Intro:

Here are the points that will be covered today:

- What was Binary Search ?
- Advanced Use of Binary Search
- Two Pointers Technique

What Was Binary Search ?

Binary search is a searching algorithm. It solves the following generic problem:

Searching Problem:

Given a **sorted** array A of n integers and an integer x, find the position of x in A if it exists, or print -1 if x does not exist in A.

Solution:

This problem can be solved in $O(logn)$ using Binary Search algorithm. The idea is to utilize the fact that the array is sorted, and check the middle element of the array, let's call it Mid. If the Mid is greater then x, then this means that x can be only found to the left of Mid, else, if Mid is smaller than x, then x (if exists in A), can only be found to the right of of Mid. With this in mind, we have reduced our search space (number of elements through which we are searching) to half of the original size. Hence we perform the same analysis on the left or right half of the array, each time reducing the size by half, until -at worst case- we are left with only one element, which is either our answer, or a different number denoting that x does not exist in A.

Why O(logn) ?

Observe the number of steps we are performing depending on the size n:

$$\frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow ... \rightarrow 1$$

Mathematically, this is the definition of $log_2(n)$: the number of times you need to divide a number n by 2 to reach 1. Hence the complexity is $O(1)$

Advanced Use of Binary Search:

The previous section describes the classical use of Binary Search, and also the intuitive use which we all would think of when talking about Binary Search. Nevertheless, Binary search can be used to solve some problems which do not appear to require binary search or fit in its definition described above. Let's a motivating problem to understand better.

Motivating Problem:

We have n chairs, and these chairs are placed on 1-dimensional coordinate system starting from 0 and ending at 10^9. Each of these n chairs is located on some integer coordinate point, let these points be $x_1, x_2, ..., x_n$. This means that for any $i \in [1, n]$, it is mandatory that $x_i \in [0, 10^9]$, and n can be up to 10^6. There are m children to be placed in these seats, where $m \leq n$. But the problem is that each child would prefer to be placed as far as possible from other children.

Your task is to assign children into seats such that the minimum distance between any two children is maximum possible. In other words, among all the possible ways the children can be placed in these chairs, you need to choose the assignment for which if you calculate the minimum distance among every adjacent children, this minimum distance found is the maximum possible among all possible ways.

You only need to print the largest minimum distance possible.

Example:

- Chair Positions: $[1, 5, 17, 22, 25]$
- Number of children $(m) = 4$

The optimal solution in this case is to choose $[1, 5, 17, 25]$, because the minimum distance would be between the first two children $5 - 1 = 4$. Hence the output for this test case must be 4. Any other configuration would result in an answer more than or equal to 4. For example, we cannot take $[5, 17, 22, 25]$ because the minimum distance is 3 hence our previous answer is better.

Answer: Binary Search the Answer!

First, you might think that a greedy solution would work. Probably the solution you thought of is to start placing children on the sides, then put a child in the middle, and then take the middle positions of every interval and insert a child there. Here is a test case that would make your solution fail:

Positions of chairs: $[1, 3, 5, 7, 13, 25, 100]$ $(n = 7)$
Number of children : $m = 3$.

According to the greedy solution, you would choose $[1, 7, 100]$, which produced an answer of 7, where the optimal solution is to choose $[1, 25, 100]$, which gives an answer of 25. Hence this greedy approach will not work. We need somthing else.

You might think how can Binary search help us in this situation. The solution is to **perform Binary Search on the Answer!**.

Observe the following property of the problem: Let's take a random integer d, and let's assume that the answer should be greater than or equal to d. Can we validate if this is the case ? Yes we can. How ? Let's start placing the children from left most chair x_1. Then, let's place the next child in the first chair that satisfies $x_i - x_1 \geq d$, and in the same way keep placing the children in the first next possible chair as long as the distance between the newly chosen chair and previously chosen one if greater than or equal to d.

