Binary Search

Agenda

1. Motivation
2. Guessing game
3. Basic problem
4. Rotated sorted Array problem
5. Binary search on answer
6. To Solve

- **Motivation**

Imagine you have a large phone book with thousands of names listed in alphabetical order by first name. You need to find the phone number of a specific person, let's say "Hassan".

One way to find Hassan's phone number is to start at the beginning of the phone book and look at each name one by one until you find Hassan. This is called linear search. While this method will eventually find Hassan's phone number, it may take a long time if there are many names in the phone book.

Now, let's say you know that the phone book is sorted alphabetically by first name. You can use binary search to find Hassan's phone number more efficiently.

Here's how it works:

1. You start by looking at the middle name in the phone book (let's say it's "Ali").
2. Since "Hassan" comes after "Ali" in the alphabet, you know that Hassan's name must be located in the second half of the phone book.
3. So you ignore the first half of the phone book and repeat the process with the second half, looking at the middle name ("Kareem").
4. Since "Hassan" comes before "Kareem" in the alphabet, you know that Hassan's name must be located in the first half of the second half of the phone book.
5. You continue this process of dividing the search interval in half until you find Hassan's name.

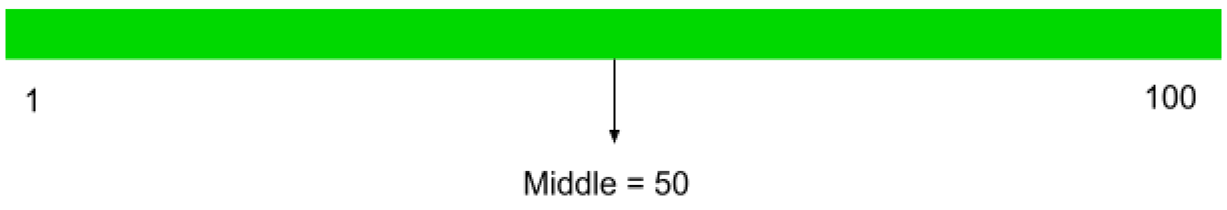
Using binary search, you can find Hassan's phone number much more quickly than with linear search because you're able to eliminate half of the phone book with each comparison.

In summary, binary search is a more efficient way to search a sorted array or list because it allows you to quickly eliminate half of the search space with each comparison. This can be especially useful when working with large datasets or when searching for a specific element multiple times.

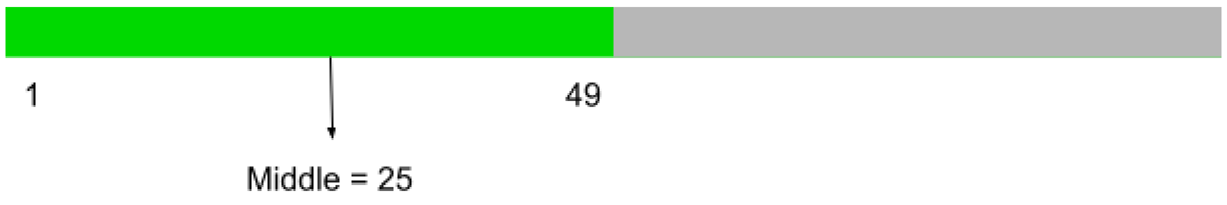
- **Guessing game:**

Let's say we're playing the guessing game in which I'm thinking of a number **T** between 1 and 100, and you're trying to guess what my number is by making a series of guesses. If your guess is too high, I will say "lower", and if your guess is too low, I will say "higher", and if your guess is equal to my number, I will say "equal" and the game is done. The goal is to find the number in the fewest number of guesses possible.

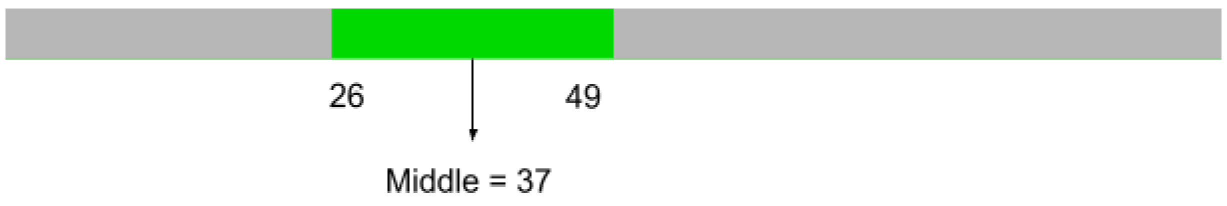
Suppose **T** = 34, let's start:
You guess the middle number, which in this case is 50.



