1 Modular Arithmetic (20 pts)

In C0, values of type `int` are defined to have 32 bits. In this problem we work with a version of C0 where values of type `int` are defined to have only 7 bits. In other respects it is the same as C0. All integer operations are still in two's complement arithmetic, but now modulo 2^7. All bitwise operations are still bitwise, except on only 7 bit words instead of 32 bit words.

Task 1 (10 pts). Fill in the missing quantities, in the specified notation.

a. The minimal negative integer, in decimal: -64

b. The maximal positive integer, in decimal: 63

c. -4, in hexadecimal: 0x 7C

d. 44, in hexadecimal: 0x 2C

e. `0x48`, in decimal: -56

Task 2 (10 pts). Assume `int x` and `int y` have been declared and initialized to unknown values. For each of the following, indicate if the expression always evaluates to `true`, or if it could sometimes be `false`. In the latter case, indicate a counterexample in the C0 dialect described here by giving a value for `x` and `y` that falsifies the claim. You may use decimal or hexadecimal notation.

a. `x >= x - 1` false, x = MIN_INT=-64

b. `~(x ^ (~x)) == -1` false for all x

c. `x+(y+1)-2*(x-1)-3 == -x+y` true

d. `(x!=-x || y!=-y) || x==y` false, x=0, y=MIN_INT=-64 or x=-64, y=0

e. `x <= (1<<(7-1))-1` true

3 Contracts Office (25pts)

In this problem, you are given contracts and want to find "*contract exploits*". For each of the contracts in the following tasks do:

- *Spot weaknesses* in the contract and briefly explain what the contract is missing to capture the informally stated purpose (one sentence explanation is sufficient).
- *Implement a simple contract exploit function*, i.e., a function that always satisfies the given contract without actually solving the informally stated problem.
  Give simple code! If your code is nontrivial (more than 3 lines), you need to provide loop invariants and a correctness argument to show why it actually satisfies the contract.
- Briefly sketch your approach on *how the contract can be fixed* (you do not need to give a full bug-fixed contract, a brief one sentence description of your approach is sufficient).

Hint: The following contract for integer square root is too weak, because the trivial implementation `{ return 0; }` is a contract exploit that clearly does not compute the square root:

```
int isqrt(int n)
//@requires n >= 0;
//@ensures 0 <= \result*\result && \result*\result <= n;
```

Task 1 (5 pts). The following contract was intended to "describe a fancy operation on two arrays to give a new array". The computation itself is so difficult and irrelevant, so only the array contract is important for this task.

Hint: Ignore what the fancy array operation might have been supposed to do. Try to implement the contract in $O(1)$.

```
int[] fancyarrayop(int[] A, int n, int[] B)
//@requires n > 0;
//@ensures \result[0] >= 0;
{
  return alloc_array(int, 1);
}
```

The contract does not look useful, because the `//@requires` contract leaves `\length(A)` and `\length(B)` unspecified, making it impossible for the function to access `A` or `B` safely. This effectively makes the arguments A and B useless and, thus, superfluous.

Suggested fix: add `\length(A)` and `\length(B)` constraints into `@requires`, preferably in relation to `n`, e.g.,

```
//@requires 0<=n && n <= \length(A) && n <= \length(B);
```

Task 2 (10 pts). The following contract was intended to specify a "super fast sort function". *Hint*: Ignore that the intention was to sort the input. Try to implement the contract super fast, i.e., faster than $O(n \log n)$.

```
int[] supersort(int[] A, int u)
//@requires 0 <= u && u <= \length(A);
//@ensures \length(\result) == u && is_sorted(\result, 0, u);
{
   return alloc_array(int, u);
}
// or
{
   for (int i = 0; i < u; i++)
       A[i] = 0;
   return A;
}
```

The contract is not exhaustive, because it does not rule out implementations that ignore and destroy the input data and result in an array that is sorted but contains different data.
Suggested fix: Add contract ensuring that `\result` is a permutation of A.

