o tuplo contém apenas a posição escolhida pelo utilizador onde colocar uma nova peça. Na fase de movimento, o tuplo contém a posição de origem da peça que se deseja movimentar e a posição de destino. Se não for possível movimentar nenhuma peça por estarem todas bloqueadas, o jogador pode passar turno escolhendo como movimento a posição duma peça própria seguida da mesma posição que ocupa. Se o valor introduzido pelo jogador não corresponder a uma posição ou movimento válidos, a função deve gerar um erro com a mensagem `'obter_movimento_manual: escolha invalida'`. A função deve apresentar a mensagem `'Turno do jogador. Escolha uma posicao: '` ou `'Turno do jogador. Escolha um movimento: '`, para pedir ao utilizador para introduzir uma posição ou um movimento.

```
>>> t = cria_tabuleiro()
m = obter_movimento_manual(t, cria_peca('X'))
Turno do jogador. Escolha uma posicao: a1
>>> posicao_para_str(m[0])
'a1'
>>> t = tuplo_para_tabuleiro(((0,1,-1),(1,-1,0),(1,-1,0)))
>>> m = obter_movimento_manual(t, cria_peca('X'))
Turno do jogador. Escolha um movimento: b1a1
>>> posicao_para_str(m[0]), posicao_para_str(m[1])
('b1', 'a1')
>>> m = obter_movimento_manual(t, cria_peca('O'))
Turno do jogador. Escolha um movimento: a2a1
Traceback (most recent call last): <...>
ValueError: obter_movimento_manual: escolha invalida
```

2.2.2 obter_movimento_auto: *tabuleiro* $\times$*peca* $\times$*str* $\mapsto$ *tuplo de posicoes* (3 valores)

Função auxiliar que recebe um tabuleiro, uma peca de um jogador e uma cadeia de carateres representando o nível de dificuldade do jogo, e devolve um tuplo com uma ou

duas posições que representam uma posição ou um movimento escolhido automaticamente. Na fase de colocação, o tuplo contém apenas a posição escolhida automaticamente onde colocar uma nova peça seguindo as regras da secção 1.3.1. Se não for possível movimentar nenhuma peça por estarem todas bloqueadas, a função devolve como movimento a posição da primeira peça do jogador correspondente seguida da mesma posição que ocupa. Na fase de movimento, o tuplo contém a posição de origem da peça a movimentar e a posição de destino. A escolha automática do movimento depende do nível de dificuldade do jogo:

- **'facil'** (1 valor): a peça a movimentar é sempre a que ocupa a primeira posição em ordem de leitura do tabuleiro que tenha alguma posição adjacente livre. A posição de destino é a primeira posição adjacente livre.
- **'normal'** (1 valor): o movimento é escolhido utilizando o algoritmo descrito na secção 1.3.2 com nível de profundidade máximo de recursão igual a 1.
  NOTA: Este nível é equivalente a escolher o primeiro movimento possível que permita obter uma vitória~~, seguido de bloquear a vitória do adversário~~. Se não existir nenhum movimento de *vitória* ~~ou *bloqueio*~~, então é seguido o mesmo critério de escolha do nível `'facil'`.
- **'dificil'** (1 valor): o movimento é escolhido utilizando o algoritmo descrito na secção 1.3.2 com nível de profundidade máximo de recursão igual a 5.

```
>>> t = cria_tabuleiro()
>>> m = obter_movimento_auto(t, cria_peca('X'), 'facil')
>>> posicao_para_str(m[0])
'b2'
>>> t = tuplo_para_tabuleiro(((1,0,-1),(0,1,-1),(1,-1,0)))
>>> m = obter_movimento_auto(t, cria_peca('X'), 'facil')
>>> posicao_para_str(m[0]), posicao_para_str(m[1])
('a1', 'b1')
>>> m = obter_movimento_auto(t, cria_peca('X'), 'normal')
>>> posicao_para_str(m[0]), posicao_para_str(m[1])
('b2', 'a2')
>>> t = tuplo_para_tabuleiro(((1,-1,-1),(-1,1,0),(0,0,1)))
>>> m = obter_movimento_auto(t, cria_peca('X'), 'normal')
>>> posicao_para_str(m[0]), posicao_para_str(m[1])
('b2', 'c2')
>>> m = obter_movimento_auto(t, cria_peca('X'), 'dificil')
>>> posicao_para_str(m[0]), posicao_para_str(m[1])
('c3', 'c2')
```

