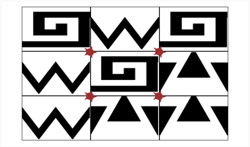
Problem A - Aztec Vaults

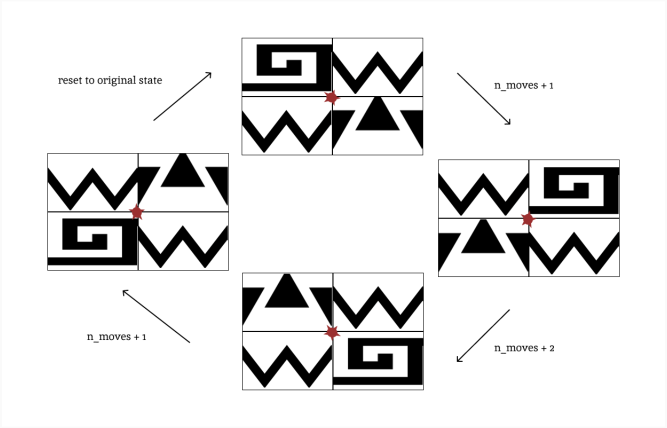


Backtracking

Backtracking

Rotate all vaults within the defined maximum number of moves.

Recurse over each vault handle and rotate three times.



Backtracking

Base case:

- Maximum number of moves is reached (invalid solution)
- Solved puzzle, update best solution so far

Backtracking

```
grid[R][C]
BEST_SOL = MAX_MOVES+1

void solve(n_moves)
    if n_moves >= BEST_SOL //exceeded maximum number of moves
        return
    if is_solved(grid)
        BEST_SOL = min(BEST_SOL,n_moves)
        return
    for(i = 0; i < R-1; i++) //iterate over each vault handle by row
        for(j = 0; j < C-1; j++)  // and column
            rotate_right(grid,i,j)
            solve(n_moves+1)
            rotate_right(grid,i,j)
            solve(n_moves+2)
            rotate_right(grid,i,j)
            solve(n_moves+1)// same as rotating left
            rotate_right(grid,i,j) //return the grid to its original state
```

Backtracking

```
void rotate_right(grid,i,j)
    a = grid[i][j]
    b = grid[i][j+1]
    c = grid[i+1][j+1]
    d = grid[i+1][j]
    grid[i][j] = d
    grid[i][j+1] = a
    grid[i+1][j+1] = b
    grid[i+1][j] = c
```

Backtracking

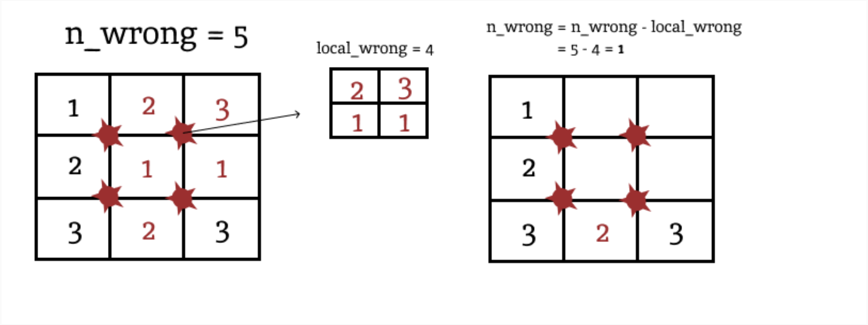
```
bool is_solved(grid)
    for(i = 0; i < R; i++)
        for(j = 0; j < C; j++)
            if grid[i][j] != i+1
                return false
    return true
```

Naive way to calculate O(R*C). There is a better way to do this.

Keep track of wrong cells

We can keep track of the total number of wrong cells in the whole grid.

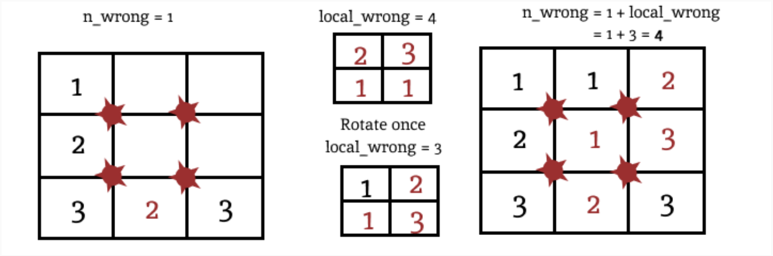
Rotations only produce local changes. O(1)



Keep track of wrong cells

We can keep track of the total number of wrong cells in the whole grid.

