Question 1

My Approach:-

```
void convert_to_power(vector<int>& queries, int q) {    // To convert
enchantments integer to 2*integer
   for (int i = 0; i<q; i++) {
      queries[i] = pow(2, queries[i]);
   }
}
```

Made a function convert_to_power() which will convert the array elements(x) to 2^x such that it will be easy to check the divisibility.

For example:- {1, 2, 3} will get converted to {2, 4, 8}

```
convert_to_power(queries, q);
for (int i = 0; i<q; i++) {
   int query = queries[i];
   for (int j = 0; j<n; j++) { // looping the array to make enchanted
      if (arr[j] % query == 0) {       // If element divisible by query
         arr[j] += query/2;    // adding to make it enchanted
      }
   }
}
```

algorithm here to change the array to make it enchanted:-

- ➔ Looping through the queries array and for each element checking the elements of array which is to be enchanted.
- ➔ If divisible then add query/2 to it because query contains (2^x) according to question and adding 2^(x-1) means adding query/2 to the element to maintain the enchantment. This keeps the "enchantment" rule consistent.

arr[] = {8, 12, 16};

queries[] = {2, 4};

Loop 1 (query = 2):

8 % 2 == 0 → add 1 → becomes 9

12 % 2 == 0 → add 1 → becomes 13

16 % 2 == 0 → add 1 → becomes 17

Now: arr = {9, 13, 17}

Loop 2 (query = 4):

9 % 4 == 1 → skip

13 % 4 == 1 → skip

17 % 4 == 1 → skip

So final arr = {9, 13, 17}

Analysis of Time complexity:-

q:- no. of queries and

n:- no. of elements in array

```
void convert_to_power(vector<int>& queries, int q) {    // To convert
enchantments integer to 2*integer
   for (int i = 0; i<q; i++) {
      queries[i] = pow(2, queries[i]);
   }
}
```

Need to traverse every element of queries to convert so O(q) < O(n*q)

```
for (int i = 0; i<q; i++) {
   int query = queries[i];
   for (int j = 0; j<n; j++) { // looping the array to make enchanted
      if (arr[j] % query == 0) {       // If element divisible by query
         arr[j] += query/2;    // adding to make it enchanted
      }
   }
}
```

The Outer loop here traverses queries array:- Total q iterations

The inner loop traverses the array for enchantment:- Total n iterations

Overall Time complexity:- O(n*q)

Which is ok according to constraints because:-

It is guaranteed that the sum of n and the sum of q over all test cases does not exceed 2×10^5.

In worst case:- n = 10^5 and q = 10^5

n*q = 10^10 which can't happen because it's the **sum** of all ni and qi across test cases that is ≤ 2×10^5. So ni and qi both can't be very large together.

Analysis of Space complexity:-

```
vector<int> arr(n);
vector<int> queries(q);
```

First vector takes O(n) space and second vector O(q) space

So overall Space complexity = O(n+q)

Question 2

My Approach:-

```
int cal_bit(const vector<int>& v, int i, int j) { // calculating total
bitwise and of vector from i to j index
    int ans = v[i];
    for (int x = i; x<=j; x++) {
        ans = ans & v[x];
    }
    return ans;
}
```

Made cal_bit function which will return the total bitwise in a vector v from index I to index j

For example:- v = {1, 2, 3,4,5} , i = 1, j = 3

The function will return (2 & 3 & 4) i.e from first to third index bitwise &.

```
if (vec.size() == 0) { // if no element the printing 0 and continuing
    cout<<"0"<<endl;
    continue;
}
```

```
int max_val = vec[0];
for (int i = 0; i<n; i++) {
    for (int j = i; j<n; j++) {
        int ans = cal_bit(vec, i, j);
        if (ans > max_val) {
            max_val = ans;
        }
    }
}
```

Initially storing maximum value that can be obtained by computing the bitwise AND over any non-empty subarray to the first element.

The algorithm is as follows:-

1) The outer loop (i from 0 to n-1) picks the starting index of a subarray.

2) The inner loop (j from i to n-1) picks the ending index of the subarray starting at i.

3) For every pair (i, j), a function cal_bit(vec, i, j) is called.

4) This function calculates bitwise over the subarray from index i to j.

5) The result from cal_bit() is stored in ans.

6) If ans is greater than the current max_val, then max_val is updated.

7) By the end of the loops, max_val will hold the maximum value returned by cal_bit() across all subarrays of the vector vec.

Basically checking every subarrays bitwise ritual value and storing them in ans. Maximum value is the output.

