HW1 in 046203 Planning and Reinforcement Learning

Submitter 1: David Valensi 342439643
Submitter 2: Yaniv Galron 206765646

Question 1 (Longest Increasing Subsequence):

a. We can adjust the solution presented in class for the LIS case.
   We need to make sure that the odd –even repetition holds.

   $$L_i = \begin{cases} \max\limits_{j<i}\{L_j : a_j < a_i, (a_j + a_i)\ is\ odd\} + 1 & if\ such\ exist \\ 1 & else \end{cases}$$

   Now using this recursion function we can complete the DP algorithm. The input is a sequence An of size n, L is a vector of size n an L1 = 1. For i from 2 to n we update according to the recursion function.

b. Time complexity -$O(n^2)$ compared to $O(2^n)$ of the exhaustive solution. Space complexity - $O(n)$
   compared to $O(1)$ of the exhaustive solution

Question 2 (Counting):

a. $\psi_1(2) = 1,\ \psi_2(4) = 3\ \ (3+1, 2+2, 1+3),$
   $\psi_3(2) = 0, \quad \psi_N(N) = 1, \quad \psi_1(X) = 1$

b. Let's begin with a simple example to understand. The number of possibilities to divide a natural number X into two numbers is like choosing an index from 1 to X-1 in an array of X elements (it gives two sides of at least 1 element). This number is $X - 1 = \binom{X-1}{1}$
   To divide X into 3 elements we need to choose two separate indices from 1 to X-1. This number is $\binom{X-1}{2}$.
   We can continue for all $X \geq N$: $\psi_N(X) = \binom{X-1}{N-1}$
   To compute $\psi_N(X)$ based on $\psi_{N-1}(X)$ we observe that
   $$\psi_N(X) = \binom{X-1}{N-1} = \frac{(X-1)!}{(N-1)!(X-N)!} = \frac{(X-1)!(X-N+1)}{(N-1)(N-2)!(X-N+1)!} = \frac{(X-1)!}{(N-2)!(X-N+1)!} * \frac{(X-N+1)}{N-1} =$$
   $$= \psi_{N-1}(X) * \frac{(X-N+1)}{N-1}$$
   With the base case, we have:
   $$\forall X \in \mathbb{N}: \psi_N(X) = \begin{cases} 1, & if\ N = 1 \\ \psi_{N-1}(X) * \dfrac{(X-N+1)}{N-1}, & else \end{cases}$$
   Equivalently, we could have written:
   $$\forall X \in \mathbb{N}: \psi_{N+1}(X) = \begin{cases} 1, & if\ N = 0 \\ \psi_N(X) * \dfrac{(X-N)}{N}, & else \end{cases}$$

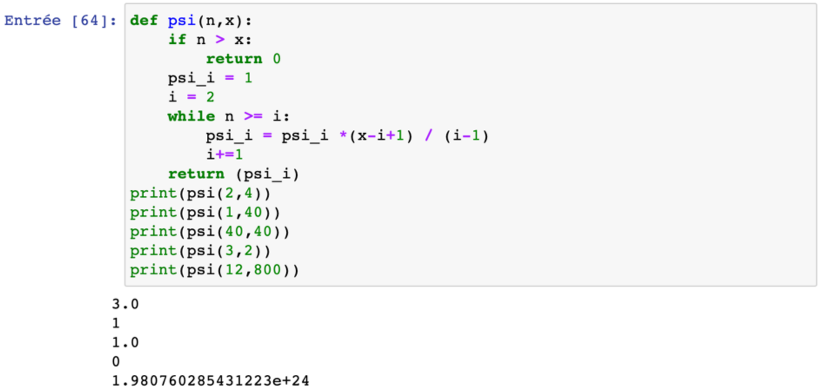
   (An alternative was to compute $\psi_N(X)$ based on $\psi_{N-1}(X)$. We have X-1 options:
   - If N-1 numbers sum up to X-1, the Nth number will be 1
   - If N-1 numbers sum up to X-2, the Nth number will be 2
   - And so on…

   Thus, we can write:
   $$\psi_N(X) = \sum_{i=1}^{X-1} \psi_{N-1}(X-i) = \sum_{i=1}^{X-1} \psi_{N-1}(i) = \psi_{N-1}(X-1) + \sum_{i=1}^{X-2} \psi_{N-1}(i)$$

