Homework 1

Matan Weksler – 302955372   Ido Glanz - 302568936

1. **Question 1 (Counting)**

    a. $\Psi_1(2) = 1; \ since: 2 = 2$

    $\Psi_2(4) = 3; \ 2 + 2 = 1 + 3 = 3 + 1 = 4$

    $\Psi_3(2) = 0$

    $\Psi_N(N) = 1; \ \sum_1^N 1 = N$

    $\Psi_1(X) = 1; \ X = X$

    b. $\Psi_N(X) = \begin{cases} 1; & N = 1 \ or \ N = X \\ 0; & N > X \\ \sum_{i=1}^{X-1} \Psi_{N-1}(i); & else \end{cases}$

    c. Write a code in Matlab for finding ΨN (X) using dynamic programming.

        a. What is the time and memory complexity of your algorithm? - For

        memory we use a N by X matrix, so our memory complexity is O(XN)

        and time is O(NX).

        b. Find $\Psi_{12}(800)$ -   1.9808e+24 - See attached Matlab code.

    d. We will Formulate the problem as a finite horizon decision problem based

    on the recursive formula that we developed in previous section.

    - The state space:

        Every combination in the recursive function which is called, including

        the combinations which matches a stop condition, is defined a state.

        For example, for the ith recursive call with the values $\Psi_n(x)$ we'll define

        the state: $S_i = \{x, n\}$.

        While:

        i – the index of the state.

        x – the leftover residue of X we still need to accumulate to.

        n – the number of natural numbers left for us to use to accumulate

        to x.

    - The Action: $a_n(x_i)$

        While:

        i – a natural integer, smaller than the current states' x value which is

        subtracted from it while transitioning to the next layer (the next n).

    - The system transition function: $f(S_{n,i}, a_i)$

- The cumulative cost function -

  The cost of being at a certain state via a certain path:

  $$C_n = \Sigma_{i=1}^{n} c_a(a_i)$$

  In other words, the sum of the costs of every action leading us to the
  current state (while transitioning between them) where the cost of an
  action is the cost of the natural integer we add (or subtract depending
  on direction).

b. According to the definition of the state space we can conclude that the
   given graph is actually a tree, which it's root is the state $S_0 = \{X, N\}$. Based
   on that we can use the DFS algorithm for finding the best combination.
   The DFS algorithm complexity is $O(|V| + |E|)$, and because we're running
   it on a tree (and therefor $|E| = O(|V|)$) we get a complexity of $O(|V|)$.
   Since at each "layer" on n we can derive a max of x^2 edges, our total
   number of vertices would be:

   $$|V| = N \cdot X^2$$

2. **Question 2 (Language model)**

   a. Find the probability of the following words:

   $P('\text{Bob}') = 0.25 \cdot 0.2 \cdot 0.325 = 0.01625$

   $P('\text{Koko}') = 0$

   $P('B') = 0.325;$

   $P('\text{Bokk}') = 0.25 \cdot 0.2 \cdot 0 \cdot 0.2 = 0;$

   $P('\text{Boooo}') = 0.25 \cdot 0.2 \cdot 0.2 \cdot 0.2 \cdot 0.2 \cdot 0.4 = 0.0008$

   b.

      a. We will Formulate the problem as a finite horizon decision problem.

      - <u>The state space:</u>

        The starting state will be: $S_0 = \{1, B\}$

        All the following states will be of the form: $S_i(l_i)$.

        Hence $\{i, L\}$ will be the vertice denoting all the words of size i ending
        with the letter $l \in \{B, K, O\}$.

        Important conditions:

           - $1 \leq i \leq k$

- $L \in \{B, K, O\}$

The last node will be $\{k + 1, -\}$, and all the nodes from level k will have an edge to it.

In addition, there's a last state: $\{K + 1, -\}$ which will denote the end of the word.

- The Action: $a(l)$

Where:

l is the added letter ($l \in \{B, K, O, -\}$) to the word.

- The cost function:

If $A = \{a(l_1), \dots, a(l_{k+1})\}$ is a valid path on the graph (or in other words a valid word In the language), then the cost function would be:

$$C(A) = P([B, l(2)]) \cdot \prod_{i=2}^{k-1} P([l(i), l(i + 1)]) \cdot P(\{l(k), '- '\})$$

b. According to the definition of the state space we can conclude that the given graph is a tree, with the root being $S_0 = \{1, B\}$, and the only leaf of it is $\{k + 1, -\}$. Based on that we can use a modified Dijkstra algorithm - instead of accumulating the edges, we'll use multiplication of $\frac{1}{C(A)}$ (under the assumption of 0<C(A)<1 in a feasible word). The starting node will be $\{1, B\}$, the ending node will be $\{K + 1, -\}$, and we'll search for the 'most expensive' path between the starting node to the ending node.

