# למידה במערכות דינמיות אביב תש"פ

# תרגיל בית 1

מגישים:

יאיר נחום 034462796

דר ערבה 205874951

1. As described in the question this is a variation of the LIS problem.
    a. The original LIS problem DP solution is as follows:
        i. We define: $L_j = \{The\ longest\ strictly\ increasing\ subsequence\ ending\ at\ position\ j\}$
        ii. Initialize $L_1 = 1$
        iii. Iterate from 1 to n:
        $L_j = max\{L_i : i < j,\ a_i < a_j\} + 1;\ s.t: j > 1$
        iv. And the size of the longest sequence is the max between all indexes in the array.

    In LIOES with add a variation that changes the max definition to be as follows:
    Iterate from 1 to n:
    $L_j = max\{L_i : i < j,\ a_i < a_j,\ a_i + a_j\ is\ odd\} + 1;\ s.t: j > 1$

    In pseudo code:

```
// we are given an array A with size n
// define an array of size n
int L[n];
// Init the array to ones
for(i=0; i < n; i++)
        L[i] = 1;

// calculate the L value for each index
for(j=0; j < n; j++)
        for(i=0; i < j; i++)
                if ((A[i] < A[j]) and
                    ((A[i] + A[j]) mod 2 !=0) and
                    (L[i] + 1 > L[j]))
                        L[j] = L[i] + 1;

// get the max value between L values
maxL = 1;
for(i=0; i < n; i++)
        maxL = max(maxL, L[i]);

return maxL;
```

    b. The naïve algorithm complexity is exponential O($2^n$) as we go over all possible sets and check for each legality (another O(n) pass for each set). The DP solution solves the problem with a polynomial complexity O($n^2$). The space complexity is O(n) on both.

2.

a.

i. $\Psi_1(2) \ = \ 1. (as\ only\ 2\ is\ possible)$

ii. $\Psi_2(4) \ = \ 3. (1+3, 3+1, 2+2)$

iii. $\Psi_3(2) \ = \ 0. (we\ can't\ create\ X\ with\ N > X\ different\ N\ natural\ numbers)$

iv. $\Psi_N(N) \ = \ 1. (only\ 1+1+..+1\ N\ times\ is\ possible)$

v. $\Psi_1(N) \ = \ 1. (as\ only\ N\ can\ sum\ to\ N\ w/\ only\ 1\ number)$

b. $\Psi_N(X) \ = \ \sum_{i=1}^{X-1} \Psi_{N-1}(X-i) = \sum_{i=1}^{X-1} \Psi_{N-1}(i)$

As long as X > N. if X == N return 1; if X < N return 0;

We can further develop the recursive equation as also the following is correct (sub problem from the above):

$$\Psi_N(X-1) \ = \ \sum_{i=1}^{X-2} \Psi_{N-1}(X-1-i) = \sum_{i=1}^{X-2} \Psi_{N-1}(i)$$

thus:

$$\Psi_N(X) \ = \sum_{i=1}^{X-1} \Psi_{N-1}(i) \ = \Psi_{N-1}(X-1) \ + \ \sum_{i=1}^{X-2} \Psi_{N-1}(i)$$
$$= \ \Psi_{N-1}(X-1) \ + \ \Psi_N(X-1)$$

c.

exe2.m

```
N = 12;
X = 800;

% initialize array of size N x X to zeroes and the
a = zeros(N,X);
a(1,:) = 1;

for i = 2:N
    for j = 1:X
        if (i == j )
            a(i,j) = 1;
        elseif( i > j )
            a(i,j) = 0;
        else
            a(i,j) = a(i-1,j-1) + a(i,j-1);
        end
    end
end

a(N,X)
```

mmand Window

ew to MATLAB? See resources for Getting Started.

```
>> exe2

ans =

   1.9808e+24
```

The time complexity is as the space complexity $O(N * X)$.
And $\Psi_{12}(800) = 1.9808 * 10^{24}$ as shown in the code answer above.

d.

1. Our time index in the finite horizon problem goes from 1 to N.
Each action holds a cost according to the value added to the sum at that time interval $i \rightarrow c_i$.
The cumulative cost function is as according to the cost of each edges/actions cost in the path to the goal:

$$C_N = \sum_{i=0}^{N-1} C_i(s_i, a_i) + C_N(s_N)$$

We calculate from goal to start the minimum cost function $V_k$(s) from that state $s \in S_k$ to goal (final time index N) using the Bellman equation:

$$V_k(s) = \min_{a \epsilon A(s)} \{ C(s,a) + V_{k+1}(f_k(s,a)) \}, s \in S_k.$$

The $S_N$ (states at final time index) $V_N$ are initialized to 0.
We define in each time index 1 to X states (although even less are needed at last stages vs first stages).

