Part 1

Value Iteration

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

According to the value iteration algorithm as shown in the book, I first define a reward matrix while mark the starting position and the wall position explicitly. I then represent the four different moves using integer and store the effect of each action in two arrays (action_row and action_col). I also keep the current, the updated utilities and optimal policy in three matrixes. The algorithm is looped until converge but given that the agent sequence is infinite, we have to set the threshold as 0 at this stage.

Inside the loop we check if the position agent locates is a wall, if yes then continue the loop. We capture the max utility by trying all the possible move and find the best one, update the optimal policy accordingly. New utilities are updated based on the reward and the maximum utility times discount factor. And set the current utilities matrix as the updated version.

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s') \ ,$$

Print some key information in every loop like the current policy and the current utilities. A minor issue is that an agent might hit the boundary, the way we deal with this situation is check and let the agent stay at the same position when hitting occurs. The way we compute expected utility is through probability where intent direction has 0.8 and left and right of the intent direction have 0.1.

Because the infinite sequence we must stop the program manually in the middle and capture the utilities value and optimal policy of each position. See the following screenshot.

```
Utilities after 5663 iterations:
(100.000000) (0.000000) (95.045901) (93.875449) (92.655057) (93.328905)
(98.393362) (95.883017) (94.545487) (94.398241) (0.000000) (90.918320)
(96.948500) (95.586428) (93.298920) (93.177235) (93.103379) (91.795808)
(95.553839) (94.452494) (93.277434) (91.121281) (91.815901) (91.889072)
(94.312519) (0.000000) (0.000000) (0.000000) (89.549839) (90.567790)
(92.937474) (91.728778) (90.535152) (89.356409) (88.570331) (89.298727)
Optimal policy:
 ^  |  <  <  <  ^
 ^  <  <  <  |  ^
 ^  <  <  ^  <  <
 ^  <  S  <  ^  ^
 ^  |  |  |  ^  ^
 ^  <  <  <  ^  ^

Utilities after 5664 iterations:
(100.000000) (0.000000) (95.045901) (93.875449) (92.655057) (93.328905)
(98.393362) (95.883017) (94.545487) (94.398241) (0.000000) (90.918320)
(96.948500) (95.586428) (93.298920) (93.177235) (93.103379) (91.795808)
(95.553839) (94.452494) (93.277434) (91.121281) (91.815901) (91.889072)
(94.312519) (0.000000) (0.000000) (0.000000) (89.549839) (90.567790)
(92.937474) (91.728778) (90.535152) (89.356409) (88.570331) (89.298727)
Optimal policy:
 ^  |  <  <  <  ^
 ^  <  <  <  |  ^
 ^  <  <  ^  <  <
 ^  <  S  <  ^  ^
 ^  |  |  |  ^  ^
 ^  <  <  <  ^  ^
```

(Plot of optimal policy and utilities of all states)

Policy iteration

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

The algorithm is very similar to the value iteration and these two programs are somewhat integrated together. The difference is that in policy iteration the program loops until policy unchanged while in value iteration it loops until the utilities difference between two continuous iterations is small enough, but again here sequence is infinite.

Another important thing worth mentioning is that, Policy Iteration refines an initial policy by alternating between policy evaluation and policy improvement, whereas Value Iteration iteratively updates the value function and derives the optimal policy from it directly.

```
Utilities in iteration 5224:
(100.00) (0.00) (95.05) (93.88) (92.66) (93.33)
(98.39) (95.88) (94.55) (94.40) (0.00) (90.92)
(96.95) (95.59) (93.30) (93.18) (93.10) (91.80)
(95.55) (94.45) (93.28) (91.12) (91.82) (91.89)
(94.31) (0.00) (0.00) (0.00) (89.55) (90.57)
(92.94) (91.73) (90.54) (89.36) (88.57) (89.30)
Current policy:
 ^  |  <  <  <  ^
 ^  <  <  <  |  ^
 ^  <  <  ^  <  <
 ^  <  S  <  ^  ^
 ^  |  |  |  ^  ^
 ^  <  <  <  ^  ^

Utilities in iteration 5225:
(100.00) (0.00) (95.05) (93.88) (92.66) (93.33)
(98.39) (95.88) (94.55) (94.40) (0.00) (90.92)
(96.95) (95.59) (93.30) (93.18) (93.10) (91.80)
(95.55) (94.45) (93.28) (91.12) (91.82) (91.89)
(94.31) (0.00) (0.00) (0.00) (89.55) (90.57)
(92.94) (91.73) (90.54) (89.36) (88.57) (89.30)
Current policy:
 ^  |  <  <  <  ^
 ^  <  <  <  |  ^
 ^  <  <  ^  <  <
 ^  <  S  <  ^  ^
 ^  |  |  |  ^  ^
 ^  <  <  <  ^  ^
```

