Blackjack Modificado

Reglas del juego

El objetivo de este juego es tener la mano cuya suma de cartas sea la mayor posible sin superar el límite (*hand_limit*).

Se cuenta con un mazo que tiene cartas del 1 al *n* y cada carta está repetida *m* veces (multiplicidad). Este mazo esta barajado.

El juego ocurre por rondas, en cada una el jugador puede realizar una de las siguientes acciones:

- **Take**: Tomar la carta que se encuentra en la cima del mazo
- **Peek**: Mirar cual es la carta que se encuentra en la cima del mazo volver a colocarla en ese lugar, esta acción tiene una recompensa negativa de "*peek_reward*" (no puede realizarse esta acción 2 veces seguidas)
- **Quit**: Abandonar el juego

Condición de finalización

El juego termina cuando ocurre una de las siguientes condiciones:

- El jugador abandona el juego, ganando una recompensa igual a la suma de las cartas que se encuentran en su mano
- El jugador toma una carta, que lo deja con estrictamente más puntos que el límite, recibiendo una recompensa de 0.
- Se acaban las cartas del mazo, el jugador recibe la recompensa equivalente a abandonar el juego.

Espacio de las observaciones (y estado)

- *valueCardsInHand*: suma de los valores de las cartas en la mano del jugador, puede tener valores entre 0 y (suma de todas las cartas)*multiplicidad.
- *deckCardsCount*: array de numpy con la cantidad de cartas que se tiene de cada valor
    - Cuando el juego termina todas las posiciones de este array deben ponerse en 0
    - Ej: si el mazo tiene como carta más grande 3 y una multiplicidad de 2
        - Al comenzar el juego se tiene: [2,2,2]

        - Si se saca una carta con valor 2 el mazo pasa a tener [2,1,2]

- *nextCardIndex*: índice de la próxima carta
    - Si no se sabe que carta sigue (no se realizó un **peek** o la última acción fue **take**) su valor es n (la carta más grande)
    - Si se sabe que carta sigue su valor es el índice de esta carta en *deckCardsCount*
    - Ej: si se tiene como mazo [3,0,1,2] (multiplicidad 3, carta más grande 4)
        - 4: no se sabe que carta sigue
        - 3: sigue la carta con valor 4

Espacio de las acciones

- 0: **take**
- 1: **peek**
- 2: **quit**

Ejemplo

Se tiene un mazo con multiplicidad 2 y carta más grande 4.

La *peek_reward* es -1 y *hand_limit* es 5.

Nota: Cuando el índice de la próxima carta (*nextCardIndex*) es 4 indica que no se sabe qué carta sigue (no se realizó un **peek** desde la última **take**).

Inicialmente el estado es - *valueCardsInHand*: 0 - *nextCardIndex*: 4 (no se sabe que carta sigue) - *deckCardsCount*: [2,2,2,2]   o Se tienen 2 de cada una de las siguientes cartas: 1, 2, 3, 4

Turno 1

Se realiza la acción 1 (**peek**)

Los siguientes estados son posibles resultados

Probabilidad 0.25 | Recompensa -1 - *valueCardsInHand*: 0 - *nextCardIndex*: 0 (la próxima carta vale 1) - *deckCardsCount*: [2,2,2,2]
Probabilidad 0.25 | Recompensa -1 - *valueCardsInHand*: 0 - *nextCardIndex*: 1 (la próxima carta vale 2) - *deckCardsCount*: [2,2,2,2]

Probabilidad 0.25 | Recompensa -1 - *valueCardsInHand*: 0 - *nextCardIndex*: 2 (la próxima carta vale 3) - *deckCardsCount*: [2,2,2,2]
Probabilidad 0.25 | Recompensa -1 - *valueCardsInHand*: 0 - *nextCardIndex*: 3 (la próxima carta vale 4) - *deckCardsCount*: [2,2,2,2]

Al azar se elige el caso en el que *nextCardIndex*: 2 (la próxima carta vale 3)

Estado: - valueCardsInHand: 0 - nextCardIndex: 2 (la próxima carta vale 3) - deckCardsCount: [2,2,2,2] Recompensa: -1

Turno 2

Se realiza la acción 0 (take)

Es siguiente estado es el único posible, por lo que es también el resultante

Probabilidad 1 | Recompensa 0 - *valueCardsInHand*: 3 - *nextCardIndex*: 4 (no se sabe que carta sigue) - *deckCardsCount*: [2,2,1,2]

Turno 3

Se realiza la acción 0 (**take**) (sin antes haber realizado un **peek**, por lo que no se sabe que carta sigue).