State space $s \in S_k$ :
Each state defines/saves the sum of numbers up to that state denoted by Sum(s). It implies what is left to accumulate in future time indexes, thus we can't add any number.

Actions space a$\in A_k(s)$ :
$A_k(s)$ = $\{ a_k \in 1,..,X | X - Sum(s) - a_k \geq N - (k+1) \}$
Meaning, we can't take an action that is bigger in value to accumulation from the remaining number of transitions as we must have each transition add at least 1. So, the amount of possible actions that can be taken are decreasing as Sum(s) rises.

Transition function: $f_k(s_k, a_k) = a_k$

To calculate the minimum cost path, after we calculate $V_k(s)$ to all states, the $V_0(s_0)$ holds the minimum cost. To retrieve the path, one should follow the minimum $V_k$ from each s we get to.

2. The time complexity can be bounded as follows:
   we go through all states N*X and in each state we do max of X calculations (according to bellman equation we minimize over X next states $V_{k+1}(s)$). Thus, the time complexity is $O(X^2N)$
   The space complexity is $O(XN)$ as we save $O(1)$ memory for each state at each time index (We have the Sum and V functions' values at each state) and we have total of XN states.

3.

a.

i. P("Bob") = 0.25 * 0.2 * 0.325 = 0.01625
ii. P("Koko") = 0 (as it doesn't start w/ 'B')
iii. P("B") = 0.325
iv. P("Bokk") = 0.25 * 0.2 * 0 = 0 (as 'K' state doesn't have a directed edge to itself)
v. P("Boooo") = 0.25 * 0.2 * 0.2 * 0.2 * 0.4 = 0.0008

b.

i. We wish to find the most probable word in the language with the length K. Thus, we actually need the total probability of this word to be high as possible. Meaning, maximizing the multiplication of the probabilities.
we can define the cost of each transition to another letter as 1/P, In MATLAB, division by zero would set the value to inifinty (Inf).
We can define a finite horizon problem as follows:
The number of time transitions will be K as required to be the length.
The state space:
$S_k = {}'B','K','O'$ , s.t $k \in 1,2,3,..,K-1$
$S_0 = {}'B'$ , $S_K = {}'-'$ ,

Meaning, except for the start state (which has only 'B') and the goal time index (having only one state w/ the end word letter '-') we have all the letters in the language.
The Action space:
$$A_k(s) = {}'B','K','O', s.t: s \in S_k, k \in 0,1,2,..,K-2$$
$$A_{K-1}(s) = {}'-', s.t: s \in S_{K-1}$$
Meaning, the action is the next state we're going to. at the last time index it can be only the ending letter '-'.
Transition function:
$$f_k(s_k, a_k) = a_k$$
The cost is defined as:
$$C_K = \prod_{i=0}^{K-1} C(s_i, a_i)$$
When the cost we want to minimize is the probability inverse (as we want to minimize the difference from 1):
$C(s_i, a_i) = 1 - P(a_i|s_i)$.

ii. At end transition (K-1) and start transition we have only 3 computations. At the middle we have K-2 time transitions w/ 3 computations per state, and we have 3 states at each time index. Thus, at the middle we have 3*3*(K-2) computations.

The total amount is 2*3 + 3*3*(K-2) = 9*K - 12 (for $K \geq 2$). The 'B' word has its own cost (1- 0.325) to compare to.
To conclude we are bounded to an $O(K)$ complexity.

iii. We can apply a log operation on the cost function (As it is an injective one to one function) and make it an additive cost. This reduction is used a lot in optimization problems to minimize a sum instead of a product. So, the cost is a sum of -log(P) (see in code). BTW, it lets the log function handle the numeric problem when the P == 0 in which the log function of MATLAB would set -infinity (-Inf) for us.

iv. In case we have a very long word, the probabilities may become too small to calculate on a standard OS and CPU as we have lot of zeros after the dot (the same problem could have been if we used real values instead of probabilities and we could overflow above the maximum number). In such cases, it's better to use logarithmic values which are much smaller (by orders of magnitude) and can be added instead of multiplied (also a performance issue on CPU).