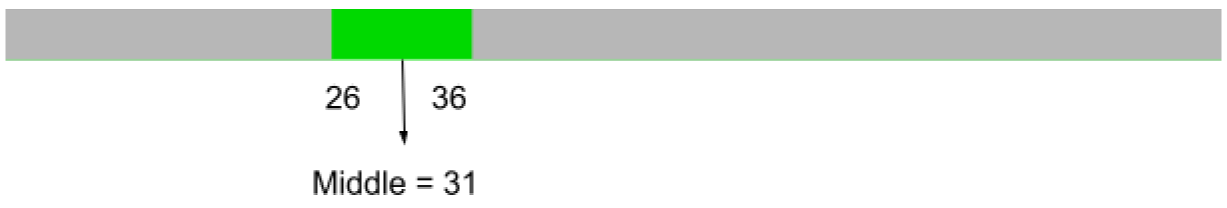
I say "lower". You now know that the number is somewhere between 1 and 49, so you guess the middle number of that range, which is 25.



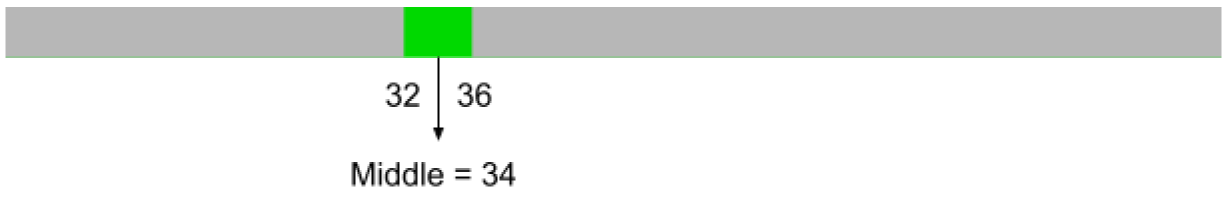
I say "higher". You now know that the number is somewhere between 26 and 49, so you guess the middle number of that range, which is 37.



You now know that the number is somewhere between 26 and 36, so you guess the middle number of that range, which is 31.



I say "higher". You now know that the number is somewhere between 32 and 36, so you guess the middle number of that range, which is 34.



I say "equal".

Using binary search, you were able to find the number in only 5 guesses, which is much more efficient than guessing randomly. By making a series of guesses and eliminating half of the remaining possibilities with each guess, you were able to quickly narrow down the range of possible numbers and find the correct answer.

In the worst case, it will take a maximum of 7 guesses to find the number out of 100 possible numbers, as log2(100) is approximately 7.

We use a left pointer and a right pointer to keep track of the possible range of numbers.

Code:

```
int BinarySearch(int n, int T) {
   int l = 1, r = n, mid;
   while (l <= r) {
       mid = (l + r) / 2;
       if (T == mid) return mid;
       else if (T < mid) r = mid - 1;
       else l = mid + 1;
   }
   return -1; // T does not exist
}
```

Time complexity: O(log2(n))

- **Basic problem:**

Given a sorted array **A** of **N** integers and **Q** queries, each asks for the index of a value **T**, find the index of **T** or report that **T** does not exist.

0	1	2	3	4	5	6	7
1	5	7	9	11	14	16	25

For example if T = 14, the answer is 5. If T = 6, the answer is -1.

Since the array is sorted, when we pick the middle index **mid**, we will always be able to know if **A[mid]** equals, lower than, or higher than **T**.