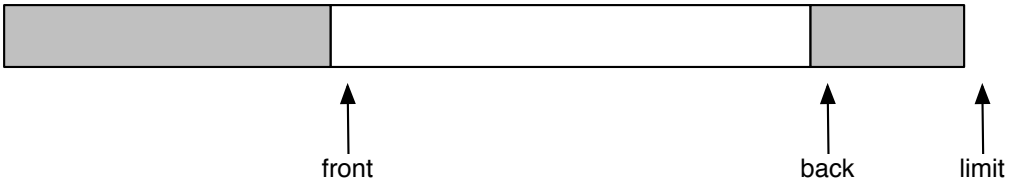
Task 3 (10 pts). The following contract was intended to specify a "super fast find function". *Hint*: Ignore that the intention was to find x in the array. Try to implement the contract super fast, e.g., faster than $O(\log n)$.

```
int supersonicfind(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
/*@ensures (-1 == \result && !is_in(x, A, n))
        || ((0 <= \result && \result < n) && A[\result] == x); @*/
{
   if (n == 0) return -1;
   A[0] = x;
   return 0;
}
```

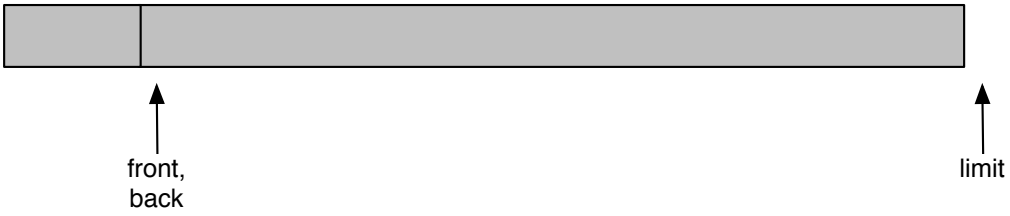
The contract is not quite exhaustive, because it does not rule out implementations that ignore and overwrite the original input data.
Suggested partial fix: at least `//@ensures A[\result]==\old(A[\result])`

1 Message Buffer (45 points)

You are implementing a network driver that stores the messages that it received in a queue. We can implement that queue in an array, remembering its length and two indices, one to the *front* of the queue and one to the *back*:



As usual, messages are inserted at the back and removed from the front. If either index ever passes the limit, it wraps around to the beginning of the array. You may assume that the limit is greater than zero. Due to this wrapping nature of the indices, the array can be visualized as a ring, which is why it is called a *ring buffer*. The *front* and *back* indices should only be equal if the queue is empty:




```
typedef struct queue* queue;
struct queue {
    int limit;
    message[] A;
    int front;
    int back;
};
```

As an example, here is the function that checks whether a queue is empty.

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
    return Q->front == Q->back;
}
```

(Continued)

Task 1 (10 pts). Identify all relevant data structure invariants for the (ring buffer) queue by implementing the is_queue function to check them.

```
bool is_queue(queue Q) {
```

```
  if (Q == NULL) return false;
  if (!(Q->limit > 0)) return false;
  //@assert \length(Q->A) == Q->limit;
  if (!(0 <= Q->front && Q->front < Q->limit)) return false;
  if (!(0 <= Q->back && Q->back < Q->limit)) return false;
  return true;
```

```
}
```

Task 2 (5 pts). Implement the deq function that dequeues the message at the front of the queue.

```
message deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
```

```
  assert(!queue_empty(Q));
  message e = Q->A[Q->front];
  Q->front = (Q->front + 1) % Q->limit;
  return e;
```

```
}
```

Task 3 (5 pts). How can you use the idea behind how unbounded arrays double arrays to make sure the queue is never full, i.e., that enq always works successfully (without losing messages), no matter how big the array of the queue used to be before calling enq? You do not need to give code.

Copy the array over into a new array that is twice as big whenever the old array is full. For future reference in Task 4, we call this function

```
void queue_double(queue Q);
```