v. The most probable word with length 5 is "BKBKO" as calculated from the below MATLAB code:

```
K = 5;
P = [ 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];

% the cost function log
C = -log(P)
%C = log(1-P);
%C = 1-P;

% define Vk 2D array and init to inf
Vk = inf(3,K);
letters_group_size = size(P,1)-1;

% update for the last K-1 time index (next is '-')
for i=1:letters_group_size
    Vk(i,K) = C(i,letters_group_size+1);
end

% we go back using the bellman recursive equation
% note that we allow in the backword direction also to start
% with any character. for the HW we would start only from 'B'
% when we get the most probable word.
for k=K-1:-1:1

    % states, actions and next state function for
    % the middle time indexes
    Sk = [1:letters_group_size];
    Ak = [1:letters_group_size];
    Fk = Sk;
```

```
    % go through the states at that time index
    for i=Sk(1):Sk(end)
        costs = C(i,Ak)+ Vk(Fk,k+1)';
        Vk(i,k) = min(costs);
    end
end

%building the most probable word
path = zeros(1,K);

% we start only w/ 'B'
B_index = 1;
path(1) = B_index;

prev_index = B_index;
for k=1:K-1
    costs = C(prev_index, Ak)+ Vk(Fk,k+1)';
    [value, index] = min(costs);
    path(k+1) = index;
    prev_index = index;
end

lut = [char('B'), char('K'), char('O')];
word = lut(path);
string(word)
```

nand Window

```
ıs =

  "BKBKO"
```

## שאלה 4

נפתור את בעיית LCS עבור 3 סדרות- $X, Y, Z$, נסמן- $m = length\ X, n = length\ Y, l = length\ Z$

I. פתרון Brute Force יעשה באופן הבא-
מעבר על כל תתי הסדרות של X (קיימות $2^m$ תתי סדרות), ועבור כל תת סדרה לבדוק האם קיימת בY וב Z (מעבר על $n + l$ איברים).
סיבוכיות החישוב של הפתרון - $\boldsymbol{O((n + l)2^m)}$.

II. נכליל את המשפט מתרגול 1 עבור המקרה של 3 מחרוזות-
יהיו $X, Y, Z$ שלוש סדרות באורכים $m, n, l$ בהתאמה. יהי $W$ LCS של שלושת הסדרות הנ"ל ואורכו $k$. אזי מתקיים:

1. אם $x_m = y_n = z_l$, אזי $w_k = x_m = y_n = z_l$ ו- $W_{k-1}$ הוא LCS של $X_{m-1}, Y_{n-1}, Z_{l-1}$.
2. אם $x_m, y_n, z_l$ לא כולם שווים, אזי $w_k \neq x_m$ גורר כי $W$ הוא LCS של $X_{m-1}, Y, Z$.
3. אם $x_m, y_n, z_l$ לא כולם שווים, אזי $w_k \neq y_n$ גורר כי $W$ הוא LCS של $X, Y_{n-1}, Z$.
4. אם $x_m, y_n, z_l$ לא כולם שווים, אזי $w_k \neq z_l$ גורר כי $W$ הוא LCS של $X, Y, Z_{l-1}$.

**הוכחה:**

1. מתקיים- $x_m = y_n = z_l$
   a. נניח בשלילה כי $w_k \neq x_m = y_n = z_l$.
   אזי נוכל להוסיף לסוף W את האיבר $w_{k+1} = x_m = y_n = z_l$ (נסמן את הסדרה החדשה $\widetilde{W}$ ) ולקבל כי הסדרה $\widetilde{W}$ היא עדיין תת סדרה משותפת של $X, Y, Z$ (שכן W היא תת סדרה משותפת שלהם), אך גודלה $k + 1$ –
   $$k + 1 = \left|\{\widetilde{W}\}\right| = \left|\{W\} \bigcup x_m\right| > |\{W\}| = k$$
   בסתירה לכך ש- $W$ LCS של $X, Y, Z$.
   לכן $w_k = x_m = y_n = z_l$.
   *b.* נניח בשלילה כי $W_{k-1}$ אינה LCS של $X_{m-1}, Y_{n-1}, Z_{l-1}$.
   אזי קיימת תת סדרה $\widetilde{W}$ שגודלה $\tilde{k} - 1$ גדול מ- $k - 1$ אשר משותפת ל$X_{m-1}, Y_{n-1}, Z_{l-1}$. נוכל להוסיף לסוף הסדרה $\widetilde{W}$ את האיבר $w_{\tilde{k}} = x_m = y_n = z_l$.
   נקבל כי הסדרה החדשה היא sub sequence של $X, Y, Z$, אך גודלה $\tilde{k} > k = |\{W\}|$
   בסתירה לכך ש-$W$ LCS של $X, Y, Z$.
   לכן $W_{k-1}$ היא LCS של $X_{m-1}, Y_{n-1}, Z_{l-1}$.