Analysis of Time complexity:-

```
int cal_bit(const vector<int>& v, int i, int j) { // calculating total
bitwise and of vector from i to j index
    int ans = v[i];
    for (int x = i; x<=j; x++) {
        ans = ans & v[x];
    }
    return ans;
}
```

The Time complexity of cal_bit() function is when we traverse the whole array to find bitwise

i.e O(n) where n is number of elements

```
for (int i = 0; i<n; i++) {
        for (int j = i; j<n; j++) {
            int ans = cal_bit(vec, i, j);
            if (ans > max_val) {
                max_val = ans;
            }
        }
    }
    cout<<max_val<<endl;
}
```

Outer loop runs from i=0 to i<n:- Traversing the array i.e O(n)

Inner loop contains the function cal_bit() which has O(n) time complexity

The total number of iterations per test case is:

Overall Time complexity : $O(n^2)$

For n = 100, this is about 5000 iterations per test case.

In the worst case: $1000 \times 100 \times 100 = 10^7$ operations

Therefore, the algorithm with $O(n^2)$ per test case is efficient and safe under the given constraints.

Analsis of space complexity:-

```
vector<int> vec(n);
```

This vector takes space complexity of O(n) per test case

All other variables takes O(1) space complexity

Thus:- Space complexity:- O(n)

Question 3

My approach:-

According to question

1)The first prime number, p_1, must be strictly greater than d.

2)The second prime number, p_2, must be strictly greater than p_1 and at least p_1 + d.

So we need to check whether a number is prime repeatedly.

For T($1 \le t \le 10^4$), this becomes inefficient to check primes for every test case.

```
void precompute_primes() {     // Precomputation for primes
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= MAX; i++) {
        if (is_prime[i]) {
            for (int j = i*i; j < MAX; j += i) {
                is_prime[j] = false;
            }
        }
    }
}
```

So, I precompiled the primes before handed using a function precompute_primes().

```
vector<bool> is_prime(MAX, true);
```

This vector stores Boolean elements :- True if it is prime and false if it is not

```
int findP1(int d) {    // function to find p1
    d++;        // strictly greater than d
    while (true) {
        if (is_prime[d]) {     // returning the first prime after d
            return d;
        }
        d++;
    }
}

int findP2(int d, int p1) {
    int p2 = p1 + d;
    if (p2 == p1) {
        p2++; // ensure p2 ≠ p1
    }
    while (true) {
        if (is_prime[p2]) {
            return p2;
        }
        p2++;
    }
}
```

Here are the two functions to calculate p1(next prime strictly greater than p) and p2()

```
cin>>d;
p1 = findP1(d);
p2 = findP2(d, p1);
cout<<p2*p1<<endl;
```

Finally calculating the product of p1 and p2. To keep it minimum we found the next prime after d as p1 and next prime p2 after p1+d ensuring p1 != p2.

Analysis if time complexity:-

sieve of Eratosthenes (precompute_primes) runs once and takes O(n log log n) time where n = 10^6.

So, time for sieve = O(10^6 log log 10^6) ≈ O(10^6)

In each test case, findP1(d) finds the next prime greater than or equal to d.

Similarly, findP2(d, p1) finds the next prime after p1 + d.

These loops run linearly until the next prime is found, so worst case time for each is O(d).

So total time complexity is O(10^6 + d).

Final answer:

Time Complexity: O(10^6 + d)

Analysis of space complexity:-

is_prime array is of size 10^6 → space used is O(10^6). For pre computation

All other variables like t, p1, p2, d, etc. are integers → O(1) space.

There is no recursion or extra memory allocation → no stack or heap overhead.

Therefore, total space complexity is O(10^6).

Question 4:-

My Approach

```
vector<pair<int,int>> Give_coordinates(int a, int b, int x, int y) {
    vector<pair<int,int>> result;
    result.emplace_back(x + a, y + b);
    result.emplace_back(x - a, y + b);
    result.emplace_back(x + a, y - b);
    result.emplace_back(x - a, y - b);
    result.emplace_back(x + b, y - a);
    result.emplace_back(x - b, y - a);
    result.emplace_back(x + b, y + a);
    result.emplace_back(x - b, y + a);
    return result;
}
```

The function Give_coordinates(a, b, x, y) returns 8 possible positions reachable from (x, y) using an L-shaped move:

- ➔ Move in all combinations of ±a and ±b, and their swaps.
- ➔ This mimics movement like a knight in chess but with custom steps a and b.
- ➔ It adds 8 possible coordinates using the combinations of `±a` and `±b`, and also swaps their positions to cover all directions.

The function returns a vector containing coordinates from which x, y is reachable using custom steps as mentioned in the problem.

We have pair<int, int> for saving {x, y} coordinate in each element of the vector.

Suppose you call the function like this:
-> Give_coordinates(2, 1, 4, 4)

This means:

- Step sizes: a = 2, b = 1
- Starting position: (x, y) = (4, 4)

The function will generate 8 positions using all combinations of ±a and ±b, and their swaps.

