```
import pandas as pd
import numpy as np
from enum import Enum
import matplotlib.pyplot as plt
```

```
class Constant_DP(Enum):
  ACTION = [i for i in range(21)]
  T = 10
  D = [i for i in range(11)]
  INITIAL_STATES = [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
def cost_function(s_t, a_t, D_t, o_t, h_t, b_t):
  s_t_plus1 = s_t + a_t - D_t
  return o_t*a_t + max(h_t*(s_t_plus1), -b_t*(s_t_plus1))
```

```
def plot_function(y, ylabel, title):
  # x = [i for i in range(len(y))]
  plt.plot(Constant_DP.INITIAL_STATES.value, y)
  plt.xlabel("Initial States (s_0)")
  plt.ylabel(f"{ylabel}")
  plt.title(f"{title}")
  plt.show()
```

```
print(Constant_DP.INITIAL_STATES.value)
```

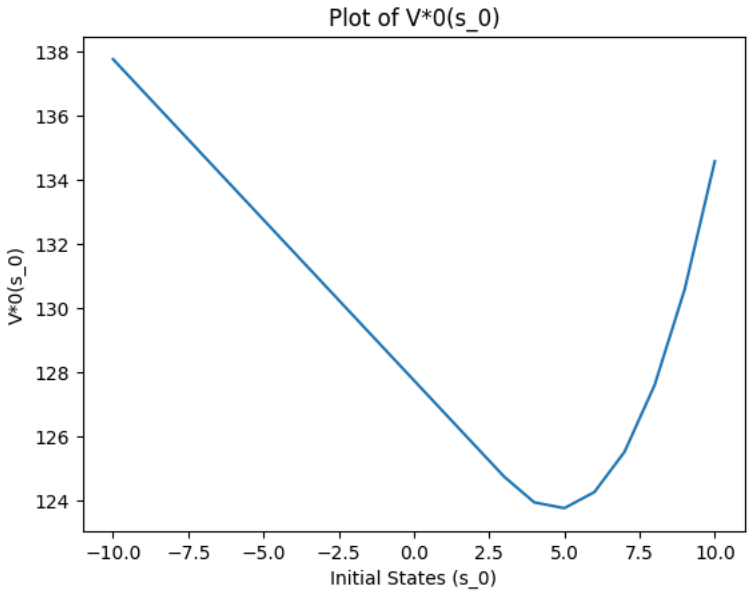
```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

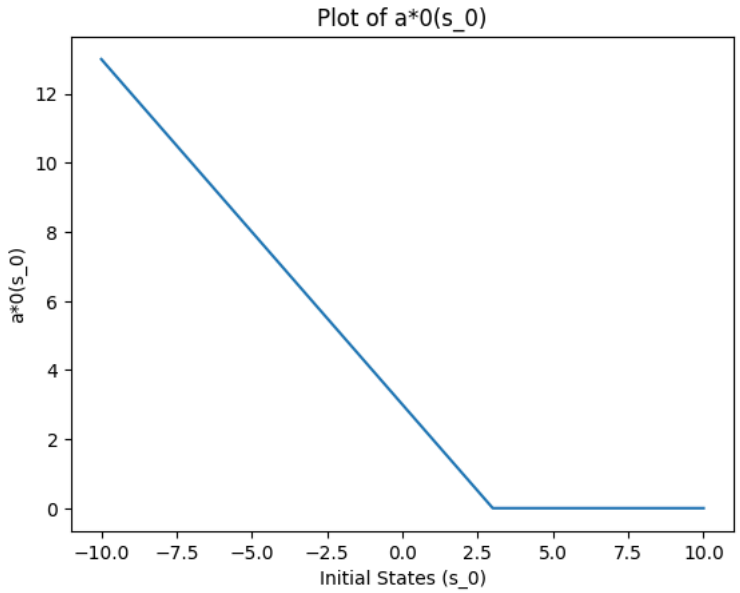
```
def compute_possible_states():
  max_demand = np.max(Constant_DP.D.value) # this is min s_t+1
  max_action = np.max(Constant_DP.ACTION.value) # this is max s_t+1
  horizon = 10 # number of time steps
  min_initial_state = -10
  max_initial_state = 10

  return [value for value in range(min_initial_state - horizon*max_demand, horizon*max_action + max_initial_state + 1)]
```

