773. Sliding Puzzle

On an `2 x 3` board, there are five tiles labeled from `1` to `5`, and an empty square represented by `0`. A **move** consists of choosing `0` and a 4-directionally adjacent number and swapping it.

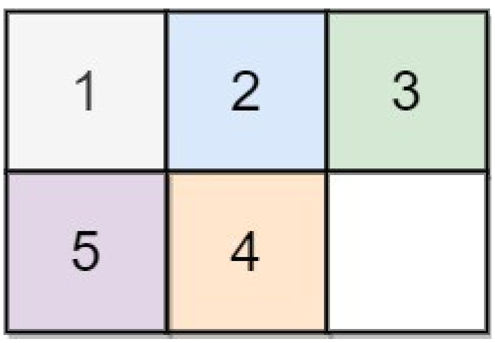
The state of the board is solved if and only if the board is `[[1,2,3],[4,5,0]]`.

Given the puzzle board `board`, return *the least number of moves required so that the state of the board is solved*. If it is impossible for the state of the board to be solved, return `-1`.

Example 1:

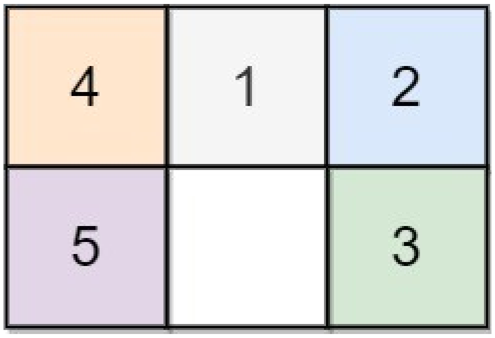

```
Input: board = [[1,2,3],[4,0,5]]
Output: 1
Explanation: Swap the 0 and the 5 in one move.
```

Example 2:




```
Input: board = [[1,2,3],[5,4,0]]
Output: -1
Explanation: No number of moves will make the board solved.
```

Example 3:



```
Input: board = [[4,1,2],[5,0,3]]
Output: 5
Explanation: 5 is the smallest number of
moves that solves the board.
An example path:
After move 0: [[4,1,2],[5,0,3]]
After move 1: [[4,1,2],[0,5,3]]
After move 2: [[0,1,2],[4,5,3]]
After move 3: [[1,0,2],[4,5,3]]
After move 4: [[1,2,0],[4,5,3]]
After move 5: [[1,2,3],[4,5,0]]
```

Constraints:

- `board.length == 2`
- `board[i].length == 3`
- `0 <= board[i][j] <= 5`
- Each value `board[i][j]` is **unique**

773. Sliding Puzzle

```
/*
    DFS
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution{
    private:
        int m; // Row
        int n; // Cols
        vvi directions={
            {-1,0}, // Up
            {1,0}, // Down
            {0,-1}, // Left
            {0,1} // Right;
        };
        vvi target={{1,2,3},{4,5,0}};
        std::map<vvi,int> visited;
        class Puzzle{
            public:
                vvi board;

                // Position of empty tile in the cureent state
                int x_empty_tile;
                int y_empty_tile;

                int steps; // Number of steps to reach the current state
            public:
                Puzzle(vvi board,int x_empty_tile,int y_empty_tile,int steps):
                    board(board),
                    x_empty_tile(x_empty_tile),
                    y_empty_tile(y_empty_tile),
                    steps(steps) {}
        };
    public:
        // Find the position of the empty tile (0)
        std::pair<int,int> find_zero_pos(vvi& board){
            for(int i=0;i<m;++i){
                for(int j=0;j<n;++j){
                    if(board[i][j]==0) return {i,j};
                }
            }
            return {-1,-1}; // Never reached
        }
```

```
    void dfs(Puzzle* puzzle){
        if(visited.find(puzzle->board)!=visited.end() && visited[puzzle->board]<=puzzle->steps)
return;
        visited[puzzle->board]=puzzle->steps;

        for (auto& dir: directions){
            int x=puzzle->x_empty_tile + dir[0];
            int y=puzzle->y_empty_tile + dir[1];

            if ( x < 0 || x >= m || y < 0 || y >= n) continue;

            vvi next_board = puzzle->board;
            int tmp = next_board[x][y];
            next_board[x][y] = 0;
            next_board[puzzle->x_empty_tile][puzzle->y_empty_tile] = tmp;


            Puzzle* next_puzzle = new Puzzle(next_board, x, y,puzzle->steps+1);

            dfs(next_puzzle);
        }
    }

    int slidingPuzzle(vvi& board){
        m=board.size();
        n=board[0].size();

        // Find position of empty tile
        std::pair<int,int> zero_pos=find_zero_pos(board);

        // Create the puzzle
        Puzzle* puzzle=new Puzzle(board,zero_pos.first,zero_pos.second,0);

        dfs(puzzle);

        return visited.find(target)!=visited.end()?visited[target]:-1;
    }
};
```

