3) (10 pts) DSN (Bitwise Operators)

In the game of NIM, there are several piles with stones and two players alternate taking 1 or more stones from a single pile, until there are no more stones left. The person who takes the last stone wins. It turns out that if it's someone's turn, if they play optimally, they can win as long as the bitwise XOR of all of the number of stones in each pile is not equal to 0. Write a function that takes in an array of values representing the number of stones in the piles of NIM and the length of that array, and returns 1, if the current player can win, and 0 otherwise, assuming both players play optimally.

```
int canWinNIM(int piles[], int numPiles) {

    int res = 0;
    for (int i = 0; i < numPiles; i++)
        res = res ^ piles[i];
    if (res != 0)
        return 1;
    else
        return 0;

    // return res == 0 ? 0 : 1;

}
```

3) (5 pts) ALG (Bitwise Operators)

There are a total of 25 cards, numbered 0 through 24. We can represent a set of cards with a single integer by setting the i^{th} bit to 1 if the set contains card i, and setting the bit to 0 otherwise. For example, the set of cards {2, 6, 7} would be stored as the integer 196, since $196 = 2^7 + 2^6 + 2^2$. Two sets of cards are disjoint, if and only if no card appears in both sets. Complete the function below so that it returns 1 if the sets of cards represented by the integers set1 and set2 are disjoint, and returns 0 if they are not disjoint. (For example, disjoint(196, 49) should return 1 because $49 = 2^5 + 2^4 + 2^0$, and there is no common value in the two sets {2, 6, 7} and {0, 4, 5}. On the other hand, disjoint(196, 30) should return 0 because $30 = 2^4 + 2^3 + 2^2 + 2^1$, so that card number 2 is included in both sets 196 and set 30.)

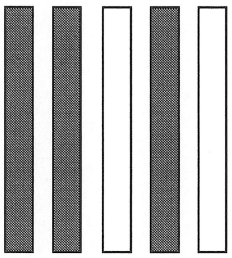
```
// Pre-condition: set1 and set2 are bitmasks in between 0 and
//                (1<<25)-1.
// Post-condition: Returns 1 if the two bitmasks are disjoint,
//                 meaning that the sets they represent don't have
//                 any items in common, and returns 0 otherwise, if
//                 the two represented sets do have common items.
int disjoint(int set1, int set2) {

    int temp = set1 & set2;
    if (temp == 0)
        return 1;
    return 0;

}
```

3) (10 pts) DSN (Bitwise Operators)

Imagine the task of painting a picket fence with 20 pickets. Let each picket be numbered from 0 to 19, from left to right and initially each is painted white. A pattern is 5 pickets long and can be placed with the pattern's left end aligned with any picket in between number 0 and number 15, inclusive. (If you line the pattern up with any of the pickets 16 through 19, the right end of the pattern goes past the right end of the fence and this isn't allowed.) Below is a picture of an example pattern, with the 3 of the 5 possible pickets painted:



If this pattern was lined up with picket number 4, then pickets 4, 5 and 7 would get painted. Think of the process as a placing a stamp on a portion of the whole fence. We can represent this pattern with the integer 11 ($2^0 + 2^1 + 2^3$), the integer where bits 0, 1 and 3 are set to 1. The bit positions represent, relative to the left end of the pattern, which positions have paint on them.

One way to paint a fence with a pattern is to line up the pattern with a few different picket numbers and apply the pattern. For example, if we lined up this pattern with the pickets at positions 1 and 4, then the pickets that would be painted would be at positions 1, 2, 4, 5 and 7. (Notice that painting an individual picket more than once leaves it still painted.)

