EXPLANATION AND TIME/SPACE COMPLEXITY

1) QUESTION NUMBER 1:

a) EXPLANATION:

```
for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]); //take input for array
}
for (int i = 0; i < q; i++) {
    int x;
    scanf("%d", &x);    //take input for enchantment
    int power = (int)pow(2, x);       // 2^x
    int add = (int)pow(2, x - 1);     // 2^(x - 1)
    for (int j = 0; j < n; j++) {
        if (a[j] % power == 0) {      //add if divisible by power
            a[j] += add;
        }
    }       You, 30 minutes ago • question 1 completed
```

- Take input for the array and the enchantment array.
- For each element in the enchant array run a for loop for elements in the actual array.
- If an element is divisible by 2^x, then add 2^{x-1}

b) TIME COMPLEXITY [**O(n × q)**]:

- Let n be the number of elements in the array.
- Let q be the number of enchantments (queries).
- For each enchantment we iterate throughout the array of size n to apply the condition given bellow

```
for (int j = 0; j < n; j++) {
    if (a[j] % power == 0) {      //add if divisible by power
        a[j] += add;
    }
}       You, 30 minutes ago • question 1 completed
```

- So total operations = **q × n.**
- **Time Complexity = O(n × q)** per test case.

c) SPACE COMPLEXITY [**O(n)**]:

- We only use an Array of size n.
- A few variables like x, power, add.
- And for enchantment array we are storing the values so q.
- **Space Complexity = O(n+q)** per test case.

2) QUESTION NUMBER 2:

a) EXPLANATION:

```
int max = 0;                    //max value
for (int i = 0; i < n; i++) {  //start subarray
    int curr = a[i];
    if (curr > max) max = curr;
```

- Start with a variable max to store the maximum AND result.
- For loop starts from i=0 till n, this is the start index of the subarray.
- Then a current value is assumed equal to a[i], because each element is a subarray of itself.
- If curr is greater than max, then max becomes curr.

```
for (int j = i + 1; j < n; j++) { //completes subarray
    curr = curr & a[j];
    if (curr > max) max = curr; //if the curr value is grater than the max, then set max equals cur
    if (curr == 0) break;
}
```

- The subarray ends at j, it starts from i as said.
- Now do the bitwise operator with curr itself.
- And satisfy the condition i.e., if curr is greater than max, then max becomes curr.

b) TIME COMPLEXITY [O(n²)]:

- Let n be the size of the array.
- The code uses **two nested loops**:
    - Outer loop which runs from 0 to n. (n times)
    - Inner loop which runs from i+1 to n. (n-i times)

```
for (int i = 0; i < n; i++) {  //start subarray
    int curr = a[i];
    if (curr > max) max = curr;

    for (int j = i + 1; j < n; j++) { //completes subarray
        curr = curr & a[j];
        if (curr > max) max = curr; //if the curr value is gr
        if (curr == 0) break;
    }
}
```

- In the **worst case**, the total number of subarrays is: $n(n+1)/2$
- **Time Complexity = $O(n^2)$** per test case.

c) <u>SPACE COMPLEXITY **[O(n)]**:</u>

- The array uses O(n) space.
- Only few variables like curr, max, i, j.
- **Space Complexity = O(n)** per test case.

3) QUESTION NUMBER 3:

a) <u>EXPLANATION:</u>

```c
//function to check if a number is prime
bool is_prime(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```c
//function to find the next prime greater than a given number
int next_prime(int start) {
    int num = start + 1;
    while (!is_prime(num)) {
        num++;
    }
    return num;
}
```

- The code uses **two functions:**
  - A function to check if a number is prime (Basic logic).
  - A function to find the next prime greater than a given number using the previous function (basic logic

```c
int p1 = next_prime(d);
int p2 = next_prime(p1 + d - 1);// p2 > p1 and p2 >= p1 + d
printf("%d\n", p1 * p2);
```

- Later then it uses both functions to get:
  - p1, **smallest prime** strictly greater than d.
  - p2, **smallest prime** strictly greater than p_1 and **at least p_1 + d.**
- Then return p1*p2 as the output.

b) <u>TIME COMPLEXITY **[O(√d)]**:</u>

- To check if a number k is prime → takes **O(√k)**.
- **Worst-case** for p1: each check takes **O(√p_1).**
- **Worst-case** for p2: each check takes **O(√p_2).**
- Time per test case = O(√p_1 + √p_2)

- Since $p_1 > d$ and $p_2 \geq p_1 + d$, the upper bound is **$O(\sqrt{(2d)})$**
- **Time Complexity = $O(\sqrt{d})$** per test case.

c) SPACE COMPLEXITY**[O(1)]**:

- No arrays used
- Only variables used
- **Space Complexity = O(1)** per test case.

4) QUESTION NUMBER 4:

a) EXPLANATION:

- We use two arrays i.e., dx[8], dy[8] to put all combinations of the directions the scout can move

```
int a, b;
int x1, y1, x2, y2;
scanf("%d %d", &a, &b);
scanf("%d %d", &x1, &y1); // Monument 1
scanf("%d %d", &x2, &y2); // Monument 2
dx[0] = a; dy[0] = b;
dx[1] = a; dy[1] = -b;
dx[2] = -a; dy[2] = b;
dx[3] = -a; dy[3] = -b;

dx[4] = b; dy[4] = a;
dx[5] = b; dy[5] = -a;
dx[6] = -b; dy[6] = a;
dx[7] = -b; dy[7] = -a;
```

- After taking input for the monuments and the steps scouts can walk, we must put all combinations of the scout's marchpast i.e.
  - $(x \pm a, y \pm b)$
  - $(x \pm b, y \pm a)$
  - These 8 positions are derived by permutating a and b with all sign combinations.

- Generate the 8 possible positions for monument 1 (i.e., from where the scout could've come to reach (x_1, y_1))
- Do the same for monument 2
- Count how many positions are common in both sets

```
int k = (a == b) ? 4 : 8;      //assuming a k to run th
for (int i = 0; i < k; i++) {
    int px = x1 + dx[i];
    int py = y1 + dy[i];

    // Check if same move also reaches (x2, y2)
    for (int j = 0; j < k; j++) {
        int qx = x2 + dx[j];
        int qy = y2 + dy[j];
         if (px == qx && py == qy) {
            count++;
            break; // no need to check further
        }
    }
```

- **One Problem that we can encounter here is, when a=b, we will get only 4 permutations, that is why we use k, k is 4 when a==b.**

b) <u>TIME COMPLEXITY [O(1)]:</u>

- For each test case:
  - 8 possible positions from each monument → constant time
  - Compare 8 × 8 = 64 position pairs → still constant
- **Time Complexity = O(1)** per test case.

c) <u>SPACE COMPLEXITY [O(1)]:</u>

- Only uses a few integer variables.
- No extra arrays or dynamic allocation.
- **Space Complexity = O(1)** per test case.