   If we already computed $\psi_{N-1}(X-1)\ and\ \psi_N(X-1)$, we write:
   $\psi_N(X) = \psi_{N-1}(X-1) + \psi_N(X-1)$

   Then, adding the base cases, the recursive equation is:
   $$\forall X \in \mathbb{N}: \psi_N(X) = \begin{cases} 1 & if\ X = N\ or\ N = 1 \\ 0 & if\ N > X \\ \psi_{N-1}(X-1) + \psi_N(X-1) & else \end{cases})$$

c. We apply the recursive equation in Python:



```
Entrée [64]: def psi(n,x):
                 if n > x:
                     return 0
                 psi_i = 1
                 i = 2
                 while n >= i:
                     psi_i = psi_i *(x-i+1) / (i-1)
                     i+=1
                 return (psi_i)
             print(psi(2,4))
             print(psi(1,40))
             print(psi(40,40))
             print(psi(3,2))
             print(psi(12,800))
```

```
3.0
1
1.0
0
1.980760285431223e+24
```

1. The algorithm requires to save the single value of $\psi_{N-1}(X)$. So, space complexity $O(1)$.
We can compute each element $\psi_n(X)$ from $\psi_{n-1}(X)$ values in$O(1)$. In total it requires $O(N)$ time complexity.
2. $\psi_{12}(800) = 1.980760285431223e + 24$

d. 1. Given a cost for each natural number c_i, we formulate a finite horizon DP problem with horizon N:
State space: states represent the sum of numbers we chose until stage t.
$S_0 = \{0\}, S_N = \{X\}, S_t = \{1, \ldots, X\} \; \forall N > t > 0$
Action space: actions represent the number to sum to the actual state.
$A_t = \{1, \ldots, X\}$. But in fact, an action a_t is possible only if $s_t + a_t \leq X - (N - t)$. Else the action causes an illegal state (since at each stage we add at least 1 to the current state. Then we have:
$A_t(s_t) = \{1, \ldots, X - s_t - (N - t)\} = \{1, \ldots, X - s_t - N + t\}$.
Transition function: $f(s_t, a_t) = s_{t+1} = s_t + a_t$
Cost function: $c(s_t, a_t) = c_{a_t}$
Cumulative cost function: $C_N = \sum_{t=0}^{N-1} c(s_t, a_t) = \sum_{t=0}^{N-1} c_{a_t}$

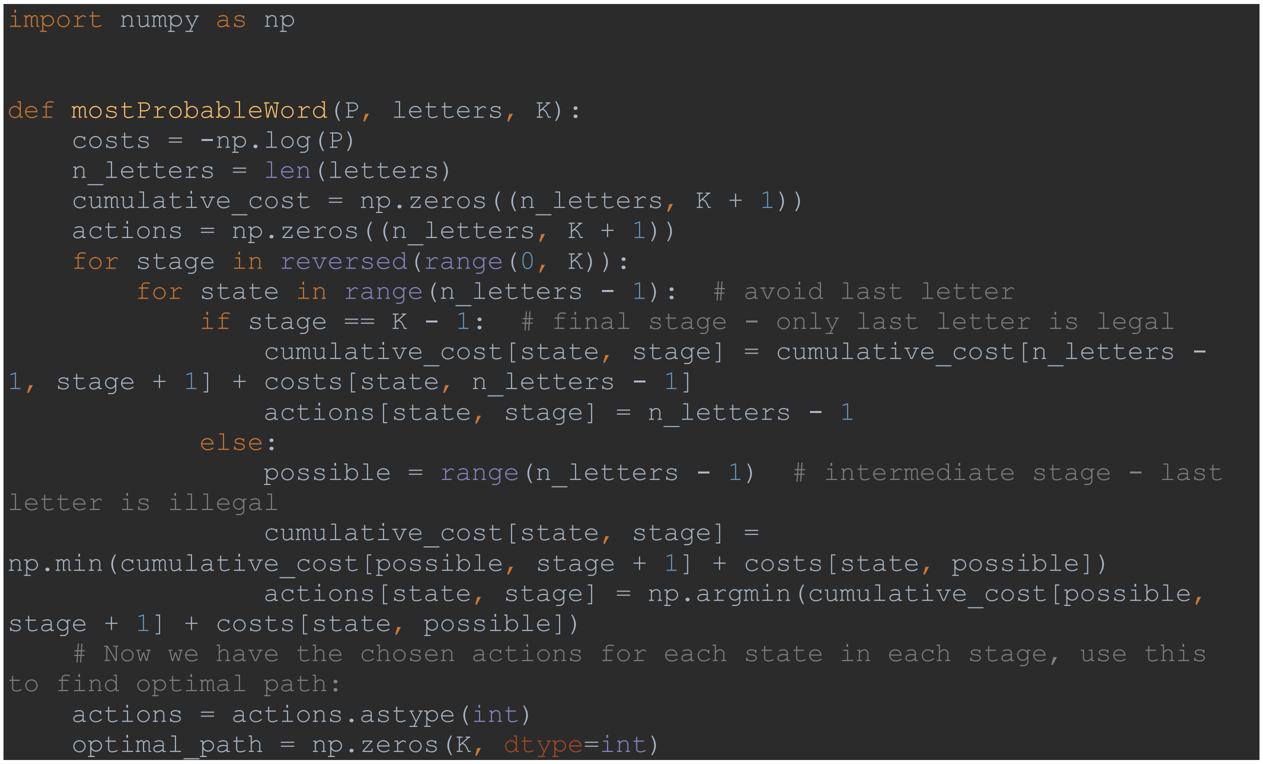
2. Obviously the number of actions $|A_t(s_t)|$ we might take at each time step from each state s_t is bounded by X. The number of states $|S_t|$ also bounded by X (let's look at the intermediate states for $N > t > 0$.
For each step t in a backward way (from N-1 to 0), we will find the best way to compose the numbers $s_t \in S_t$ from legal actions $A_t(s_t)$ based on the cumulative cost function. This optimization step takes $O(X^2)$ time.
Moreover, the number of steps is N. Then an upper bound for the time complexity of the DP backward algorithm is $O(NX^2)$.