If we were able to place all of the m children before running out of chairs, then we have proven that the answer should be greater than or equal to d, so we can look for the answer in a higher range. If we have passed over all the chairs and we still have children to place this means that the answer cannot be greater then d, hence we must look for a lower answer. Do you see now where the binary search comes in ? Yes, what we will do is that we will pick some value in the middle of all possible solution values, do the same check we did above, from which we can find if we need to search in the left interval or the right interval. We keep reducing the size of the interval until we reach the greatest possible value for which our check returns true.

Understanding the Idea:

To get more intuition about the answer. Let's analyze what happens when we try d to be a very great value and a very small value. When d is very big, it would be difficult to place children next to each other so we might have to skip many chairs. This would lead us to run out of chairs before placing all children. On the other hand, with a small value of d (say 1), it is always possible to place the children as long as $m \leq n$. So as the value of d increases (from 1), we will still be able to find a solution until we reach a value of d with which we will not be able to find a solution. Not only that but we can prove that all the values greater than the threshold would not give us a solution. Hence, by being able to validate if some value d is a valid solution, we can use binary search the same way we search for numbers, but this time what tells to choose the left or the right interval is not the target value, but based on a check we do on the array, given that the check function has the following shape:

$$YES \,|\, YES \,|\, ... \,|\, YES \,|Threshold|\, NO \,|\, NO ... \; NO$$

Complexity Analysis:

What is the cost of all the mess? First, let's see how much 1 validation trial costs us. You should be convinced that validation costs $O(n)$, because it only requires a single iteration over the chairs. How many times do we validate ? $O(logn)$, or more precisely, $log(SizeOfSetOfAllPossibleSolutions)$, but we can approximate this to $O(logn)$. So, the complexity of the algorithm is $O(nlogn)$, which is enough to pass the constraint $n \leq 10^6$.

Code:

Algorithm 1: Binary Search the Answer

```
#include <bits/stdc++.h>
#define LL long long
#define PI pair<int,int>
#define PL pair<LL,LL>
#define st first
#define nd second
#define all(x) x.begin(),x.end()

using namespace std;

// define variables globally
// in order not to pass them as parameters
// because this takes time
int t,n,m;
vector<int> A;


// the function that validates the answer
bool Valid(int x){
  // copy m to and decrease k in every time we place a child
  int k = m;
  // initialize last to be a very small value
  // in order to always take the first element
  int Last = -1e9;
  for(int i = 0 ; i < n && k ; ++i){
    // if the distance between current chair
    // and last placed chair is greater than or equal tox
    // decrease k, and set this as the last chair
    if(A[i] - Last >= x){
      k--;
      Last = A[i];
    }
  }
  // this means return true if k == 0, false otherwise
  return k == 0;
}
```

```
int main(){
  // I use scanf usually because it is faster than cin
  scanf("%d␣%d",&n,&m);
  for(int i = 0 ; i < n ; ++i){
    int x; scanf("%d",&x);
    A.push_back(x);
  }
  // sort the array because it is not guaranteed to be sorted
  sort(all(A));
  // L is the left bound of the interval being processed now
  int L = 0;
  // R is the right bound of the interval being processed now
  int R = 1e9;
  // as long as our interval is of length greater than 1 keep searching
  while(L < R){
    // we take the middle of the interval
    // this is equivalent to M = (L + R) / 2; but faster.
    int M = (L + R) >> 1;
    // if valid, then reduce the range to half by setting
    // the left bound to the middle, or otherwise move the left bound
    if(Valid(M))
      L = M + 1;
    else
      R = M;
  }
  // The answer is L - 1, because at the threshold
  // R will be on a NO value, and L will be on a Yes value
  // Mid will be same as L, because integer division rounds down.
  // In which case since M will give YES, then L will be move upwards
  // to R and the loop will terminate, so to get the biggest YES we
  // subtract one
  printf("%d\n",L - 1);
}
```

The General Case:

To sum it all up, this technique of Binary Searching the answer can be used to find the answer of a problem, if it has a check function that can validate the answer for some value and this function must demonstrate two continuous chunks of consistents answers: a series of YES and then a series of NO (as in our example above) or vice-versa, a series of NO, and then a series of YES. Our answer would always lie in the threshold separating the two areas. You may find links to several problems that can be solved using this technique at the end of the document. The code above is the solution to one of these problems (called Agressive Cows on SPOJ).

