Assignment 2

Task1:

Value iteration

For value iteration initially, a random optimal policy was initialized, post that V was updated till it converged (difference between prev value of V and current <1e-7 or the max number of iterations crossed 12500) according to:

$V[s] = \max_{a \in A} \sum_{s' \in S} T[s][a][s'](R[s][a][s'] + \gamma V[s'])$

Consequently, the optimal policy is given by:

$\pi = \arg\max_{a \in A} \sum_{s' \in S} T[s][a][s'](R[s][a][s'] + \gamma V[s'])$

Howard's policy iteration

Initially, a random policy was initialized and value function was initialized according to that, then Q (action value function) was calculated and policy was calculated by switching to all improvable states possible. The algorithm is run till either the iterations have crossed 500 or diff between the values becomes smaller than 1e-5

```
policy = np.random.randint(0, A, size = S)
    V = policy_cal(policy)
    Q = np.zeros(shape = (S,A))
    V_old = np.zeros(S)
    k = 0
    while(True):
        Q = np.sum((T*(R + gamma*V)), axis = 2)
        policy = np.argmax(Q, axis = 1)
        V = policy_cal(policy)
        if np.all(np.abs(V_old - V))<1e-5 or k>5000:
            break
        V_old = V.copy()
        k = k + 1
    for j in range(0,S):
        print(V[j],policy[j])
```

Linear programming

For this algorithm, the constraints were set up and the linear equations were solved with the help of Pulp library

```
prob = pulp.LpProblem('Problem', pulp.LpMinimize)
    states = range(0,S)
    V = pulp.LpVariable.dicts("V", states, cat='Continuous')

    prob += pulp.lpSum([V[i] for i in states])

    for i in range(0,S):
        for j in range(0,A):
            prob += pulp.lpSum([T[i][j][k]*(R[i][j][k] + gamma*V[k]) for k in states]) <= V[i]

    prob.solve(pulp.apis.PULP_CBC_CMD(msg=False))
    V_soln = np.array([V[i].varValue for i in states])
    policy = np.zeros(S)
    policy = np.argmax(np.sum(T*(R + gamma*V_soln), axis = 2), axis = 1)
    f  = open("value_and_policy_file.txt", "w+")
    for s in range(0,S):
        f.write("%f %d\n" %(V_soln[s], int(policy[s])))
        print(V_soln[s]," ", int(policy[s]))
```

Task2:

MDP formulation:

Approach:

To emulate an actual cricket game where two batters are present, additional states for player B were concatenated in additional to the corresponding states of player A. Since a match of cricket can end with either them winning or losing, two more additional states were appended to account for that scenario.

Next, using the action-outcome probabilities given for both the players, the transitions were mapped taking into considerations the strike and over changes.

The following cases are possible:

- Either of the players gets out: The MDP transitions into the terminal state created with transition probabilities given
- The runs scored exceeds the runs left i.e. game terminates with reward 1
- No balls left with runs still remaining takes the MDP into losing terminal state with reward 0
- Over finishes with even runs scored by player causes strike change leading to the outgoing state to be of the opposite batsman with the respective transition probabilities given.
- Over finishes with odd runs scored by player doesn't cause strike change leading to the outgoing state to be of the same batsman with the respective transition probabilities given.
- Lastly, if the runs scored are even the batman remains same and if odd, it leads to strike change causing the outgoing transition to be of the opposite batsman.

