Data Structure and Algorithm, Spring 2021
Homework 0

Release Date: February 12, 2021
Due Date: no due date; self-graded
Contact: dsa_ta@csie.ntu.edu.tw

Rules and Instructions

- In homework 0, the problem set contains 2 programming and 1 non-programming problems.
- In the entire semester, *C* (not *C++*) will be the only permitted programming language for the programming parts in both homework assignments and exams. This assignment is designed to help you verify if your *C* background is sufficient for this course. The homework is self-graded—you do not need to hand in your solutions. Doing homework 0 or not will not directly affect your final grade.
- However, if you cannot solve homework 0 properly, it is likely that you will experience difficulty and struggle with the subsequent homework sets in this course. You are strongly encouraged to make a serious attempt to complete all problems in homework 0. Paying tribute to our famous Professor P. Liu, "*It is never too late as you begin the endless Write-Test-Forever loop.*"
- For those who need tutorial and references, we recommend the following *C programming* videos presented by Professor Liu:
  - C Programming by Prof. P. Liu (the first six weeks)
  - C Programming by Prof. P. Liu (second part)
  - C2018
  - C2019
- If you have questions about HW0, please send an email to us, following the rules outlined below:
  - The subject should contain two tags, `"[HW0]"` and `"[Px]"`, specifying the problem where you have questions. For example, `"[HW0][P1] What does GCD mean?"`. Adding these tags allows the TAs to track the status of each email and to provide faster responses to you.
  - If you want to provide your code segments to us as part of your question, please upload it to Gist or similar platforms and provide the link. Screenshots or code segments directly included in the email is not allowed and will not be reviewed.

Problem Outline

- Problem 1 - Greatest Common Divisor of Big Integers (Programming)
  Foundations: array, loop
- Problem 2 - Nonogram Solver (Programming)
  Foundations: string, recursion, two dimensional array, etc.
- Problem 3 - *(Human Compiler) (Non-programming)
  Foundations: pointer, etc.

Problem 1 - Greatest Common Divisor of Big Integers (Programming)

Problem Description

The greatest common divisor $gcd(a, b)$ between two positive integers a and b is defined as the largest positive integer that divides both a and b without a remainder. Mathematically speaking, $\forall k \in \mathbb{N}$ and $k > gcd(a, b)$, $(a \not\equiv 0 \mod k)$ or $(b \not\equiv 0 \mod k)$.

GCD is a very powerful tool in modern cryptography, and when the target integers to be calculated are small (less than 10^8), GCD can be calculated in a few seconds with a naïve method. However, the numbers in modern cryptography requires at least 512 digits to prevent attackers from using a brute-force method to derive the secret key. This required number is too large for the naïve methods to calculate GCD in a reasonable time and the numbers exceeds the limit of even `long long` in the C language. In this problem, we will guide you to calculate the GCD of two big integers efficiently.

Implementation

First, to deal with the big integers, we need a "data structure", such as an integer array in C to represent larger values. We will call it BigInt. For instance, you can use an integer array where each element represents one (decimal) digit, like representing 17202 by the following code snippet.

```
int digits[10]={2,0,2,7,1};
```

It is not required to use the representation above, though. You can use any representation that facilitates your implementation of the following algorithm. It is an opportunity to think about how "well-designed data structures" can lead to "efficient algorithms."

Next, please implement the following algorithm, called Binary Algorithm, using the C language. Then you will miraculously get the correct GCD. Note that you need to implement four components: a comparator between two BigInts, subtraction between two BigInts, division by 2 for one BigInt, and multiplication by 2^k for one BigInt with some k. It is strongly recommended to implement the four components as separate functions, and test their correctness separately, before combining them with the Binary Algorithm.

