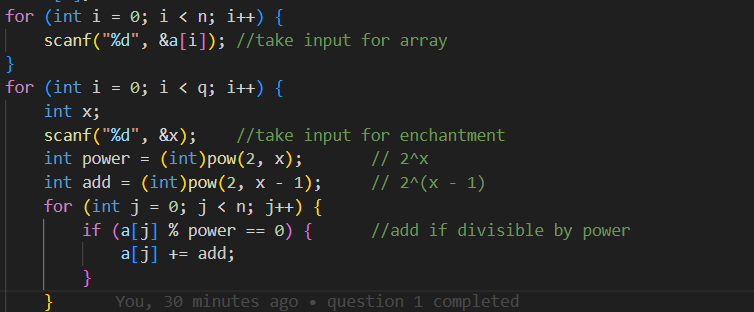
**EXPLANATION AND TIME/SPACE COMPLEXITY**

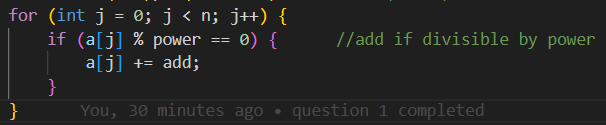
1. **QUESTION NUMBER 1:**
2. EXPLANATION:



* 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 2x, then add 2x-1

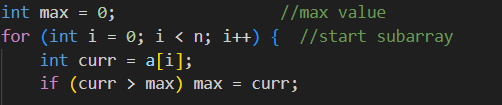
1. TIME COMPLEXITY [**O(n × q)]**:

* 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

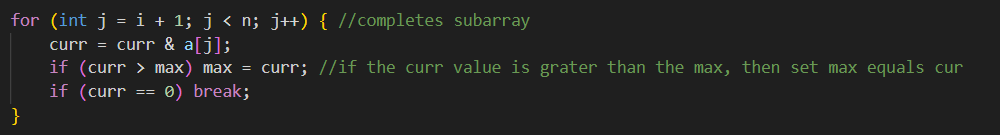


* So total operations = **q × n.**
* **Time Complexity = O(n × q)** per test case.

1. 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:**
3. EXPLANATION:



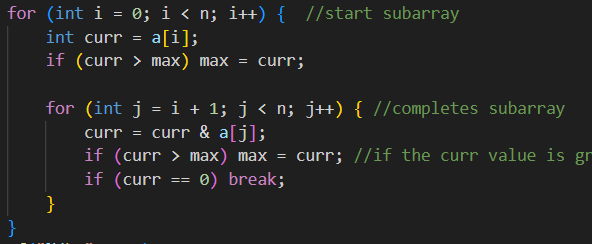
* Start with a variable max to store the maximum AND result.
* For loop starts from i=0 till n, this is the start index of the subarray.
* Then a current value is assumed equal to a[i], because each element is a subarray of itself.
* If curr is greater than max, then max becomes curr.



* 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.

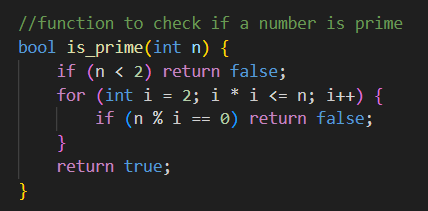
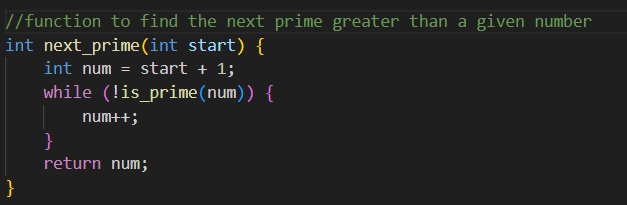
1. TIME COMPLEXITY **[O(n2)]**:

* 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)

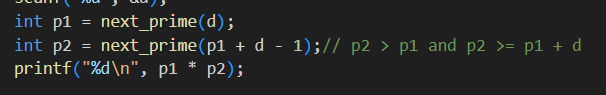


* In the **worst case**, the total number of subarrays is: n(n+1)/2
* **Time Complexity = O(n2)** per test case.

1. SPACE COMPLEXITY **[O(n)]**:
   * The array uses O(n) space.
   * Only few variables like curr, max, i, j.
   * **Space Complexity = O(n)** per test case.
2. **QUESTION NUMBER 3:**
3. EXPLANATION:

* 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).

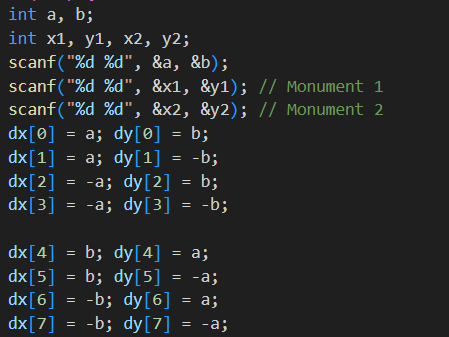


* Later then it uses both functions to get:
  + p1, **smallest prime** strictly greater than d.
  + p2, **smallest prime** strictly greater than p₁ and **at least p₁ + d.**
* Then return p1\*p2 as the output.

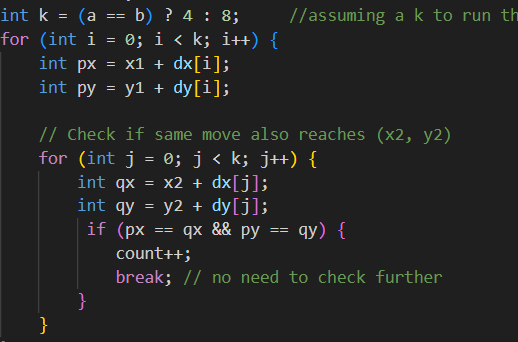
1. TIME COMPLEXITY **[O(√d)]**:

* To check if a number k is prime → takes **O(√k)**.
* **Worst-case** for p1: each check takes **O(√p₁).**
* **Worst-case** for p2: each check takes **O(√p2).**
* Time per test case = O(√p₁ + √p₂)
* Since p₁ > d and p₂ ≥ p₁ + d, the upper bound is **O(√(2d))**
* **Time Complexity = O(√d)** per test case.

1. SPACE COMPLEXITY**[O(1)]**:
   * No arrays used
   * Only variables used
   * **Space Complexity = O(1)** per test case.
2. **QUESTION NUMBER 4:**
3. EXPLANATION:
   * We use two arrays i.e., dx[8], dy[8] to put all combinations of the directions the scout can move



* + 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 ± a, y ± b)
      * (x ± b, y ± 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 (x₁, y₁))
  + Do the same for monument 2
  + Count how many positions are **common** in both sets



* + **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.**

1. TIME COMPLEXITY [O(1)]:
   * 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.
2. SPACE COMPLEXITY [O(1)]:
   * Only uses a few integer variables.
   * No extra arrays or dynamic allocation.
   * **Space Complexity = O(1)** per test case.