(Plot of optimal policy and utilities of all states)

Part 2

```
// initialize rewards array 10*10
private static final double[][] rewards = {
        { 1, 0, 1, -0.04, -0.04, 1, 0, -0.04, -0.04, -0.04},
        { -0.04, -1, -0.04, 1, 0, -1, -0.04, 1, 0, 1 },
        { -0.04, -0.04, -1, -0.04, 1, -0.04, -0.04, -0.04, 1, -0.04 },
        { -0.04, -0.04, 0, -1, -0.04, 1, 0, 0, 0 , 1 },
        { -0.04, 0, 0, 0, -1, -0.04, -1, 1, -1, -0.04 },
        { -0.04, -0.04, -0.04, -0.04, -0.04, -0.04, -0.04, -0.04, 1, -0.04 },
        { -0.04, 1, -0.04, -0.04, -0.04, -0.04, -0.04, -0.04, -0.04, -0.04 },
        {-0.04, 0, 0, 0, -1, -0.04, 1, 1, 1, -0.04 },
        {-0.04, -0.04, -1, -1, -0.04, -1, 0, 0, 0 , -1 },
        { -0.04, 1, -1, -0.04, 1, -0.04, -0.04, -0.04, 1, -0.04 },
        };
```

```
private static boolean isWall(int row, int col) {
    if ((row == 0 && col == 1) || (row == 1 && col == 4) || (row == 4 && col == 1) || (row == 4 && col == 2)
            || (row == 4 && col == 3) || (row == 0 && col == 6) || (row == 1 && col ==8) ||
            (row==3 && col==8) || (row==3 && col==7) || (row==3 && col==6) || (row==7 && col==1) ||
            (row==7 && col==2) || (row==7 && col==3) || (row==8 && col==6) || (row==8 && col==7) ||
            (row==8 && col==8))
        return true;
    else
        return false;
}
```