Rotations only produce local changes.



How can we improve?

Look for ways to reduce unnecessary computation.

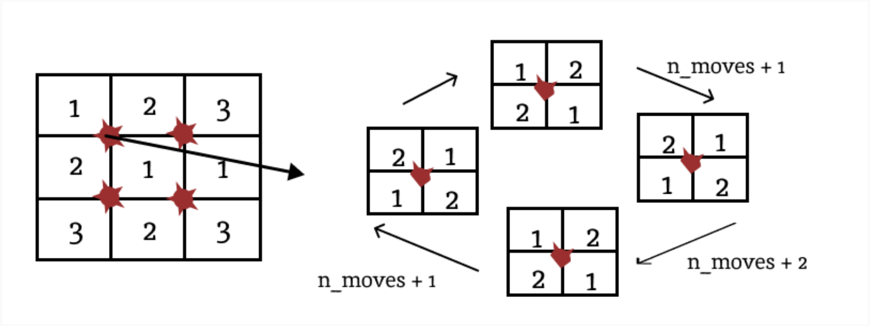
How can we improve?

Don’t process the same vault on the next recursion.

Backtracking

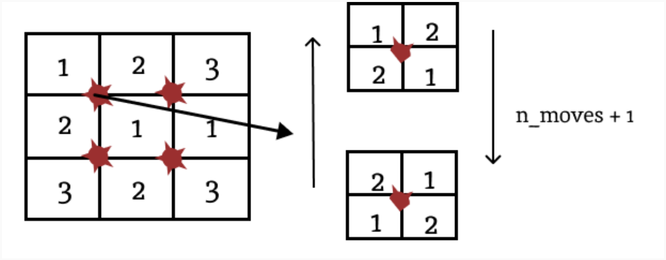
```
void solve(n_moves, previous_vault)
    if n_moves >= BEST_SOL //exceeded maximum number of moves
        return
    if is_solved(grid)
        BEST_SOL = min(BEST_SOL,n_moves)
        return
    for(i = 0; i < R-1; i++) //iterate over each vault handle by row
        for(j = 0; j < C-1; j++)  // and column
            if (i,j) == previous_vault continue
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j))
            rotate_right(grid,i,j)
            solve(n_moves+2, (i,j))
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j)) // same as rotating left
            rotate_right(grid,i,j) //return the grid to its original state
```

Diagonal Cells



Diagonal Cells

If the current handle is diagonal, we only need to **rotate once** and reset to the original state.

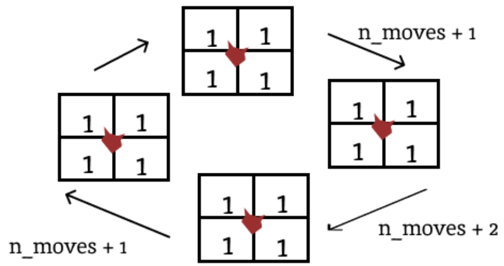


Backtracking

```
void solve(n_moves, previous_vault)
    if n_moves >= BEST_SOL //exceeded maximum number of moves
        return
    if is_solved(grid)
        BEST_SOL = min(BEST_SOL,n_moves)
        return
    for(i = 0; i < R-1; i++) //iterate over each vault handle by row
        for(j = 0; j < C-1; j++)  // and column
            if (i,j) == previous_vault continue
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j))
            rotate_right(grid,i,j)
            if is_diagonal(grid,i,j) continue
            solve(n_moves+2, (i,j))
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j)) // same as rotating left
            rotate_right(grid,i,j) //return the grid to its original state
```

Redundant Handles

If the current handle has all 4 cells with the same number, **don't rotate**.



Backtracking

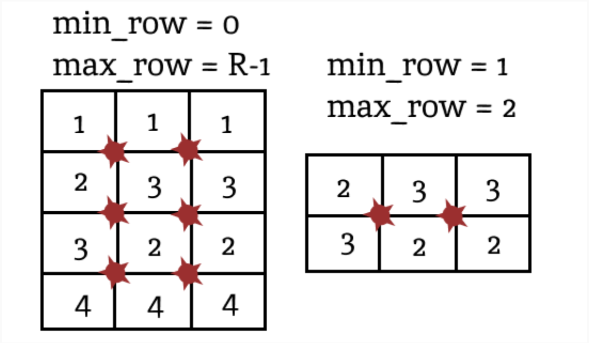
```
void solve(n_moves, previous_vault)
    if n_moves >= BEST_SOL //exceeded maximum number of moves
        return
    if is_solved(grid)
        BEST_SOL = min(BEST_SOL,n_moves)
        return
    for(i = 0; i < R-1; i++) //iterate over each vault handle by row
        for(j = 0; j < C-1; j++)  // and column
            if (i,j) == previous_vault continue
            if is_redundant_handle(grid,i,j) continue
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j))
            rotate_right(grid,i,j)
            if is_diagonal(grid,i,j) continue
            solve(n_moves+2, (i,j))
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j)) // same as rotating left
            rotate_right(grid,i,j) //return the grid to its original state
```

Solved Rows

We can freeze rows from top to bottom and from bottom to top.