The space complexity of such an algorithm is $O(NX)$ since for each time-step, we keep a record of the state value of up to X states.

Question 3 (Language Model):

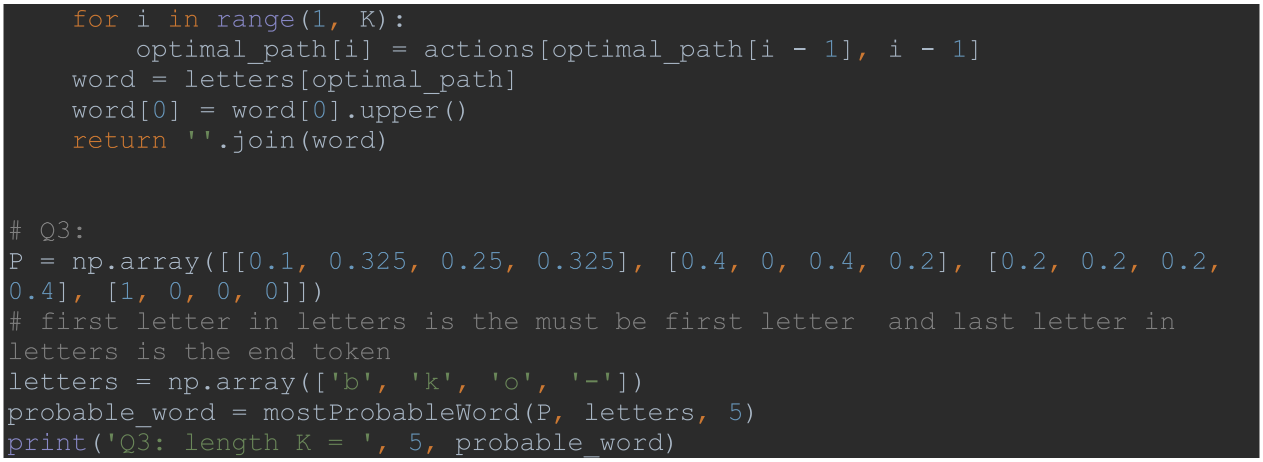
a. $P('Bob') = 0.25 * 0.2 * 0.325 = 0.01625$
$P('Koko') = 0$
$P('B') = 0.1$
$P('Bokk') = 0.25 * 0.2 * 0 * 0.2 = 0$
$P('Booo') = 0.25 * 0.2^3 * 0.4 = 0.0008$