```
def dp_implementation(order, holding, backlog):
  """
  """
  all_possible_states = compute_possible_states()
  value_cost_matrix = [[0 for _ in range(len(all_possible_states))] for t in range(11)] # rows corresponds to time step and columr
  value_cost_matrix[-1] = np.array([max(holding*state, -backlog*state) for state in all_possible_states])
  uniform_probability = 1/len(Constant_DP.D.value)
  policy_matrix = [[0 for _ in range(len(all_possible_states))] for t in range(11)] # final step you take no action, so policy of

  state_to_index_map = {state:idx for idx, state in enumerate(all_possible_states)}

  # backwards iteration
  for time_step in range(10, -1, -1):

    if time_step == 10:
      continue

    for idx, state in enumerate(all_possible_states):
      expected_min_cost = float('inf')
      optimal_action = 0

      for action in Constant_DP.ACTION.value:
        expectation = 0

        for demand in Constant_DP.D.value:

          value_row = value_cost_matrix[time_step+1]
          next_state = state + action - demand

          # makes sure the next state is in range of all possible valid states
          if next_state < all_possible_states[0]:
            next_state_idx = 0

          elif next_state > all_possible_states[-1]:
            next_state_idx = len(all_possible_states) - 1
```

```
            else:
              next_state_idx = state_to_index_map[next_state]
            next_value_cost = value_cost_matrix[time_step+1][next_state_idx]

            # finding the expected cost with the probability of getting the demand
            expectation += uniform_probability*(cost_function(state, action, demand, order, holding, backlog) + next_value_cost)

          if expectation < expected_min_cost:
            optimal_action = action
            expected_min_cost = expectation

          policy_matrix[time_step][idx] = optimal_action # keeps track of the optimal action
          value_cost_matrix[time_step][idx] = expected_min_cost # keeps track of optimal cost

    initial_state_indices = [state_to_index_map[initial_state] for initial_state in Constant_DP.INITIAL_STATES.value]
    value_cost_initial = [value_cost_matrix[0][initial_state_idx] for initial_state_idx in initial_state_indices]
    policy_initial = [policy_matrix[0][initial_state_idx] for initial_state_idx in initial_state_indices]

    plot_function(value_cost_initial, "V*0(s_0)", "Plot of V*0(s_0)")
    plot_function(policy_initial, "a*0(s_0)", "Plot of a*0(s_0)")

    # s*0
    optimal_s0_idx = np.argmin(value_cost_initial)
    optimal_s0 = Constant_DP.INITIAL_STATES.value[optimal_s0_idx]

    # base stock
    base_stock_idx = np.argmin(policy_initial)
    base_stock = Constant_DP.INITIAL_STATES.value[base_stock_idx]

    print("Base stock optimal s_t:", base_stock)
    print("The value s0 that minimizes V*0 is: ", optimal_s0)

    return value_cost_initial, policy_initial
```

```
o_t = 1
h_t = 4
b_t = 2
optimal_value_cost, _ = dp_implementation(o_t, h_t, b_t)
```





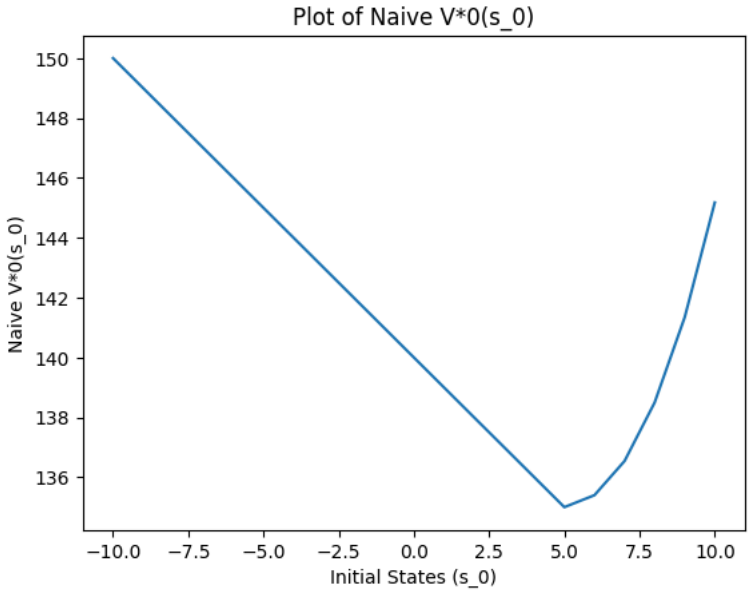