Steps for each query:

1. Set the left and right pointers to the minimum & maximum possible indices respectively: l = 0, r = N - 1.
2. Get the middle: mid = (l + r) / 2.
3. A[mid] has 3 cases:
   - Either A[mid] == T: return mid.
   - Or A[mid] < T: move the left pointer to mid + 1.
   - Or A[mid] > T: move the right pointer to mid - 1.
4. Repeat steps 2 & 3 until either you find the value T, or the range [l : r] becomes empty indicating that T does not exist.

Code:

```cpp
int BinarySearch(vector<int>& A, int T) {
   int n = (int)A.size();
   int l = 0, r = n - 1, mid;
   while (l <= r) {
       mid = (l + r) / 2;
       if (A[mid] == T) return mid;
       else if (A[mid] < T) l = mid + 1;
       else r = mid - 1;
   }
   return -1; // T does not exist
}

int main() {
   int n;
   cin>> n;
   vector<int> A(n);
   for (int i = 0; i < n; ++i)
       cin>> A[i];

   int q;
   cin>> q;
   while (q--) {
       int T;
       cin>> T;
       cout<< BinarySearch(A, T) <<"\n";
   }
}
```

- **Rotated sorted Array problem:**

You are given an array **A** of **N distinct** values, the array is sorted then rotated. Find the minimum value in the array. For example:

0	1	2	3	4	5	6	7	8
6	8	9	11	15	20	3	4	5

The minimum value is A[6] = 3.

Let's analyze the problem, a rotated sorted array consists of two parts: the first part contains all numbers greater than the end of the array, and the second part contains all numbers less than or equal to the end of the array.

We can do the following:

1. Let l = 0, r = N - 1.
2. Get the middle: mid = (l + r) / 2.
3. The middle element has two cases:
   - Either it's in the first part: then we need to narrow down the search range to [mid + 1 : r].
   - Or, it's in the second part: then we need to narrow down the searching range to [l : mid]. Note that we will not eliminate the mid itself from our range in this case, as it's possible that A[mid] is the smallest element.
4. Repeat steps 2 & 3 until the search range is narrowed down to only one element: the minimum value.

Code:

```cpp
int findMinimum(vector<int>& A, int T) {
   int n = (int)A.size();
   int l = 0, r = n - 1, mid;
   while (l < r) {
       mid = (l + r) / 2;
       if (A[mid] > A[n - 1]) l = mid + 1; // first part
       else r = mid; // second part
   }
   // l & r are now pointing to the minimum value
   return l;
}
```

This problem has other several versions, such as:

- Find a target value T in a rotated sorted array of distinct values.
- Find the minimum value in a rotated sorted array that may have duplicates.
- Find a target value T in a rotated sorted array that may have duplicates.

Try to think about how we can modify our binary search algorithm to solve each of these versions.

- **Binary search on answer:**

Example:

Let $f(x) = x^2$ where x is a positive value in a continuous range $[a, b]$, find the value of x such that $f(x)$ is closest to a specific target value t.

Solution:

First lets observe that $f(x)$ is a monotonic function, that is, $f(x)$ increases as x increases.

Let $l = a$, $r = b$. We aim to find a value x in the range $[l, r]$ such that x^2 is closest to t. We can imagine that the search space is split into two parts:

- The left part: $f(x) \leq t$.
- The right part: $f(x) > t$.

We need to find the last value of x in the left part and the first value of x in the right part then the answer is the one that has the closest value of $f(x)$.

We can simply binary search for the first x in the right part (let it be xr) and then get the last x in the left part (let it be xl) by decrementing the value of xr.

The first value of $f(x)$ that is greater than t is called the upper bound of t.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int f(int x) {
   return x * x;
}

int upperBound(int a, int b, int t) {
   int l = a, r = b, mid;
   while (l < r) {
       mid = (l + r) / 2;
```

```
        if (f(mid) <= t) l = mid + 1;
        else r = mid;
    }
    return l;
}

int main() {
    int a, b, t;
    cin >> a >> b >> t;
    int xr = upperBound(a, b, t);
    int xl = xr - 1 >= a ? xr - 1 : -1;
    cout << (xl == -1 || f(xr) - t < t - f(xl) ? xr : xl) << "\n";
    return 0;
}
```

- **To Solve:**
    - Queries about less or equal elements
    - Aggressive cows
    - Worms
    - Pair of Topics
    - Magic Powder - 2