773. Sliding Puzzle

```
/*
  BFS
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution{
  private:
    int m; // Row
    int n; // Cols

    vvi directions={
      {-1,0}, // Up
      {1,0}, // Down
      {0,-1}, // Left
      {0,1} // Right;
    };
    vvi target={{1,2,3},{4,5,0}};


    class Puzzle{
      public:
        vvi board;

        // Position of empty tile in the cureent state
        int x_empty_tile;
        int y_empty_tile;

        int steps; // Number of steps to reach the current state
      public:
        Puzzle(vvi board,int x_empty_tile,int y_empty_tile,int steps):
          board(board),
          x_empty_tile(x_empty_tile),
          y_empty_tile(y_empty_tile),
          steps(steps) {}
    };
    public:
    // Find the position of the empty tile (0)
    std::pair<int,int> find_zero_pos(vvi& board){
      for(int i=0;i<m;++i){
        for(int j=0;j<n;++j){
          if(board[i][j]==0) return {i,j};
        }
      }
      return {-1,-1}; // Never reached
    }
```

```
// Perfom BFS
int bfs(Puzzle* puzzle){
    std::queue<Puzzle*>q;
    q.push(puzzle);

    std::map<vvi, bool> visited;

    while(!q.empty()){
        Puzzle* current_puzzle=q.front();
        q.pop();

        vvi current_board = current_puzzle->board;

        if(current_board==target) return current_puzzle->steps;

        // State visited, no need to fo it again
        if (visited[current_board]) continue;
        visited[current_board] = true;

        // Move empty tile in four directions
        for (auto& dir: directions){
            int x=current_puzzle->x_empty_tile + dir[0];
            int y=current_puzzle->y_empty_tile + dir[1];

            if ( x < 0 || x >= m || y < 0 || y >= n) continue;

            vvi next_board = current_board;
            int tmp = next_board[x][y];
            next_board[x][y] = 0;
            next_board[current_puzzle->x_empty_tile][current_puzzle->y_empty_tile] = tmp;

            // If the next state is not visited,
            if (!visited[next_board]){

                // Create next puzzle
                Puzzle* next_puzzle = new Puzzle(next_board, x, y,current_puzzle->steps+1);

                // Add it to the queue
                q.push(next_puzzle);
            }
        }
    }
    return -1; // If target not reached at all
}
```

```
    int slidingPuzzle(vvi& board){
        m=board.size();
        n=board[0].size();

        // Find position of empty tile
        std::pair<int,int> zero_pos=find_zero_pos(board);

        // Create the puzzle
        Puzzle* puzzle=new Puzzle(board,zero_pos.first,zero_pos.second,0);

        return bfs(puzzle);
    }
};
```

773. Sliding Puzzle

```
/*
  A*
*/
typedef std::vector<int> vi;
typedef std::vector<vi> vvi;
class Solution{
  private:
    int m; // Row
    int n; // Cols

    vvi directions={
      {-1,0}, // Up
      {1,0}, // Down
      {0,-1}, // Left
      {0,1} // Right;
    };

    vvi target={{1,2,3},{4,5,0}};

    class Puzzle{
      public:
        vvi board;

        // Position of empty tile in the cureent state
        int x_empty_tile;
        int y_empty_tile;

        int steps; // Number of steps to reach the current state
        int h; // Heuristic value: number of misplaced tiles
        int coast; // The coast to get target state from an actual state.
      public:
        Puzzle(vvi board,int x_empty_tile,int y_empty_tile,int steps,int h, int coast):
          board(board),
          x_empty_tile(x_empty_tile),
          y_empty_tile(y_empty_tile),
          steps(steps),
          h(h),
          coast(coast) {}
    };
```