```
Base stock optimal s_t: 3
The value s0 that minimizes V*0 is:  5
```

```
def find_naive_action(state):
  s_bar = 5
  action = max(0, s_bar-state)

  # returns only valid actions between 0 and 20
  return min(action, 20)
```

```
def dp_naive_implementation(order, holding, backlog):
  """
  """
  all_possible_states = compute_possible_states()
  value_cost_matrix = [[0 for _ in range(len(all_possible_states))] for t in range(11)] # rows corresponds to time step and columns
  value_cost_matrix[-1] = np.array([max(holding*state, -backlog*state) for state in all_possible_states])
  uniform_probability = 1/len(Constant_DP.D.value)

  state_to_index_map = {state:idx for idx, state in enumerate(all_possible_states)}

  # backwards iteration
  for time_step in range(10, -1, -1):

    if time_step == 10:
      continue

    for idx, state in enumerate(all_possible_states):
```

```
      action = find_naive_action(state)
      naive_expectation = 0

      for demand in Constant_DP.D.value:

        value_row = value_cost_matrix[time_step+1]
        next_state = state + action - demand

        # makes sure the next state is in range of all possible valid states
        if next_state < all_possible_states[0]:
          next_state_idx = 0

        elif next_state > all_possible_states[-1]:
          next_state_idx = len(all_possible_states) - 1

        else:
          # next_state_idx = np.argwhere(all_possible_states == next_state)
          next_state_idx = state_to_index_map[next_state]
        next_value_cost = value_cost_matrix[time_step+1][next_state_idx]

        # finding the expected cost with uniform probability of getting a specific demand
        naive_expectation += uniform_probability*(cost_function(state, action, demand, order, holding, backlog) + next_value_cost)

      value_cost_matrix[time_step][idx] = naive_expectation # keeps track of optimal cost

  initial_state_indices = [state_to_index_map[initial_state] for initial_state in Constant_DP.INITIAL_STATES.value]
  naive_value_cost_initial = [value_cost_matrix[0][initial_state_idx] for initial_state_idx in initial_state_indices]

  plot_function(naive_value_cost_initial, "Naive V*0(s_0)", "Plot of Naive V*0(s_0)")

  # s*0
  optimal_s0_idx = np.argmin(naive_value_cost_initial)
  optimal_s0 = Constant_DP.INITIAL_STATES.value[optimal_s0_idx]
  print("The value s0 that minimizes V*0 is: ", optimal_s0)

  return naive_value_cost_initial, []
```

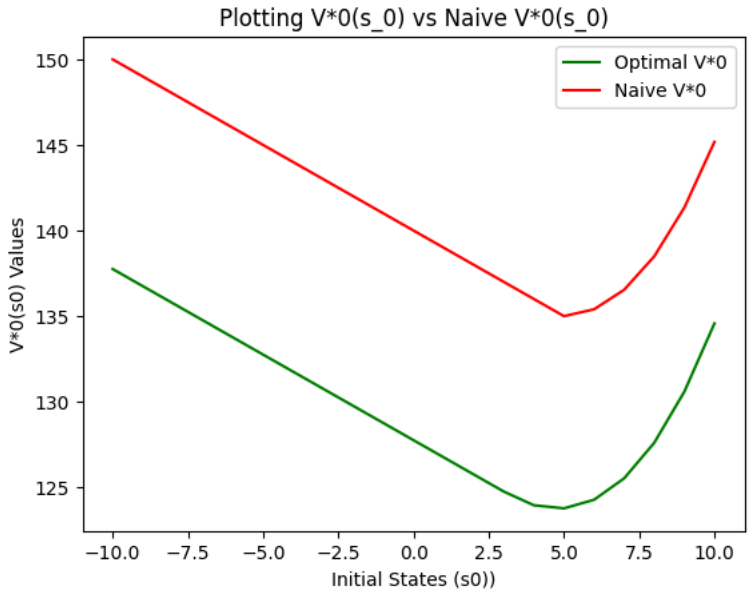
```
naive_value_cost, _ = dp_naive_implementation(o_t, h_t, b_t)
```



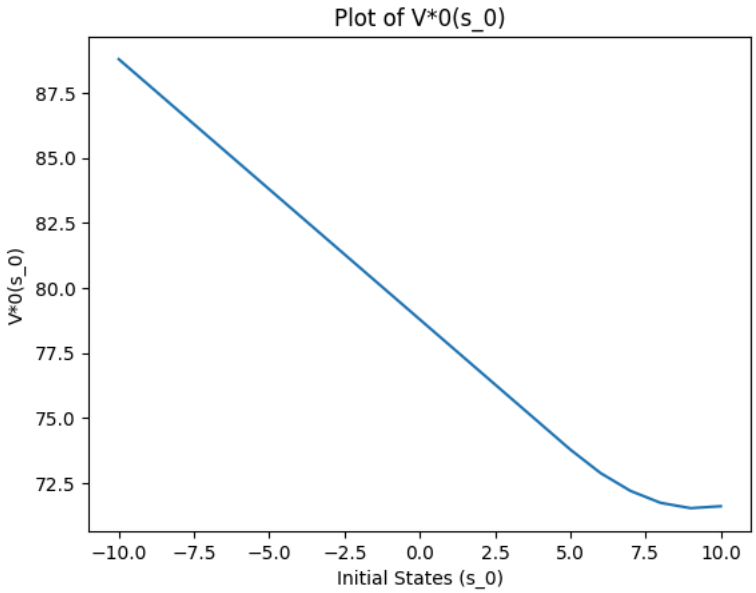