2.2.3 moinho: *str* $\times$ *str* $\mapsto$ *str* (1.5 valores)

Função principal que permite jogar um jogo completo do jogo do moinho de um jogador contra o computador. A função recebe duas cadeias de caracteres e devolve a repre-

sentação externa da peça ganhadora ('[X]' ou '[O]'). O primeiro argumento corresponde a representação externa da peça com que deseja jogar o jogador humano, e o segundo argumento selecciona o nível de dificuldade do jogo. Se algum dos argumentos dados forem inválidos, a função deve gerar um erro com a mensagem `'moinho: argumentos invalidos'`. A função deve apresentar a mensagem `'Turno do computador (<nivel>):'`, em que `<nivel>` corresponde à cadeia de caracteres passada como segundo argumento, quando for o turno do computador.

Exemplo

```
>>> moinho('[X]', 'facil')
Bem-vindo ao JOGO DO MOINHO. Nivel de dificuldade facil.
   a   b   c
1 [ ]-[ ]-[ ]
   | \ | / |
2 [ ]-[ ]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
Turno do jogador. Escolha uma posicao: a2
   a   b   c
1 [ ]-[ ]-[ ]
   | \ | / |
2 [X]-[ ]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
Turno do computador (facil):
   a   b   c
1 [ ]-[ ]-[ ]
   | \ | / |
2 [X]-[O]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
Turno do jogador. Escolha uma posicao: a1
   a   b   c
1 [X]-[ ]-[ ]
   | \ | / |
2 [X]-[O]-[ ]
   | / | \ |
3 [ ]-[ ]-[ ]
Turno do computador (facil):
   a   b   c
1 [X]-[ ]-[ ]
   | \ | / |
2 [X]-[O]-[ ]
   | / | \ |
```

```
3 [O]-[ ]-[ ]
Turno do jogador. Escolha uma posicao: c1
   a   b   c
1 [X]-[ ]-[X]
   | \ | / |
2 [X]-[O]-[ ]
   | / | \ |
3 [O]-[ ]-[ ]
Turno do computador (facil):
   a   b   c
1 [X]-[O]-[X]
   | \ | / |
2 [X]-[O]-[ ]
   | / | \ |
3 [O]-[ ]-[ ]
Turno do jogador. Escolha um movimento: c1c2
   a   b   c
1 [X]-[O]-[ ]
   | \ | / |
2 [X]-[O]-[X]
   | / | \ |
3 [O]-[ ]-[ ]
Turno do computador (facil):
   a   b   c
1 [X]-[ ]-[O]
   | \ | / |
2 [X]-[O]-[X]
   | / | \ |
3 [O]-[ ]-[ ]
Turno do jogador. Escolha um movimento: a1b1
   a   b   c
1 [ ]-[X]-[O]
   | \ | / |
2 [X]-[O]-[X]
   | / | \ |
3 [O]-[ ]-[ ]
Turno do computador (facil):
   a   b   c
1 [O]-[X]-[O]
   | \ | / |
2 [X]-[ ]-[X]
   | / | \ |
3 [O]-[ ]-[ ]
```

```
Turno do jogador. Escolha um movimento: b1b2
   a   b   c
1 [O]-[ ]-[O]
   | \ | / |
2 [X]-[X]-[X]
   | / | \ |
3 [O]-[ ]-[ ]
'[X]'
```

3 Condições de Realização e Prazos

- A entrega do 2º projecto será efectuada exclusivamente por via eletrónica. Deverá submeter o seu projecto através do sistema Mooshak, até às **17:00 do dia 05 de Janeiro de 2021**. 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, ou seja, não são permitidos `import`, 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 cópia de um projecto.

4 Submissão

A submissão e avaliação da execução do projecto de FP é feita utilizando o sistema Mooshak[4]. Para obter as necessárias credenciais de acesso e poder usar o sistema deverá:

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

- 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 5 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 [5].

- Não pode ter submissões duplicadas, ou seja, o sistema pode recusar uma submissão caso seja igual a uma das anteriores.

- 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] 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.

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 posteriormente o respeito pelas barreiras de abstração do código desenvolvido pelos alunos.

3. **Avaliação manual (20%).** Estilo de programação e facilidade de leitura[6]. Em particular, serão consideradas as seguintes componentes:

   - 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.

[6] 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.