2. מתקיים- $x_m, y_n, z_l$ לא כולם שווים, ו-$w_k \neq x_m$
   נניח בשלילה כי $W$ אינה LCS של $X_{m-1}, Y, Z$.

אזי קיימת תת סדרה $\widetilde{W}$ המשותפת ל-$X_{m-1}, Y, Z$ שאורכה גדול מ$k$. $\widetilde{W}$ הינה גם תת סדרה משותפת של $X, Y, Z$ , אך אורכה גדול מ-$|\{W\}| = k$ , בסתירה לכך ש-$W$ LCS של $X, Y, Z$.
לכן $W$ היא LCS של $X_{m-1}, Y, Z$.

3. הוכחה סימטרית ל-2
4. הוכחה סימטרית ל-2

III. מימוש אלגוריתם LCS בDynamic Programming עבור 3 סדרות(פסאודו קוד)-

```
LCS(X,Y,Z):
    m = X.length
    n = Y.length
    l = Z.length

    C[m][n][l] # equivalent to C array in two string LCS solution
(holds LCS length for the 3 strings until mth, nth, and lth indexes)
    B[m][n][l] # equivalent to B array in two series LCS solution
(holds '---' \'-XX' \'X-X' \'XX-' referring to actions made in
increasing the indexes- equivalent to ←\↑\↖ in two series LCS
solution from class)

    for i from 0 to m
        for j from 0 to n
            for k from 0 to l
                if i==0 || j==0 || k==0
                        C[i][j][k]=0
                else if X[i]==Y[j] && X[i]==Z[k]
                        C[i][j][k] = C[i-1][j-1][k-1] +1
                        B[i][j][k] = '---' #equivalent to '↖' in two
                      series LCS solution from class
                else
                    C[i][j][k] = max(C[i-1][j][k],
                                     C[i][j-1][k],C[i][j][k-1])
                    if C[i][j][k]==C[i-1][j][k]
                        B[i][j][k] = '-XX' # equivalent to '←' or '↑

                    else if C[i][j][k]==C[i-1][j][k]
                        B[i][j][k] = 'X-X' # equivalent to '←' or '↑

                    else
                        B[i][j][k] = 'XX-'# equivalent to '←' or '↑'

    W[C[m][n][l]] #init W to size of LCS
    i=m
    j=n
    k=l
    s=W.length

# Go over array B and receive LCS characters (X[i]= Y[j]= Z[k] at
indexes i,j,k for which B[i][j][k] '---'
```

```
while(s>0)
    if B[i][j][k] '---'
        W[s] = X[i]  # X[i]= Y[j]= Z[k]
        s--; i-- ;j--; k--;
    else if B[i][j][k] = '-XX'
        i--
    else if B[i][j][k] = 'X-X'
        j--
    else
        k--
return W #W holds LCS
```

## שאלה 5

a. אלגוריתם DP לפתרון הבעיה הנתונה-

נגדיר פונקציית ערך –

$$V_k: S_k \to R_k$$

כאשר $V_k(s_k)$ הוא התגמול המצטבר עבור מסלול $h_{k:N} = (s_k, a_k, b_k, \dots, s_n)$ אופטימלי. הכוונה במסלול $h_{k:N}$ אופטימלי היא מסלול עבורו $\min_{\pi_b}(R_N(h_{k:n}))$ מקסימלי.

כלומר, עבור החלטות המדיניות $\pi_b$ הממזערות את התגמול המצטבר, ועבור מדיניות אופטימלית $\pi_a^*$ אשר תביא את התגמול המצטבר למקסימום, $V_k(s_k)$ תתאר את ערך התגבול המתקבל.