Algorithm 1: Binary Algorithm for Greatest Common Divisor

Input: Two positive integers a and b.
Output: A positive integer ans representing greatest common divisor of a and b.
$n \leftarrow \min(a, b)$, $m \leftarrow \max(a, b)$, $ans \leftarrow 1$
while $n \neq 0$ and $m \neq 0$ **do**
  if n is even and m is even **then**
    $ans \leftarrow ans \times 2$, $n \leftarrow n/2$, $m \leftarrow m/2$
  else if n is even **then**
    $n \leftarrow n/2$
  else if m is even **then**
    $m \leftarrow m/2$
  end
  if $n > m$ **then**
    swap(n,m)
  $m \leftarrow (m - n)$
end
return $n \times ans$

Input

One line containing two integers, a and b, where $0 < a, b < 10^{256}$.

Output

Please output an integer representing $gcd(a, b)$.

Sample Input 1

```
20210208 80201202
```

Sample Output 1

```
6
```

Sample Input 2

```
98765432198765432198765432 123456789123456789123456789
```

Sample Output 2

```
9000000009000000009
```

Correctness (Optional)

You are encouraged to think about how we can prove that the Binary Algorithm is correct. The proof can be done if you can prove the correctness of the following theorems *from the definition of gcd*. We will teach you more about proving the correctness of algorithms in the semester, but feel free to make attempts by yourself now.

1. Assume that k is a positive integer and $k \mid a$ but $k \nmid b$, then $gcd(a/k, b) = gcd(a, b)$
2. Assume that k is a positive common divisor of a and b, then $k \cdot gcd(a/k, b/k) = gcd(a, b)$
3. Assume that $a > b$, then $gcd(a, b) = gcd(a - b, b)$

Problem 2 - Nonogram Solver (Programming)

Problem Description

Nonogram, also known as Paint by Numbers, Picross, Griddlers, Pic-a-Pix, and various other names, is a logic puzzle in which cells in a grid must be painted according to the given row clues and column clues. Players are requested to paint each grid cell into either "white" or "black", such that the segments consisting of consecutive black cells in each row or column matches the corresponding row clue and column clue. The clues or numbers are a form of discrete tomography that measures the numbers of consecutive painted cells in any given row or column. For example, a clue of "`4 8 3`" would mean that there are sets of four, eight, and three filled (black) cells, in that order, with at least one blank (white) cell between successive sets. Moreover, solving nonogram puzzles is proved to be a NP-complete problem, which roughly means that no *known* algorithm can solve the puzzles efficiently in general. See Figure 1 for an example.

If you are interested in this puzzle game, you can play the simple version (link) or download similar games to your smartphone.

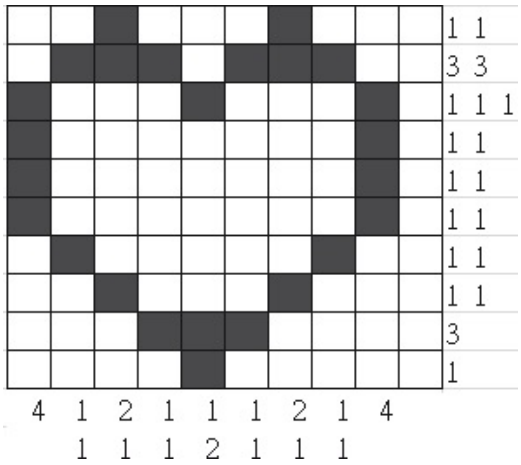

Figure 1: A nonogram example: clues and solution.

Now given the height and width of the nonogram puzzle, denoted as N and M, and clues of each row and column, please put together a program to solve this puzzle and draw it.

Input

The first line of the input contains two integers N and M, representing the number of rows and the columns of the nonogram puzzle.

Next $N + M$ lines are the clues. The first N lines represent the clues of the N rows while the rest M lines represent the clues of the M columns. Each clue line starts with an integer n, representing the number of segments of consecutive black cells the row or column contains. The next n integers represent the numbers of consecutive black cells in the segments. For row clues, the numbers are marked from left to right, and, for column clues, the numbers are marked from top to bottom.

The puzzle size $N \times M$ is less than 25, meaning that a brute-force algorithm with little optimization should be able to solve the puzzle within reasonable time. However, if you are interested in better solutions, try out other approaches (e.g., develop new algorithms or improve pruning mechanisms) and see if you can solve larger puzzles.