The more complexed 10*10 maze if defined as above where we still use the start point as (2,3). Please note that all the other positions with rewards 0 are wall. Run the algorithm developed in Part 1 and we can get the following result. "^" represents up, "<" represents left, ">" represents right and "_" represents down.

```
Utilities after 6895 iterations:
(100.000000) (0.000000) (95.045901) (93.875454) (92.655062) (93.328910) (0.000000) (89.589491) (89.479925) (90.660227)
(98.393362) (95.883017) (94.545487) (94.398289) (0.000000) (90.784912) (89.704135) (90.784873) (0.000000) (92.002970)
(96.948500) (95.586428) (93.298920) (93.177712) (93.107813) (91.818815) (90.522987) (90.803612) (92.002970) (91.901997)
(95.553839) (94.452494) (93.277434) (91.124606) (91.852445) (92.219403) (0.000000) (0.000000) (0.000000) (93.100260)
(94.312519) (0.000000) (0.000000) (0.000000) (89.925541) (91.834071) (92.263841) (94.423527) (93.234596) (93.013142)
(92.938869) (91.741469) (90.556647) (90.832692) (91.914795) (93.228942) (94.406943) (94.771694) (95.553492) (94.210373)
(91.741469) (91.833129) (90.629241) (91.754929) (93.079236) (94.473599) (95.751111) (95.966681) (95.762963) (94.582898)
(90.547685) (0.000000) (0.000000) (0.000000) (92.894753) (95.444984) (97.143338) (97.281297) (97.144640) (95.472772)
(89.232655) (88.129849) (87.236682) (89.512005) (91.792743) (92.874482) (0.000000) (0.000000) (0.000000) (93.035456)
(88.129849) (88.613447) (87.946889) (90.408732) (91.712852) (91.540761) (90.349480) (90.743611) (91.939869) (91.838100)
Optimal policy:
 ^  |  <  <  <  ^  |  _  >  _
 ^  <  <  <  |  ^  <  _  |  _
 ^  <  <  <  ^  <  <  <  >  _
 ^  <  S  ^  ^  >  |  |  |  _
 ^  |  |  |  >  _  >  _  _  _
 ^  <  <  >  >  _  _  _  _  <
 ^  <  <  >  >  >  _  _  _  <
 ^  |  |  |  >  >  >  _  <  <
 ^  <  >  >  >  ^  |  |  |  ^
 ^  _  >  >  ^  ^  <  >  >  ^

Utilities after 6896 iterations:
(100.000000) (0.000000) (95.045901) (93.875454) (92.655062) (93.328910) (0.000000) (89.589491) (89.479925) (90.660227)
(98.393362) (95.883017) (94.545487) (94.398289) (0.000000) (90.784912) (89.704135) (90.784873) (0.000000) (92.002970)
(96.948500) (95.586428) (93.298920) (93.177712) (93.107813) (91.818815) (90.522987) (90.803612) (92.002970) (91.901997)
(95.553839) (94.452494) (93.277434) (91.124606) (91.852445) (92.219403) (0.000000) (0.000000) (0.000000) (93.100260)
(94.312519) (0.000000) (0.000000) (0.000000) (89.925541) (91.834071) (92.263841) (94.423527) (93.234596) (93.013142)
(92.938869) (91.741469) (90.556647) (90.832692) (91.914795) (93.228942) (94.406943) (94.771694) (95.553492) (94.210373)
(91.741469) (91.833129) (90.629241) (91.754929) (93.079236) (94.473599) (95.751111) (95.966681) (95.762963) (94.582898)
(90.547685) (0.000000) (0.000000) (0.000000) (92.894753) (95.444984) (97.143338) (97.281297) (97.144640) (95.472772)
(89.232655) (88.129849) (87.236682) (89.512005) (91.792743) (92.874482) (0.000000) (0.000000) (0.000000) (93.035456)
(88.129849) (88.613447) (87.946889) (90.408732) (91.712852) (91.540761) (90.349480) (90.743611) (91.939869) (91.838100)
Optimal policy:
 ^  |  <  <  <  ^  |  _  >  _
 ^  <  <  <  |  ^  <  _  |  _
 ^  <  <  <  ^  <  <  <  >  _
 ^  <  S  ^  ^  >  |  |  |  _
 ^  |  |  |  >  _  >  _  _  _
 ^  <  <  >  >  _  _  _  _  <
 ^  <  <  >  >  >  _  _  _  <
 ^  |  |  |  >  >  >  _  <  <
 ^  <  >  >  >  ^  |  |  |  ^
 ^  _  >  >  ^  ^  <  >  >  ^
```

The complexity of the environment and the number of states both affect the convergence time of the algorithms. In general, as the size of the environment increases, the time it takes for the algorithms to converge also increases. This is because the algorithms need to explore more states and perform more calculations to find the optimal policy. The complexity of the environment also plays a role in the convergence time. For example, if there are many walls, the algorithms may need to explore more alternative paths to find the optimal policy, which can increase convergence time.

How complex can you make the environment and still be able to learn the right policy?
The complexity of the environment that can be handled by these algorithms mainly depends on the available computational resources and time. With more powerful hardware and more time, it is possible to learn the right policy for even more complex environments. However, for very large and complex environments, other approaches, such as reinforcement learning with function approximation (e.g., deep Q-learning), might be more suitable due to their ability to generalize across states.

We can also focus on the problem representation and algorithm efficiency: The representation of the environment and its states can also impact the complexity. For example, using state abstraction or state aggregation techniques can simplify the problem, allowing the algorithms to tackle more complex environments. The efficiency of the implementation of policy/value iteration matters. Optimized algorithms can handle more complex environments within the same resource constraints.