public:
```
// Find the position of the empty tile (0)
std::pair<int,int> find_zero_pos(vvi& board){
    for(int i=0;i<m;++i){
        for(int j=0;j<n;++j){
            if(board[i][j]==0) return {i,j};
        }
    }
    return {-1,-1}; // Never reached
}
```

```
// Heuristic function
// #misplaced tiles in actual state by comparing with the target state.
int heuristic(vvi& board){
    int h=0;
    int tile=1;
    for (int i=0;i<m;++i){
        for (int j=0;j<n;++j){
            if ( (i!=m-1 || j!=n-1) &&board[i][j]!=tile) h++;
            tile++;
        }
    }
    return h+int(board[m-1][n-1]!=0);
}
```

```
    // A* algorithm
    int a_star(Puzzle* puzzle){
        // Use min heap to get the minimum coast
        std::priority_queue<std::pair<int,Puzzle*>,std::vector<std::pair<int,Puzzle*>>,
std::greater<std::pair<int,Puzzle*>>> min_heap;

        // Push initial state to the min heap
        min_heap.push({puzzle->coast, puzzle});

        std::map<vvi,bool> visited;

        while (!min_heap.empty()){
            // Get the puzzle with the minimum coast
            Puzzle* current_puzzle=min_heap.top().second;
            min_heap.pop();

            vvi current_board=current_puzzle->board;

            // If number of misplaced tiles is equal to 0, means we reached the target
            // then return the number of steps
            if (current_puzzle->h==0) return current_puzzle->steps;

            // if the board's state is visited, no need to continue with it
            if (visited[current_board]) continue;

            // If not visited, mark it as visited
            visited[current_board]=true;

            // Move empty tile in four directions
            for (auto& dir: directions){
                int x=current_puzzle->x_empty_tile + dir[0];
                int y=current_puzzle->y_empty_tile + dir[1];

                if ( x<0 || x>=m || y<0 || y>=n) continue;

                // Get next state
                vvi next_board=current_board;
                int tmp=next_board[x][y];
                next_board[x][y]=0;
                next_board[current_puzzle->x_empty_tile][current_puzzle->y_empty_tile]=tmp;

                // If the next state is not visited,
                if (!visited[next_board]){
                    int f=current_puzzle->steps+1; // Add one step to the current state
                    int h=heuristic(next_board); // Determine the heuristic value of the next state
                    int coast=f+h; // Compute the coast to go from current state to next state

                    // Create next puzzle
                    Puzzle* next_puzzle=new Puzzle(next_board,x,y,f,h,coast);

                    // Add it to min heap
                    min_heap.push({coast,next_puzzle});
                }

            }
        }

        return -1; // If target not reached at all
    }
```

```
    int slidingPuzzle(vvi& board){
        m=board.size();
        n=board[0].size();

        // Find position of empty tile
        std::pair<int,int> zero_pos=find_zero_pos(board);

        // Compute heuristic value
        int h0=heuristic(board);

        // Create the puzzle
        Puzzle* puzzle=new Puzzle(board,zero_pos.first,zero_pos.second,0,h0,0+h0);

        return a_star(puzzle);
    }
};
```

Time Complexity Analysis

1. **State Space Size:**
   - The problem involves solving the sliding puzzle. Each configuration of the board is a unique state.
   - For a $m \times n$ board, the total number of states is $(m \times n)!$, as all tiles can be permuted (factorial of the total number of tiles).
2. **Heuristic Calculation (`heuristic`):**
   - The `heuristic` function iterates over all tiles in the $m \times n$ grid, making its complexity $O(m \cdot n)$.
3. **Priority Queue Operations:**
   - The algorithm uses a **min-heap** to store puzzle states, and operations on the heap (insertion and extraction) have a complexity of $O(\log(S))$, where S is the number of states in the heap.
   - In the worst case, all possible states $(m \times n)!$ can be generated.
4. **State Expansion:**
   - Each state expands to at most 4 neighbors (up, down, left, right), though some moves may be invalid.
   - For each valid neighbor, the `heuristic` function is called, costing $O(m \cdot n)$.
5. **Visited Map:**
   - Checking and marking states as visited involves hashing the m×n board, which takes $O(m \cdot n)$.
6. **Overall Time Complexity:**
   - In the worst case, the algorithm could explore all possible states, and for each state, it performs $O(m \cdot n)$ operations.
   - **Time complexity**: $O((m \cdot n) \cdot (m \cdot n)!)$.

Space Complexity Analysis

1. **Priority Queue:**
    - The min-heap can store at most $(m \cdot n)!$ states in the worst case.
    - Each state requires space to store:
        - The $m \times n$ board $(O(m \cdot n))$.
        - Additional metadata (like position of the empty tile, steps, heuristic, and cost).
    - Space for the priority queue: $O((m \cdot n)! \cdot (m \cdot n))$.
2. **Visited Map:**
    - Stores $(m \cdot n)!$ states as keys. Each state (board configuration) requires $O(m \cdot n)$ space.
    - Space for visited map: $O((m \cdot n)! \cdot (m \cdot n))$.
3. **Recursive Calls and Local Variables:**
    - There are no recursive calls in this implementation, so no additional stack space is needed.
4. **Overall Space Complexity:**
    - Dominated by the storage requirements of the priority queue and the visited map.
    - **Space complexity**: $O((m \cdot n)! \cdot (m \cdot n))$.

Key Points

- **Time complexity** is exponential due to the factorial growth of the state space.
- **Space complexity** is also exponential, driven by the need to store all explored states.
- Practical performance depends heavily on the heuristic function's ability to prune the search space effectively. The better the heuristic, the fewer states are expanded.