נגדיר את $V_k$ באופן הבא-

1. $V_N(s_N) = r_N(s_N)$ לכל מצב $s_N \in S_N$ (לכל מצב בצעד האחרון)
2. באופן רקורסיבי, לכל $k \in N-1, \dots, 0$-

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

והמדיניות האופטימלית היא-

$$\boldsymbol{\pi_{a_k}^* = argmax_{a_k \in A_k}\{min_{b_k \in B_k}\{r(s_k, a_k, b_k) + V_{k+1}(f_k(s_k, a_k, b_k))\}\}}$$

b. סיבוכיות האלגוריתם היא-

מספר צעדי הזמן ($N$) כפול גודל מרחב המצב ($|S|$) כפול גודל מרחב הפעולות של שחקן a ($|A|$) כפול גודל מרחב הפעולות של שחקן b ($|B|$)

נקבל- $\boldsymbol{O(N * |S| * |A| * |B|)}$

c. <u>**ניתן**</u> לנתח את משחק האיקס עיגול באופן תואם לאלגוריתם DP הנ"ל.

**מרחב המצבים $\boldsymbol{S_k}$** הוא מצבי לוח המשחק האפשריים לאחר $k$ תורות (כלומר על הלוח $k$ צורות X או O). אם מצב הלוח $s_k \in S_k$ הוא מצב של <u>ניצחון עבור שחקן a</u> (נניח ששחקן a הוא X), אזי נגדיר- $\boldsymbol{r(s_k) = 1}$. בנוסף, אם $s_k \in S_k$ הוא <u>מצב שמוביל למצב של ניצחון לשחקן a</u> נגדיר- $r(s_k) = 1$. אם מצב הלוח $s_k \in S_k$ הוא מצב של <u>ניצחון עבור שחקן b</u> (נניח ששחקן b הוא O), אזי נגדיר - $\boldsymbol{r(s_k) = -1}$. בנוסף, אם $s_k \in S_k$ הוא מצב <u>שמוביל למצב של ניצחון לשחקן b</u> נגדיר-

$r(s_k) = -1$. עבור כל יתר מצבי הלוח (מצב סיום של תיקו/מצב שלא מכריע את המשחק) נגדיר-

$\boldsymbol{r(s_k) = 0}$.

את **מרחב הפעולות** $\boldsymbol{A_k(s_k)}$ של שחקן a במצב לוח $s_k$ נגדיר כמקומות הריקים בלוח בהן שחקן b יוכל בתורו להוסיף X. באותו אופן $B_k(s_k)$ הוא המקומות בלוח בהם שחקן b

יוכל בתורו להוסיף O. ברור כי $f_k(s_k, a_k, b_k)$ יהיה מצב הלוח לאחר ששני השחקנים הוסיפו X ו-O למצב הלוח $s_k$.

אם $s_k$ הוא מצב של סיום משחק ברור כי אין פעולות אפשריות לשני השחקנים.

לפי האלגוריתם מסעיף א', שחקן b יפעל עם מדיניות הממזערת את התגמול המצטבר, כלומר יבחר הכולל הכי הרבה מצבים המובילים לניצחונו\מונעים משחקן a לנצח. שחקן a יפעל עם מדיניות הממקסמת את התגמול המצטבר, כלומר יבחר במסלול עם הכי הרבה מצבים המובילים לניצחונו\מונעים משחקן b לנצח.

d. **<u>לא ניתן</u>** לנתח משחק שחמט לפי האלגוריתם הנ"ל מכמה סיבות.

- בשונה ממשחק איקס עיגול, משחק שחמט אינו Finite Horizon Decision Problem ויכול להימשך "לנצח" במקרים מסוימים.
- במשחק שחמט מרחב המצבים גדול מאוד. בלוח 64 משבצות ובו יכולים להיות עד 32 חיילים (16 לכל שחקן). בנוסף יש כמה סוגים של חיילים(פרש\מלך\מלכה וכו') לכן מרחב המצב המכיל את כל מצבי הלוח האפשריים גדול בהרבה ממרחב המצב עבור משחק איקס עיגול.
- במשחק שחמט מרחב הפעולות גדול מאוד. לשחקן אפשרות להזיז סוג שונה של חייל מבין 16 חיילים למספר גדול של מקומות אפשריים על הלוח.

מכל הסיבות הנ"ל, פתרון משחק השחמט לפי האלגוריתם הנ"ל יהיה חישוב ארוך של מספר עצום של משתנים לכן לא באמת אפשרי.