```
The value s0 that minimizes V*0 is:  5
```

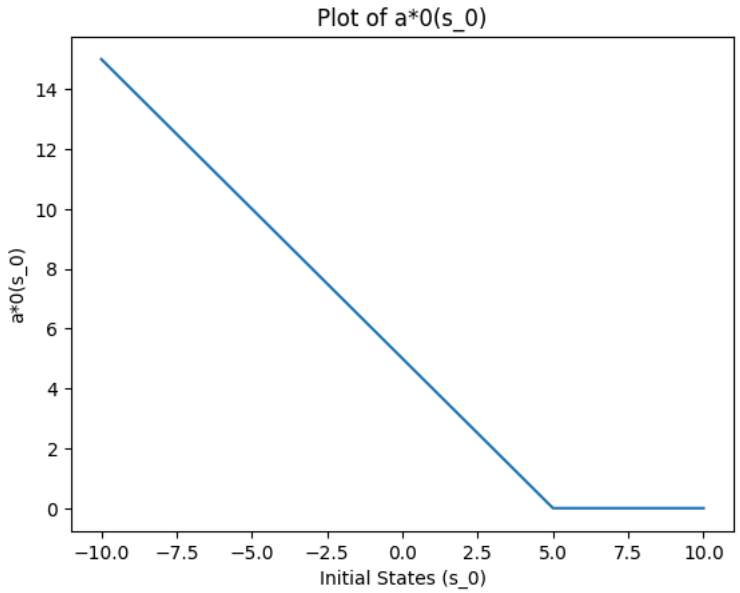
```
value_cost_comparison_array = [optimal_value_cost, naive_value_cost]
colors = ["green", "red"]
labels = ["Optimal V*0", "Naive V*0"]

for idx, val_cost in enumerate(value_cost_comparison_array):
  color = colors[idx]
  plt.plot(Constant_DP.INITIAL_STATES.value, val_cost, color, label=labels[idx])
  plt.title("Plotting V*0(s_0) vs Naive V*0(s_0)")
  plt.xlabel("Initial States (s0))")
  plt.ylabel("V*0(s0) Values")
```