b. 1. We will know introduce this problem as a FHDP whilst K is the number of stages and in each stage we will represent the letter taken for the most probable word. We will now define the state space, action space, transition function and multiplicative cost function.
State space - $S_0 = \{B\}, S_K = \{'-'\}, S_k = \{'b','k','o'\}$
Action space - $A_{K-1}(S_{K-1}) = \{'-'\}\ A_k(s_k) = \{'b','k','o'\}$
Transition Function - $f(s_k, a_k) = a_k$
Multiplicative cost function - $c(s_k, a_k) = \frac{1}{P(a_k|s_k)}), C_K = \Pi_{k=0}^{K-1} c(s_k, a_k)$ in this way when we try to minimize the cost we equivalently try to maximize the probability.
2.Time complexity is given by O(K*|A|*|S|)=O(9K)=O(K)
Space complexity is the total number of states which is 3(K-1) + 2 = O(K)
3. analogous problem with additive cost function is one that applies the log function to the multiplicative cost function mention in b.1.
4. the tradeoff is between computation error and memory size. When K is big we will prefer using log in order to deal with smaller numbers and when K is small we will prefer using multiplication because log will have some numerical errors.
Code:



```
import numpy as np


def mostProbableWord(P, letters, K):
    costs = -np.log(P)
    n_letters = len(letters)
    cumulative_cost = np.zeros((n_letters, K + 1))
    actions = np.zeros((n_letters, K + 1))
    for stage in reversed(range(0, K)):
        for state in range(n_letters - 1):  # avoid last letter
            if stage == K - 1:  # final stage - only last letter is legal
                cumulative_cost[state, stage] = cumulative_cost[n_letters -
1, stage + 1] + costs[state, n_letters - 1]
                actions[state, stage] = n_letters - 1
            else:
                possible = range(n_letters - 1)  # intermediate stage - last
letter is illegal
                cumulative_cost[state, stage] =
np.min(cumulative_cost[possible, stage + 1] + costs[state, possible])
                actions[state, stage] = np.argmin(cumulative_cost[possible,
stage + 1] + costs[state, possible])
    # Now we have the chosen actions for each state in each stage, use this
to find optimal path:
    actions = actions.astype(int)
    optimal_path = np.zeros(K, dtype=int)
```



```
    for i in range(1, K):
        optimal_path[i] = actions[optimal_path[i - 1], i - 1]
    word = letters[optimal_path]
    word[0] = word[0].upper()
    return ''.join(word)


# Q3:
P = np.array([[0.1, 0.325, 0.25, 0.325], [0.4, 0, 0.4, 0.2], [0.2, 0.2, 0.2,
0.4], [1, 0, 0, 0]])
# first letter in letters is the must be first letter  and last letter in
letters is the end token
letters = np.array(['b', 'k', 'o', '-'])
probable_word = mostProbableWord(P, letters, 5)
print('Q3: length K = ', 5, probable_word)
```

Result: for K =5 we get that ‘Bkbko’ is the most probable word

Question 4 (MinMax dynamic programming)

a. In this problem, the agent had to take actions that will minimize the worst-case scenario where the opponent takes the action $b_k \in B_k(s_k, a_k)$ minimizing the reward $r(s_k, a_k, b_k)$ of the agent.

Thus, for each time step we must consider all the possible opponent actions for each agent actions and take the action that will maximize the minimal reward following opponent action.

Formally, let's define the value function $V_k(s)$ which represents the maximal reward that the agent can achieve if the opponent always acts in the worst way for the agent (and in the best way for him).

Initialize: $V_N(s) = R_N(S_N)$

Backward recursion:

$\forall k = N-1, \dots, 0$: $\forall s_k \in S_k$:

$$V_k(s_k) = \max_{a_k \in A_k} \left\{ \min_{b_k \in B_k(s_k,a_k)} \{ r(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right) \} \right\}$$

$$\pi_k(s_k) \operatorname*{argmax}_{a_k \in A_k} \left\{ \min_{b_k \in B_k(s_k,a_k)} \{ r(s_k, a_k, b_k) + V_{k+1}\left(f_k(s_k, a_k, b_k)\right) \} \right\}$$

b. For simplicity, we denote $|S_k| = S, |A_k| = A,\ |B_k| = B.$

For each time step, we compute the value function $V_k(s_k)$ of all states. Each $V_k(s_k)$ requires checking all combinations of (a_k, b_k).