Update the minimum and maximum rows **at each recursion**.

The minimum subproblem dimension should have 2 rows (or else we can't solve the puzzle).



Backtracking

```
void solve(n_moves, previous_vault, min_row, max_row)
    if n_moves >= BEST_SOL //exceeded maximum number of moves
        return
    if is_solved(grid)
        BEST_SOL = min(BEST_SOL,n_moves)
        return
    min_row,max_row = update_frozen_rows(grid,min_row,max_row)
    for(i = min_row; i <= max_row; i++) //iterate over each vault handle by row
        for(j = 0; j < C-1; j++)  // and column
            if (i,j) == previous_vault continue
            if is_redundant_handle(grid,i,j) continue
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j))
            rotate_right(grid,i,j)
            if is_diagonal(grid,i,j) continue
            solve(n_moves+2, (i,j))
            rotate_right(grid,i,j)
            solve(n_moves+1, (i,j)) // same as rotating left
            rotate_right(grid,i,j) //return the grid to its original state
```

Backtracking

```
update_frozen_rows(grid,min_row,max_row)
    for(i = min_row; i <= max_row; i++)
        if is_solved_row(i) && max_row-min_row-1 >= 2
            min_row++
        else
            break

    for(i = max_row; i > min_row; i--)
        if is_solved_row(i) && max_row-min_row-1 >= 2
            max_row--
        else
            break
    return min_row, max_row
```

Backtracking

```
bool is_solved_row(row)
```

Iterate over all the elements in the row and check if they are correct. O(C)

Can we do better?

Backtracking

```
bool is_solved_row(row)
```

Iterate over all the elements in the row and check if they are correct. O(C)

Can we do better?

```
wrongs_by_row[R]
```

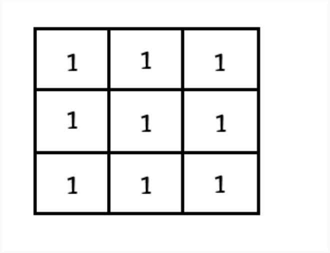
For each row, store how many wrong cells it has. Update after each rotation.

```
bool is_solved_row(row)
    return wrongs_by_row[row] == 0
```

O(1)

Invalid Inputs

There is no guarantee that the given input is solvable.



1	1	1
1	1	1
1	1	1

You should check if the puzzle contains the correct number of each distinct cell.

In a 3x3 input, you should have three 1's, three 2's and three 3's.

Preprocessing

There are ways to estimate the optimal number of minimum moves needed to solve the puzzle.

Only two cells change rows when we rotate once.

Each move, at most, decreases the total number of wrong cells by 2.

Each move, at most, decreases the distance of a cell to its target row by 1. Therefore, each move reduces the total distance of cells to target rows by 2.

Preprocessing

Count the total number of wrong cells and the total distance of cells to target rows when reading the input.

```
n_wrong = 0
total_distance = 0
for(i = 0; i < R; i++)
    for(j = 0; j < C; j++)
        read_input(grid,i,j)
        if grid[i][j] != i+1
            n_wrong++
            total_distance += abs(i+1-grid[i][j])
```

Preprocessing

Compare the estimated optimal number with the maximum number of moves given in the problem input.

```
if floor(n_wrong / 2) > MAX_MOVES or
floor(total_distance / 2 ) > MAX_MOVES
    print("the treasure is lost!\n")
```

Memorizing Previous States

The number of different puzzle configurations for a given input size (R,C) is given by

$$\frac{(R * C)!}{(C!)^R} \tag{1}$$

For a 5x5 input, we have 623360743125120 different configurations.

For a 7x7 input, we have 73636156661571896039825854620303360000 different configurations.

Memorizing Previous States

This is bounded by the number of possible moves M we can make which is $(2 * (R - 1) * (C - 1))^M$.

For a 7x7 input with 10 maximum moves, we could possibly generate up to 3743906242624487424 different configurations.

Not the best problem to memorize states.