Two Pointers Technique:

This technique can be used at places where usually Binary Search is used, but it results in a faster algorithm. We will try to understand the concept by solving the following problem.

Problem:

Given a **sorted** array A of size $n \leq 10^6$, and an integer x, we need to count the number of pairs of indexes i, j such that $i < j$ and $A[i] + A[j] = x$.

Binary Search Solution:

This problem can be solved using Binary Search, we start iterating from the left, and for each element at index i, we know that for it to exist in a pair, the other element in the pair must be $x - A[i]$. So we do a normal binary search on the interval $[i+1, n]$ to find $A[j] = x - A[i]$. If the element was found then we increment the solution by 1. We stop when we either reach the end of the array or we reach index j such that $A[j] > x$. The complexity of this approach is $O(nlogn)$ because we iterate over all elements, and we do a single binary search per index.

One important thing to note about this problem is that we need to see if numbers are unique or note. If numbers are not unique, then there is an extra work that needs to be done, because there might be more than one value that satisfies $A[j] = x - A[i]$. But since the elements are sorted, these elements would be adjacent. One solution would be doing 2 binary searches. 1 binary search to find the $A[j] = x - A[i]$ with smallest index, and another one to find the one with the highest index. then we can take the difference and add it to our answer.

So now we have an $O(nlogn)$ solution. Can we do better ? Yes, with the two pointers technique.

Two Pointers Approach:

Let's maintain two pointers i, j. Let i point to the beginning of A, and j to the end of A i.e. $i = 0,\ \ j = n - 1$. Observe that while the sum $A[i] + A[j]$ is greater than x, the only way to reach x is to decrease j, because increasing i would only make the sum greater, so we keep decreasing j until we have $A[i] + A[j] \leq x$. At this point, we can increase i, and whenever we hit $A[i] + A[j] = x$ we increase the total answer. We keep doing so until we again have $A[i] + A[j] > x$, so we decrease j again and follow the same described logic. We shall stop when i reaches the end of the array, or j reaches the start of the array.

As you might have noticed, each index i, j is visiting each array element at most once, hence this would cost $O(n)$! But why does this work ?

- When $A[i] + A[j] > x$, it always optimal to decrease j, because increasing i would set the sum $A[i] + A[j]$ to go further away from x. Hence we decrease.
- When $A[i] + A[j] \leq x$, it is always optimal to increase i, because if we decrease j, the sum $A[i] + A[j]$ would go further away from x, and we do not want this to happen. Hence we increase i.

Code:

Algorithm 2: Two Pointers

```
#include <bits/stdc++.h>
#define LL long long
#define PI pair<int,int>
#define PL pair<LL,LL>
#define st first
#define nd second
#define all(x) x.begin(),x.end()

using namespace std;


int n,x;
vector<int> A;

int main(){
  scanf("%d %d",&n,&x);
  for(int i = 0 ; i < n ; ++i){
    int x; scanf("%d",&x);
    A.push_back(x);
  }
  int i = 0;
  int j = n - 1;
  int Ans = 0;
  while(i < n){
    while(A[i] + A[j] > x && j > 0) j--;
    if(A[i] + A[j] == x) Ans++;
    i++;
  }
  cout << Ans << endl;
}
```

Problems:

- Agressive Cows: This is exactly the problem solved above but a different story.

- Books
- Worms
- Anton and Fairy Tale
- Interesting Drink
- Sagheer and Nubian Market
- A Problem asked by google previously.

References:

- https://codeforces.com/blog/entry/4586
- Comptitive Programming Handbook by Antti Laaksonen.
- The Algorithm Design Manual by Steven S. Skiena.
- https://www.topcoder.com/community/competitive-programming/tutorials/binary-search