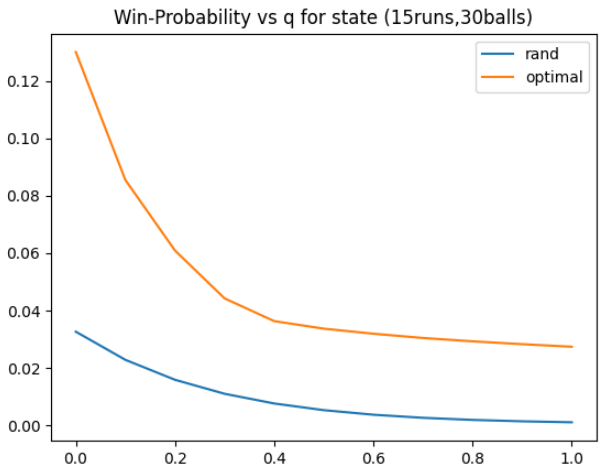
Code for A to be the batsman initially:

```
for s in range(0, len(statesA)):
    for action in actions:
        bb = statesA[s]//100
        rr = statesA[s] - bb*100
        for outcome in outcomesA:
            if outcome == -1:
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[-1], 0, R[act_dict[action]][outcome_di
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[-1], 0, R[act_dict[action]][outcome_dict[-
                continue
            elif outcome>=rr:
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[1], 1, R[act_dict[action]][outcome_dic
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[1], 1, R[act_dict[action]][outcome_dict[ou
                continue
            if bb == 1 and rr > outcome:
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[-1], 0, R[act_dict[action]][outcome_di
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[-1], 0, R[act_dict[action]][outcome_dict[o
                continue

            if (bb - 1)%6 == 0 and outcome%2 == 1:
                next_state = get_state(bb-1, rr-outcome)
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][ou
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][outcom
                continue

            elif (bb - 1)%6 == 0 and outcome%2 == 0:
                next_state = get_state(bb-1, rr-outcome) + offset
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][ou
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][outcom
                continue


            if outcome%2 == 1:
                next_state = get_state(bb-1, rr-outcome) + offset
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][ou
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][outcom
                continue
            elif outcome%2 == 0:
                next_state = get_state(bb-1, rr-outcome)
                f.write("transition %d %d %d %f %f\n" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][ou
                print("transition %d %d %d %f %f" %(states[statesA[s]], act_dict[action], states[next_state], 0, R[act_dict[action]][outcom
                continue
```

Note: Encoding of states of player B is done by adding an offset to the corresponding states of A

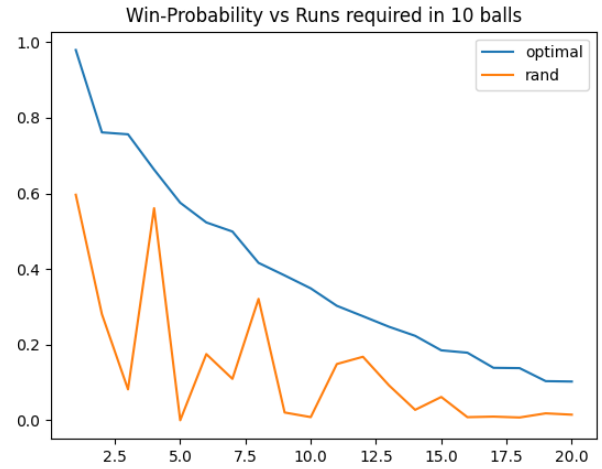
Observations:

- Trend of winning probability with B's weakness



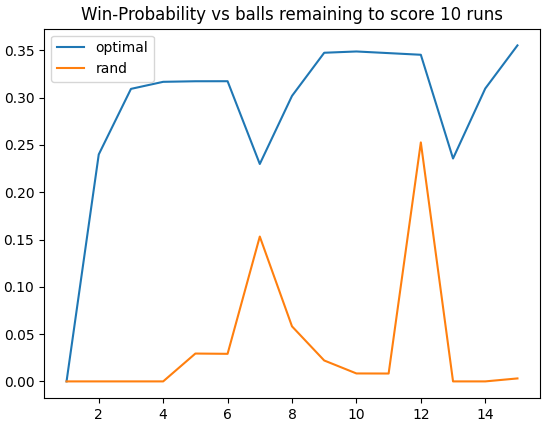
It can be seen as the weakness of B is increased for the same starting state (*bbrr*), the winning probability decreases as should be the case for both random and the optimal policy

- Trend of winning probability with number of runs left for same number of balls



For the optimal policy it is clear that as the initial number of runs increases to be scored for 10 balls, the winning probability decreases. Somewhat same trend is also observed for random policy with random jumps in between. This can be understood in the same way as no matter the policy, the winning probability should decrease intuitively as the number of runs increases to be scored for initial 10 balls

- Trend of winning probability with numbers of balls left for same number of runs to be scored



It can be seen that as the over changes there is dip in the probability to win as the batsman switches as the risk increases when the batsman switches. For random policy, no trend can be seen.