Complete the function below so that it takes in an integer, ***pattern***, in between 0 and 31, inclusive, representing the pattern, an integer array, ***paintLoc***, which stores the locations to line up the pattern with for painting, and ***paintLen***, representing the length of the array ***paintLoc*** and returns a single integer storing the state of the painted fence (for each picket number, k, that is painted, bit k in the returned integer should be set to 1). Each of the values in ***paintLoc*** will be distinct integers in between 0 and 15, inclusive.

```
int paintFence(int pattern, int paintLoc[], int paintLen) {
    int paint = 0;
    for (int i=0; i<paintLen; i++) {
        paint = paint | (pattern << paintLoc[i]);
    }
    return paint;
}
```

paintLoc = [0, 5, 6]
(0)

01011 0
01011 5
01011 6
01011 10
0101111111101011 Fence

2) (5 pts) ALG (Sorting)

Show the result after each iteration of performing Bubble Sort on the array shown below. For convenience, the result after the first and last iterations are provided. The first row of the table contains the original values of the array.

Iteration	Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7
0	12	2	8	19	13	7	1	16
1	2	8	12	13	7	1	16	19
2								
3								
4								
5								
6								
7	1	2	7	8	12	13	16	19

3) (10 pts) DSN (Bitwise Operators)

There are 20 light switches, numbered 0 to 19, each which control a single light. Initially, all of the lights the switches control are off. There are several buttons. Each button toggles several switches, when pressed. For example, if a button toggles the switches 3, 5 and 9, then pressing the button changes the state of the three switches 3, 5 and 9, leaving the other switches in the same state. (So, if lights 3 and 5 were on and light 9 was off, after the button is pressed, lights 3 and 5 would be off and light 9 would be on.) Each button can be stored in a single integer, where the k^{th} bit is set to 1 if that button toggles the k^{th} switch, and set to 0 if pressing the button doesn't affect the k^{th} switch. For example, the button described would be stored as the integer 552 since $2^3 + 2^5 + 2^9 = 552$. Write a function that takes in an array, ***buttons***, storing the buttons to press and an integer ***len***, representing the length of the array ***buttons*** and returns a single integer storing the state of the **lights** after each of the buttons has been pressed once, assuming that all of the lights were off before any of the button presses. The format for storing the state of the lights should be identical to the format of the buttons.

```
int pressButtons(int buttons[], int len) {
    int state = 0; //if all on, state = (1<<20)-1;
    for (int i = 0; i<len; i++) {
        state = state ^ buttons[i];
    return state;
}
```

3) (10 pts) DSN (Bitwise Operators)

An organization has 30 groups of employees, labeled as group 0, 1, 2, …, 29. Each individual employee is assigned to some subset of those groups. The set of groups to which an employee belongs can be stored in a single integer, called the employee's ACCESS CODE, based on the bits of that integer. For example, an employee in groups 0, 3, 13 and 18 would have ACCESS CODE $2^0 + 2^3 + 2^{13} + 2^{18}$ (this is equal to 270345.) There are several shared drives at the organization. Each shared drive is accessible by any employees in a specified set of employee groups. The ACCESS CODE of a drive is specified exactly as that of an employee. If all employees who belong to either groups 2, 3 or 6 should have access to a shared drive, then that drive's access code is $2^2 + 2^3 + 2^6$ (76). An employee with ACCESS CODE 270345 would have access to a drive with ACCESS CODE 76, since the employee is part of group 3, and all employees in group 3 get access to the drive. Write a function that takes in an employee's access code, `empCode`, (as a single integer), an array of integers (`driveCodes`) storing the access codes of every shared drive in the organization, and the length of that array (`numDrives`), and returns the number of the shared drives that the employee with the given access code has access to.

```
int numDrivesAccess(int empCode, int* driveCodes, int numDrives) {
    int res = 0;
    for (int i = 0; i < numDrives; i++) {
        if (empCode & driveCodes[i])
            res++;
    return res;
}
```

Right Truncatable Primes
(Prefix Primes)

7 3

7 is prime
73 is prime

~~173~~

~~371~~

$d_0 d_1 d_2 d_3$

74

$d_0, d_0 d_1, d_0 d_1 d_2, d_0 d_1 d_2 d_3$ are all prime

skip 4 because 74 isn't prime

Backtracker is brute force where we skip options doomed to fail!

is prime divis check to sqrt

recursive function like odometer

mylists