Thus, the time complexity is $O(N * S * A * B)$

And the space complexity is $O(N * S)$ since there are N*S state values.

c. Yes, this approach could be used to solve the tic-tac-toe game. The number of possibilities adopting a dynamic programming approach is conceivable.

The horizon would be N=9 steps.

State Space: The states would be the possible board states: in each board square, there are 3 options, there might be nothing, a X or a O. If so, there are 3^9 states. Of course, all the states are not legal at every time step (for example at time t=1, only states with 1 filled square are legal). Final states are every state where there are 3 X's (O's) in a row/column/diagonal and not 3 O's(X's) in a row/column/diagonal at the same time.

Action Space: W.l.o.g., the agent plays X's. In state s, all the possible actions are to place X in empty squares. There is up to 9 possible actions for the agent at each time step. Same for the opponent.

Reward: the agent receives 1 if the game gets into a winning state for the X's (there are 3 X's in a row/column/diagonal and not 3 O's in a row/column/diagonal), it receives -1 if the game gets into a winning state for the O's. Otherwise, the reward is 0.

d. In Chess game, the dynamic programming approach is unconceivable. First the state space is absolutely huge: for a single player with 16 pieces, the number of possible chess boards (64 squares) is 16^{64}. For 2 players it gets to $(2 * 16)^{64} = 2^{320}$. So any computer cannot compute and even store the value function for all the states explicitly, like we did in a.

Moreover, in the chess game, the horizon is not constant, and the game may never end. This makes the Dynamic Programming approach infeasible (at least the backward recursion we learned).

Question 5 (Two traversals maximal score):

a. The naïve enumeration approach examines every single combination of two traversal trajectories. There are 9 possible steps/actions for the two traversal at each row. Then the total number of trajectories to check by enumeration would be $O(9^R)$.

b. Let's denote the matrix containing the scores by:
matrix[i,j]= score of getting to cell i,j
Horizon = R (number of rows)
State space $S_t = S\{\, c_a, c_b,\ 1 \le c_a, c_b \le C\}$
Action space: $A_t = A = \{0, -1, +1\}^2$
Transition function:
We use the notation $\{C\} = \{1,2,\dots,C\}$

$$f((c_a, c_b), (a_a, a_b)) = \begin{cases} c_a, c_b, & if\ c_a + a_a \notin \{C\}\ and\ c_b + a_b \notin \{C\} \\ c_a, c_b + a_b, & if\ c_a + a_a \notin \{C\}\ and\ c_b + a_b \in \{C\} \\ c_a + a_a, c_b, & if\ c_a + a_a \in \{C\}\ and\ c_b + a_b \notin \{C\} \\ c_a + a_a, c_b + a_b, & if\ c_a + a_a \in \{C\}\ and\ c_b + a_b \in \{C\} \end{cases}$$

Instantaneous reward : $r_t(c_a, c_b) = \begin{cases} matrix[t, c_a] + matrix[t, c_b], if\ c_a \neq c_b \\ matrix[t, c_a], & , if\ c_a = c_b \end{cases}$

Cumulative reward: $V_t(c_a, c_b) = \sum_{t'=1}^{t} r_{t'}(c_a, c_b)$

$$V_t(c_a, c_b) = \max_{a_a, a_b}\left\{ V_{t+1}\left(f\big((c_a, c_b), (a_a, a_b)\big)\right) + r_t(c_a, c_b)\ \right\}$$

We could use the backward recursion to solve this from the last row.

Complexity: There are R rows.
The total number of trajectories for one traversal is $O(R * C)$ since for each row, there are C columns. Each position for the left traversal also enables the C positions for the right traversal in every row. Therefore, we multiply these possibilities and the maximal number of trajectories for both traversals together are $O(R * C^2)$. Each computation of $V_t(c_a, c_b)$ takes $9 * O(1) = O(1)$ time (there are at most 9 combinations of (a_t, b_t)). Then, the time complexity for the backward recursion is $O(R * C^2)$. Same for the space complexity.

We can optimize the above algorithm if R<C.
For the left traversal for example, the possible states of the cursor s_a is limited by the traveler that goes only right and then only left. The rightest cell the cursor attains is $O\left(\frac{R}{2}\right) = O(R)$. Same behavior for the right traversal.
Then for the two traversals, the number of states (s_a, s_b) per row is bounded by $O(R^2)$.
Then, the maximal number of trajectories for both traversals together are $O(R^3)$.