The Dijkstra algorithm complexity is $O(|E| \cdot \log(|V|))$.

$$|V| = 2 + 3^{k-1}$$
$$|E| = 6 + 9^{k-2}$$

c. A reduction of the problem to an analogous problem with additive cost function instead is to use the log function. This is according to the mathematic rule:

$$\log(A) + \log(B) = \log(A \cdot B)$$

This reduction is an analogous problem because:

$$A \leq B \rightarrow \log(A) \leq \log(B)$$

Hence,

$$C' = -\log(C) = \log\big(P([B, l(2)])\big) - \left(\sum_{i=2}^{k-1} log\big(P([l(i), l(i + 1)])\big)\right) - \log\big(P(\{l(k), '-'\})\big)$$

d. As the multiplicative method is rapidly increasing towards infinity, its' representation in the computer's memory would require far more space (bit wise) than using an additive cost function where the price would increase as well but almost linearly (bounded linearly by the most expansive action). That said, the log action might also require a large number of bits or else if cut might induce errors in the final solution.

e. Most probable word is 'BKBKO' at a probability of 0.0068. See attached Matlab code (HW1_Q2)

3. **Question 3 (Path planning):**

a. We will Formulate the problem as a finite horizon decision problem.

- **The state space:**

  The starting state will be: $S_0 = (1,1)$

  All the following states will be of the form: (m, n)

  While $1 \leq m \leq M, \ 1 \leq n \leq N$ denote for Moses location in the apartment.

- **The Action:**

  A step to the west/north. $a_i \in \{a_{north}, a_{east}\}$

- The transition function is: $f(S_i, a_i) = S_{i+1}$

- The cost function:

  If Moses made a step from room (m,n) to room (m+1,n) (or (m,n+1)) then the cost will be 1 if he finds a slice of cheese in the destination room, and 0 if not.

$$c = \begin{cases} 1, & cheese\ present \\ 0, & else \end{cases}$$

  Therefore, if the path Moses choose is $A = (a_1, \dots, a_k)$, then:

$$C(A) = \Sigma_{i=1}^{k} c(a_i)$$

b. Because Moses the mouse has limited travel options, to reach the goal he has to move exactly M-1 steps North and N-1 steps East - This means the time horizon is M+N-2.

c. As we saw in last section there are M+N-2 possible paths, hence the total number of paths is the number of options to overall take N-1 steps out of the

total number of steps (M+N-2). That is since once at the top Moses is out of options.

Hence: $No. paths = \binom{M + N - 2}{N - 1}$

For M=2; $No. paths = \binom{N}{N - 1} = N$

For M=N; $No. paths = \binom{2(N - 1)}{N - 1} = \frac{(2(N-1))!}{(N-1)!(N-1)!}$

    d.

1. if both mice ignore each other's existence and act 'optimal' with respect to the original problem they can reach only the max of one mouse alone.

2. Assume both mice decided to coordinate their efforts and split the loot. Now the state space would double, and each state represent the possible position of the two mice in the room.

   i.e. There are $M \cdot N$ positions in the room and a total of $(M \cdot N)^2$ ways to place the two mice.

   At each state each mouse has two possible actions and so there is

   $2 \cdot 2 = 4$ possible actions.

3. there are $(M \cdot N)^K$ ways to place K mice in the M by N room.

   and we have 2^K possible actions.

4. **Question 4 (MinMax dynamic programming)**

   a. The algorithm basically searches at each state S the value function V(s) which is the best worst-case possible (i.e. the action which leads to the least risk) in the case of starting from the given state. In aim so solve it dynamically, we generate a value for each state recursively and then generate a policy constructed of the actions leading to the least risk (as described above).

      a. First, we initialize the value at the goal/end: $V_N(S) = r_N(s), s \in S$

      b. Then, for all states s In S we calculate:
         - For all $s \in S$
           $$V_i(S) = max_a\{min_b\{r_i(s, a, b) + V_{i+1}(f_i((s, a, b)))\}$$

      c. And for the optimal policy we calculate for any I:
         $$\pi_i(s) = argmax_a\{ min_b\{r_i(s, a, b) + V_{i+1}(f_i((s, a, b)))\}$$

b. Computational complexity – as we have a total of N states (time samples) and at each time k we have A actions leading to B re-actions, the complexity would then be $O[\sum_N \sum_S \sum_A B_k(s,a)]$ which assuming non-varying values would become $O(|N||S||A||B|)$.

c. In order to use this approach to solve a tic-tac-toe game we would define each of the 126 possible states (9 over 5) as a possible state, an action would then be placing an X in a free space or non if one of the players won. The rewards would be either winning, losing or neither (in aim to generate a reward for winning but also for avoiding loss). Reward is hence a function of the given state and the action and could either be 1,0,-1 (in the same order) and the accumulative reward would also be 1,0,-1 as a game can only end with these outcomes. As for run-time, as the number of states and actions is bounded and no too high it is a feasible solution.

d. As in theory this approach could also work on a game of chess, while in tic-tac-toe the number of states and actions is relatively low, in chess the numbers are tremendous, and it would be near impossible to compute it all.