Homework

Part 1 (minimal vector<T> implementation)

You need to implement a custom vector (`vec<T>`) with minimal functionality. You have to use either **`operator new/operator delete`** or **`malloc/free`**. That means you have to work with uninitialized (raw) memory. Don't mix with initialized new, use only the uninitialized version and construct objects with **`placement new`**.
The strategy you should use (which is also tested by the automated tests) is the following: The default constructor should allocate default size memory which is **`INITIAL_SIZE * sizeof(T)`**. When adding a new element with **`push_back`** you should check if there is no place to add a new element (**`size_used == capacity_`**) you should double the **`capacity_`**(not only increment, but double it, and copy the elements to the new memory). The unused memory from **`data + size_used`** to **`data + capacity_`** should stay uninitialized.

```
#define INITIAL_SIZE 10

template <typename T>
class vec
{
public:
      using value_type           = T;
      using size_type            = size_t;
      using reference            = value_type&;
      using const_reference      = const value_type&;
```

// You have to implement (at least) all the following methods

```
      vec();
      explicit vec(size_type);
      vec(size_type, const value_type&);
      vec(const vec);
      vec(vec<value_type>&& v2) noexcept;
      vec(const std::vector<value_type>&);
      explicit operator std::vector<value_type>() const;
      ~vec() noexcept;
      vec<value_type>& operator=(const);
      vec<value_type>& operator=(vec<value_type>&&) noexcept;
      template <typename Q>
      friend void swap_(vec<Q>&, vec<Q>&);
      void resize(size_type);
      void reserve(size_type);
      size_type size() const;
      size_type capacity() const;
      void push_back(const value_type&);
      void pop_back();
      reference operator[](size_type);
      const_reference operator[](size_type) const;
      value_type* cbegin();
      value_type* cend()'
private:
```

```
        value_type* data;
        size_t capacity_;
        size_t size_used;

        static const size_type default_init_capacity = INITIAL_SIZE;
};
template <typename T>
bool operator==(vec<T>& v1, vec<T>& v2)
{
        return v1.size() == v2.size() && v1.capacity() ==
v2.capacity() && equal(v1.cbegin(), v1.cend(), v2.cbegin(),
v2.cend());
}
```

Header file “vec_tests.h” includes 10 test functions, if implemented correctly all the tests should print “true”.

Part 2 (algorithmic problems on vec<T>)

Problem #1

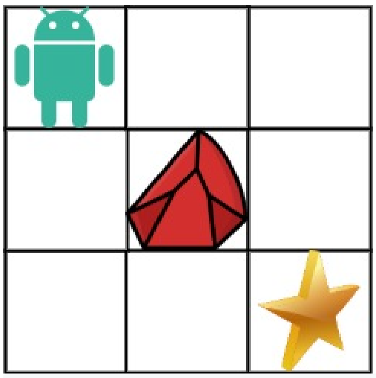
```
int unique_paths_with_obstacles(vec<vec<int>>& grid)
```

You are given an `m x n` integer vec<vec<int>>. There is a robot initially located at the top-left corner (i.e., `grid[0][0]`). The robot tries to move to the bottom-right corner (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

An obstacle and space are marked as `1` or `0` respectively in the grid. A path that the robot takes cannot include any square that is an obstacle.

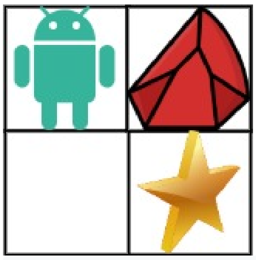
Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

Example 1:



```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

Example 2:



```
Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```

You can find more examples inside the **`void unique_paths_with_obstacles_tests()`** function. (Solve the problem with DP).