These 8 coordinates will be:

- (4 + 2, 4 + 1) → (6, 5)
- (4 - 2, 4 + 1) → (2, 5)
- (4 + 2, 4 - 1) → (6, 3)
- (4 - 2, 4 - 1) → (2, 3)
- (4 + 1, 4 - 2) → (5, 2)
- (4 - 1, 4 - 2) → (3, 2)
- (4 + 1, 4 + 2) → (5, 6)
- (4 - 1, 4 + 2) → (3, 6)

So the final output vector will be:

{(6,5), (2,5), (6,3), (2,3), (5,2), (3,2), (5,6), (3,6)}

➔ These are all the cells the piece can jump to from (4,4) using an L-shaped move with steps 2 and 1.

➔
```cpp
int count_common_elements(
    const vector<pair<int, int>>& v1,
    const vector<pair<int, int>>& v2
) {
    set<pair<int, int>> s1(v1.begin(), v1.end());
    set<pair<int, int>> s2(v2.begin(), v2.end());

    int count = 0;
    for (auto& p : s2) {
        if (s1.count(p)) count++;
    }

    return count;
}
```

This function gives the number of common elements between two vector using set stl in c++.

```cpp
int main() {
    int t, a, b, xk, yk, xq, yq;
    cin >> t;
    while(t--) {
        cin >> a >> b >> xk >> yk >> xq >> yq;

        vector<pair<int, int>> Reachable_from_k = Give_coordinates(a, b, xk,
yk);
        vector<pair<int, int>> Reachable_from_q = Give_coordinates(a, b, xq,
yq);
```

```
        int common_coordinates = count_common_elements(Reachable_from_k,
Reachable_from_q);
        cout << common_coordinates << endl;
    }
}
```

Main Algorithm:-

1. It then calls Give_coordinates(a, b, xk, yk) to get all 8 positions the first piece can jump to.
2. Similarly, it calls Give_coordinates(a, b, xq, yq) to get all 8 positions the second piece can jump to.
3. These two sets of positions are stored in Reachable_from_k and Reachable_from_q.
4. Then, it calls count_common_elements(...) to count how many positions are common in both sets.
5. Finally, it prints the number of common coordinates for that test case.

8. In short, the algorithm checks how many grid cells both pieces can jump to in one move, based on custom L-shaped movements.

Analysis of Time complexity:-

```
vector<pair<int,int>> Give_coordinates(int a, int b, int x, int y) { // O(1)
    vector<pair<int,int>> result;
    result.emplace_back(x + a, y + b);
    result.emplace_back(x - a, y + b);
    result.emplace_back(x + a, y - b);
    result.emplace_back(x - a, y - b);
    result.emplace_back(x + b, y - a);
    result.emplace_back(x - b, y - a);
    result.emplace_back(x + b, y + a);
    result.emplace_back(x - b, y + a);
    return result;
}
```

I. Always returns exactly 8 positions, regardless of input.
II. No loops depending on input size because just 8 emplace_back() calls.
III. So it runs in constant time → O(1)
IV.

```
    int count_common_elements(const vector<pair<int, int>>& v1, const
    vector<pair<int, int>>& v2) {
        set<pair<int, int>> s1(v1.begin(), v1.end());
        set<pair<int, int>> s2(v2.begin(), v2.end());

        int count = 0;
        for (auto& p : s2) {
            if (s1.count(p)) count++;
        }

        return count;
    }
```

1. It takes two vectors v1 and v2 — both always contain exactly **8 elements** (from Give_coordinates).
2. It creates two sets s1 and s2 from these vectors:

    - Each set has at most 8 elements.
    - Set construction takes O(k log k) where k = 8, so it's still constant time i.e O(1).

3. Then it loops through s2 (8 elements max), and checks if each element exists in s1.
4. Each lookup in a set takes O(log 8) = O(1) since 8 is constant.
5. So the total number of operations:

    i. Set creation: O(1)
    ii. Loop and lookup: 8 × O(1) = O(1)

I. For each test case, Give_coordinates is called twice once for each piece.
II. Give_coordinates always generates 8 coordinates, so its time complexity is O(1) (constant time).
III. Then count_common_elements is called:

IV. It creates two sets from the 8-element vectors → takes O(8) = O(1) time.

V. It loops through one set and checks existence in the other set → at most 8 checks → again O(1).

VI. So for each test case, total time complexity is O(1).
VII. For t test cases, total time complexity is:

VIII. O(t)

So, The final time complexity is O(1).

According the given constraints time complexity of O(1) is acceptable.

Analysis of space complexity:-

We are creating vectors and sets in particular instances but with definite sizes i.e 8 due to the directions which are 8 irrespective of input.

```
(t--) {
    cin >> a >> b >> xk >> yk >> xq >> yq;

    vector<pair<int, int>> Reachable_from_k = Give_coordinates(a, b, xk, yk);
    vector<pair<int, int>> Reachable_from_q = Give_coordinates(a, b, xq, yq);
    int common_coordinates = count_common_elements(Reachable_from_k,
Reachable_from_q);
    cout << common_coordinates << endl;
```

Doesn’t matter what the input is we will have 8 reachable coordinates only.

So, space complexity is O(1)