Output

Output the puzzle in N lines; each of them should contain M cells. Print `'o'` if the corresponding cell is black; otherwise print `'_'`.

Sample Input 1

```
4 4
2 1 1
2 1 1
2 1 1
1 4
2 1 2
2 1 1
2 1 2
2 1 1
```

Sample Output 1

```
o_o_
_o_o
o_o_
oooo
```

Sample Input 2

```
5 5
1 4
1 2
1 3
1 2
1 4
2 1 1
2 3 1
3 1 1 1
2 1 3
2 1 1
```

Sample Output 2

```
_oooo
oo___
_ooo_
___oo
oooo_
```

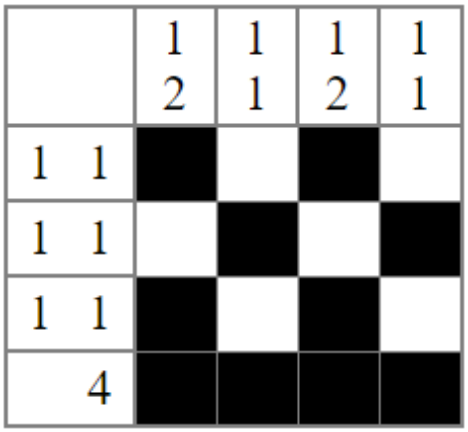

Figure 2: Puzzle of sample 1

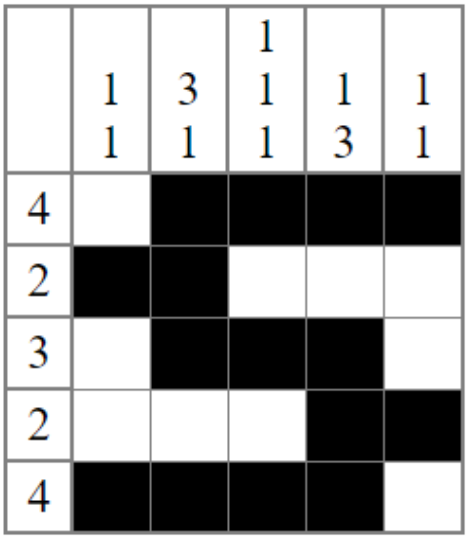

Figure 3: Puzzle of sample 2

Hint

If you don't know where to start with, you can follow the template below. Note that it is written in the form of *pseudo code*, and you have to covert it to real C code, where implementation details need to be carefully

considered.

Algorithm 2: Template for a recursive function

```
Function Recur(board):
    if board filled with colors then
        if board is valid then
            print board ;
            return True
        else
            return False
    else
        cell = find the next empty cell on board;
        /* try to paint it black                                    */
        paint cell black;
        succeed = Recur(board);
        if succeed then
            /* once succeed, we can stop trying                     */
            return True
        /* try to paint it white                                    */
        paint cell white;
        succeed = Recur(board);
        if succeed then
            /* once succeed, we can stop trying                     */
            return True
        /* what if we don't make it empty?                          */
        make cell empty;
```

Reference

1. Rules about Nonogram. (https://en.wikipedia.org/wiki/Nonogram)
2. Example puzzle in problem 2. (https://www.chessprogramming.org/Nonogram)

Problem 3 - *(Human Compiler) (Hand-written)

Problem Description

There are four questions related to *pointers* in *C*. In each of the following questions, you will be given an incomplete code segment and its purpose. Without the help of compilers, please fill in blank segments and achieve the purpose. After completing the code segments, you are encouraged to run the codes and verify the correctness on your own.

Compared with *Python*, it is easy to get a *Runtime Error* (RE) when dealing with *pointers* in *C*. For example, the following code segment may lead to a *Segmentation Fault* (depends on the compiler).

```
int *ptr = NULL;
*ptr = 42;
```

This may look stupid, but it somehow occurs when your code grows to have more than 100 lines. Therefore, you need to be very cautious when you allocate, access and free pointers. The problems below are good practices for you; think twice before going to the next one.