```
    plt.legend()
plt.show()
```



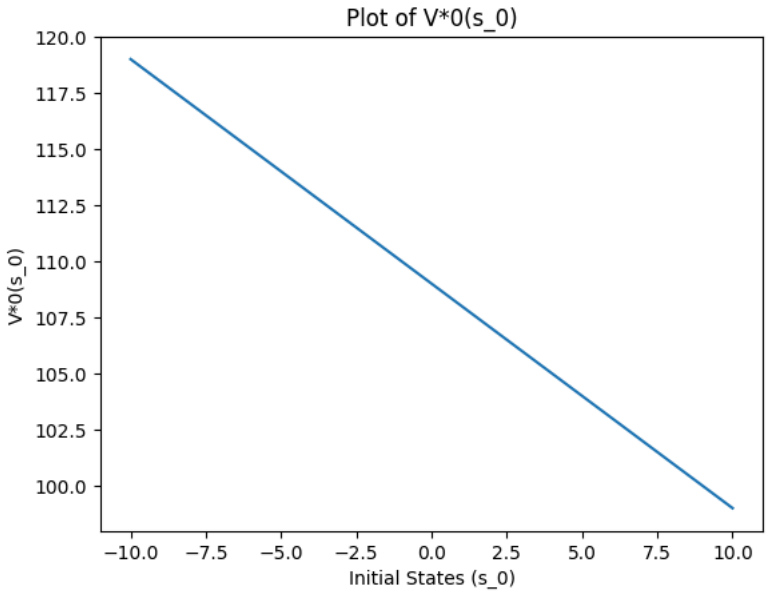

```
# baseline
o_base = 1
h_base = 1
b_base = 1
base_value_cost, base_policy = dp_implementation(o_base, h_base, b_base)
```

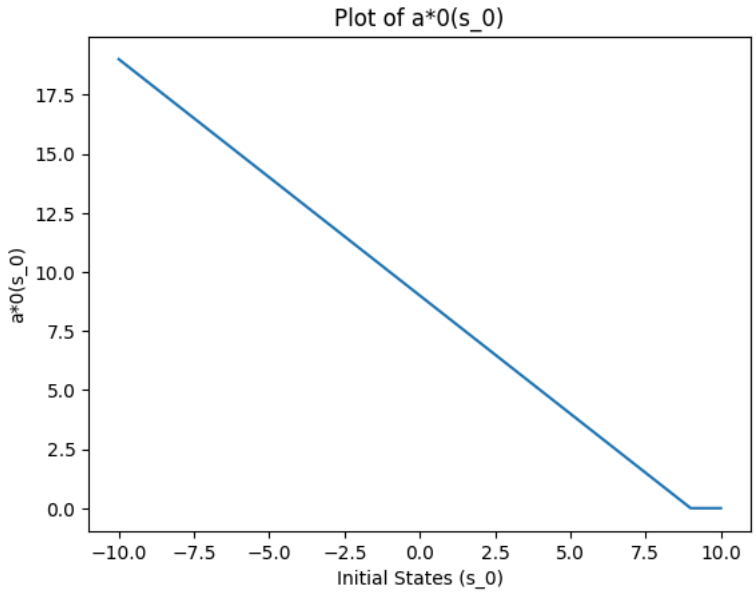





```
Base stock optimal s_t: 5
The value s0 that minimizes V*0 is:  9
```

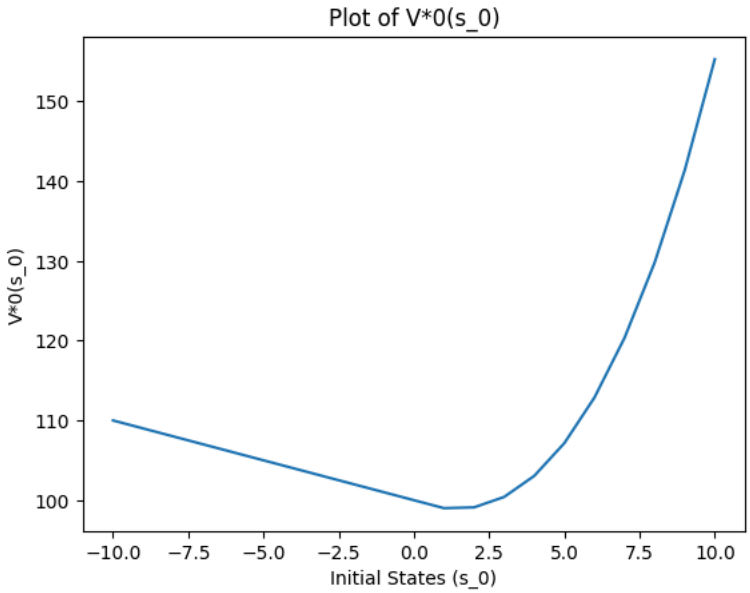
```
# high backlog
b_high = 10
high_b_value_cost, high_b_policy = dp_implementation(o_base, h_base, b_high)
```

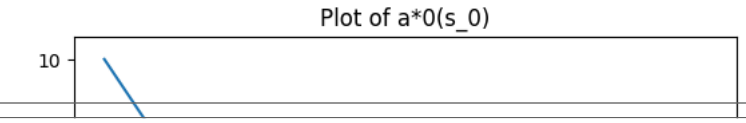





```
Base stock optimal s_t: 9
The value s0 that minimizes V*0 is:  10
```

```
# high holding
h_high = 10
high_h_value_cost, high_h_policy = dp_implementation(o_base, h_high, b_base)
```






```
# high order
o_high = 11
high_o_value_cost, high_o_policy = dp_implementation(o_high, h_base, b_base)
```