Los siguientes estados son posibles resultados

Probabilidad 2/(2+2+1+2) = 0.286 | Recompensa 0 - *valueCardsInHand*: 3 + 1 = 4 - *nextCardIndex*: 4 (no se sabe que carta sigue) - *deckCardsCount*: [1,2,1,2]
Probabilidad 2/(2+2+1+2) = 0.286 | Recompensa 0 - *valueCardsInHand*: 3 + 2 = 5 - *nextCardIndex*: 4 (no se sabe que carta sigue) - *deckCardsCount*: [2,1,1,2]
Probabilidad 1/(2+2+1+2) = 0.142 | Recompensa 0 - *valueCardsInHand*: 3 + 3 = 6 - *nextCardIndex*: 4 (no se sabe que carta sigue) - *deckCardsCount*: [0,0,0,0] (se superó el límite, termino el juego)
Probabilidad 2/(2+2+1+2) = 0.286 | Recompensa 0 - *valueCardsInHand*: 3 + 4 = 7

- *nextCardIndex*: 4 (no se sabe que carta sigue)
- *deckCardsCount*: [0,0,0,0] (se superó el límite, termino el juego)

Al azar se elige el caso en se saca la carta de valor 2

Estado:
- *valueCardsInHand*: 3 + 2 = 5
- *nextCardIndex*: 4 (no se sabe que carta sigue)
- *deckCardsCount*: [2,1,1,2]

Recompensa: 0

Turno 4

Se realiza la acción 2 (**quit**)

Es siguiente estado es el único posible, por lo que es también el resultante

Probabilidad 1 | Recompensa 5
- *valueCardsInHand*: 5
- *nextCardIndex*: 4 (no se sabe que carta sigue)
- *deckCardsCount*: [0,0,0,0] (termino el juego)

Termina el juego

Resumen:

Turno	**Acción**	**valueCardsIn Hand**	**nextCardIn deck**	**deckCards Count**	**Recompensa**
0	--	0	4	[2,2,2,2]	--
1	Peek	0	2	[2,2,2,2]	-1
2	Take	3	4	[2,2,1,2]	0
3	Take	2	1	[2,1,2,2]	0
4	Quit	5	4	[0,0,0,0]	5
Total					4

Ejercicio 1: Implementar el ambiente

Extender la clase BlackjackEnvAbs implementando los siguientes métodos:

get_next_state_prob (self, action)

- **Recibe**: una acción
- **Retorna**: una lista de todas las tuplas que pueden obtenerse aplicando la acción al estado del ambiente. Cada tupla tiene
    - Una tupla con
        - Estado (ver “Espacio de las observaciones (y estado)”)
        - Recompensa (numero)
        - Probabilidad (numero)

  La forma de lo retornado es similar a [
  ((estado_1, recompensa_1),probabilidad_1),
  ((estado_2, recompensa_2),probabilidad_2),
  ..
  ((estado_n, recompensa_n),probabilidad_n)
  ]
- **Notas**: se recomienda implementar la función helper _get_next_cards_prob que retorna las cartas que pueden ser tomadas junto con su probabilidad
  Ej: para el mazo [2,1] retorna np.array([(0, 2/3), (1, 1/3)])

_step(self, action)

- **Recibe**: una acción
- **Retorna**: consulta a get_next_state_prob y elige según las probabilidades una tupla con el próximo estado y su recompensa, la que retorna
  La forma de lo retornado es similar a:
  (estado, recompensa)
- **Nota**: se recomienda usar choice de numpy para elegir el estado siguiente, https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html

_is_end(self)

- Recibe: (void)
- Retorna: True si el juego termino, False sino

Notas de implementación:

- Se recomienda usar self.np_random para las operaciones sobre randoms, debido a que este respeta el uso del método seed del ambiente.

Documentación
https://docs.scipy.org/doc/numpy/reference/routines.random.html

- Se pueden correr pruebas básicas sobre el ambiente utilizando el archivo envBasicTest.py, al que también se le pueden agregar más pruebas

Ejercicio 2: Markov Decision Process

Implementar los siguientes algoritmos sobre el ambiente:

- Policy evaluation
- Value optimization

Se recomienda implementar:

get_all_states()

- Recibe: (void)
- Explora el juego a partir del estado inicial, para esto usa get_next_state_prob y set_state
- Retorna: una lista de todos los estados posibles
- Nota: para resolver mas fácilmente este método se recomienda utilizar flatten_state y unflatten_state, el primero aplana el estado a una tupla que puede ser usada en la estructura set y el secundo la des aplana.