Problem 3-(a) Swaps two arrays using pointers.

```
int fake_a[] = {1, 3};
int fake_b[] = {2, 4};
int *real_a = fake_a;
int *real_b = fake_b;

for (int i=0; i<2; i++)
    printf("%d ", *(real_a + i));
for (int i=0; i<2; i++)
    printf("%d ", *(real_b + i));

int *tmp = real_a;
___(1)___ = ___(2)___;
___(3)___ = ___(4)___;

for (int i=0; i<2; i++)
    printf("%d ", *(real_a + i));
for (int i=0; i<2; i++)
    printf("%d ", *(real_b + i));
```

The output should be `"1 3 2 4 2 4 1 3"`.

Problem 3-(b) An array supporting negative indices.

```
#include <stdio.h>
#define MINN -50 // inclusive
#define MAXN 50 // inclusive

int main(){
    int storage[MAXN - MINN + 1]={0};
    int *ary = ___(1)___;

    for (int i=MINN; i<=MAXN; i++)
        ary[i] = i;
    for (int i=MINN; i<=MAXN; i++)
        printf("%d ", ary[i]);
    return 0;
}
```

The output should be `"-50 -49 ... -1 0 1 ... 49 50 "`.

Problem 3-(c) Traverses data nodes in a *linked list*. Please familiarize yourself with linked list in advance. Related topics are covered in Prof. Liu's videos.

```c
#include <stdio.h>
#include <stdlib.h> // malloc / free
#include <memory.h> // memset

// Use typedef to define "struct node" as "node".
typedef struct node{
    int data;
    struct node *nxt;
} node;

node *alloc(int data, node *nxt){
    node *tmp = (node *)malloc(sizeof(node));
    tmp->data = data;
    tmp->nxt = nxt;
    return tmp;
}

void destory(node *head){
    if (___(1)___){
        destory(head->nxt);
        // clean sensitive data.
        memset(head, 0, sizeof(head));
        free(head);
    }
}

int main(){
    // create nodes [0, 1, 2, 4]
    node *head = alloc(0, alloc(1, alloc(2, alloc(4, NULL))));
    node *tmp = head;

    // print the nodes subsequently
    while (tmp != NULL){
        printf("%d -> ", ___(2)___); // print the data in the node
        tmp = ___(3)___;
    }
    printf("NULL");

    // free the nodes subsequently to avoid memory leak
    destory(head);
    return 0;
}
```

The output should be "0 -> 1 -> 2 -> 4 -> NULL".

Problem 3-(d) Traverses data nodes in a *binary tree*. Please familiarize yourself with binary trees in advance. Related topics are covered in Prof. Liu's videos.

```c
#include <stdio.h>
#include <stdlib.h> // malloc / free
#include <memory.h> // memset

// Use typedef to substitute "struct node" with "node".
typedef struct node {
    int data;
    struct node *left, *right;
} node;

// creates a node filled with predefined values
node *alloc(int data, node *left, node *right){
    node *tmp = (node*)malloc(sizeof(node));
    tmp->data = data;
    tmp->left = left;
    tmp->right = right;
    return tmp;
}

// traverses the nodes recursively
void traverse(node *root){
    if (___(1)___){
        printf("%d ", root->data);
        traverse(___(2)___);
        traverse(___(3)___);
    }
}

// frees the nodes recursively
void destory(node *root){
    if (___(1)___){
        // recursively destory nodes.
        destory(___(2)___);
        destory(___(3)___);
        // clean sensitive data.
        memset(root, 0, sizeof(root));
        free(___(4)___);
    }
}

int main(){
    // creates a hierarchical data structure
    node *root = \
        alloc(0,
            alloc(3,
                alloc(7, NULL, NULL),
                alloc(4, NULL, NULL)
            ),
            alloc(2,
                alloc(1, NULL, NULL),
                alloc(9, NULL, NULL)
            )
        );
```

```
    // traverses the nodes one by one
    traverse(root);

    // frees the nodes to avoid memory leak
    destory(root);
    return 0;
}
```

The output should be "0 3 7 4 2 1 9".