Binary Search :

```
// recursive BS
class Solution {
private:
    int f(vector<int>& nums, int target, int low, int high) {
        if(low > high) return -1; // base case

        int mid = (low + high) / 2;
        // It is completely possible that low + high might cause an overflow,
        // hence try using low + (high - low) / 2
        if(nums[mid] == target) return mid;
        else if(nums[mid] > target) return f(nums, target, low, mid-1);
        else return f(nums, target, mid+1, high);
    }
public:
    int search(vector<int>& nums, int target) {
        int n = nums.size();
        return f(nums, target, 0, n-1);
    }
};
```

Lower bound : index of first element which is ≥ target

```
int lowerBound(const std::vector<int>& nums, int target) {
    int low = 0;
    int high = nums.size();
    int ans = n; // if in the case the target is greater than
    // all the elements we return n

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (nums[mid] >= target) {
                ans = mid;
            high = mid - 1; // looking for a smaller element on left
        } else {
            low = mid + 1; // go right
        }
    }

    return ans;
}
```

- We can directly use the

```
int ans = std::lower_bound(nums.begin(), nums.end(), target) - nums.begin();
```

- Given an sorted and a number return where the number is present if not present return where it should be inserted to maintain sorted-ness. → lower_bound
- Similarly → upper_bound ⇒ index of first element which is > target (no equal here)

Given a sorted array which allows duplicates. return the first and last indices where a target appears

```
vector<int> searchRange(vector<int>& nums, int target) {
  int n = nums.size();

  int first = lower_bound(nums.begin(), nums.end(), target)
  - nums.begin();
  int last = upper_bound(nums.begin(), nums.end(), target)
  - nums.begin();

  if(first < n && nums[first] == target) return {first, last-1};
  // If the target is greater than all elements in array it return n
  else return {-1, -1};
}
```

- It is possible people might not know about lb & ub so have to write BS code.

Given a sorted array and it is rotated by elements to either left or right. return the index where the target exists. O(n) is ok, but as it is sorted always go for BS.

```
int search(vector<int>& nums, int target) {
  int left = 0;
  int right = nums.size() - 1;

  while (left <= right) {
      int mid = left + (right - left) / 2;

      // Check if mid is the target
      if (nums[mid] == target) {
          return mid;
      }

      // Determine which side is sorted
      if (nums[mid] <= nums[right]) { // Right side is sorted
          if (nums[mid] < target && target <= nums[right]) {
              left = mid + 1; // Target is in the right sorted half
          } else {
              right = mid - 1; // Target is in the left half
          }
      } else { // Left side is sorted
```

```
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1; // Target is in the left sorted half
            } else {
                left = mid + 1; // Target is in the right half
            }
        }
    }

    return -1; // Target not found
}
```

- In case there are duplicates, there is one edge case where nums[left] == nums[mid] == nums[right] in that case both the sides are sorted

```
 if (nums[left] == nums[mid] && nums[mid] == nums[right]) {
    left++;
    right--;
} // Add this before you move to any side
```

Given a rotated sorted array find the min element in it

```
int findMin(vector<int>& nums) {
    int n = nums.size();
    int left = 0;
    int right = n-1;
    int mini = INT_MAX;
    while(left <= right) {
        int mid = left + (right - left) / 2;

        if(nums[mid] <= nums[right]) {
            mini = min(mini, nums[mid]);
            // If right side is sorted the min is mid
            right = mid-1;
        }
        else {
            mini = min(mini, nums[left]);
            // If left side is sorted the min is low
            left = mid+1;
        }
    }

    return mini;
}
```

Given an array it is rotated some times to right, how many times ?

- previous answer but return index of it.

Given a sorted array, the elements in are all a pair except one. return that

```
int singleNonDuplicate(vector<int>& nums) {
  int n = nums.size();

  // edge cases
  if(n == 1) return nums[0];
  if(nums[0] != nums[1]) return nums[0];
  if(nums[n-1] != nums[n-2]) return nums[n-1];

  int left = 1;
  int right = n-2;

  while(left <= right) {
      int mid = left + (right - left) / 2;
      if(nums[mid] != nums[mid-1] && nums[mid] != nums[mid+1]) {
          return nums[mid];
      }

      if(mid % 2 == 0) {
      // as a pair always starts with even and ends at odd index
      // if there is nothing wrong
          if(nums[mid-1] == nums[mid]) {
              right = mid-1;
          } else {
              left = mid+1;
          }
      } else {
          if(nums[mid+1] == nums[mid]) {
              right = mid-1;
          } else {
              left = mid+1;
          }
      }
  }

  return -1;
}
```

- I can not directly make -2 or +2 jumps as left would cross right

---

Given an array {NOT A SORTED ONE} return one of its peak elements

```
int findPeakElement(vector<int> &arr) {
    int n = arr.size();

    // assume arr[-1] == arr[n] == -infinity
    if (n == 1) return 0;
```

```
    if (arr[0] > arr[1]) return 0;
    if (arr[n-1] > arr[n-2]) return n-1;

    int low = 1, high = n-2;
    while (low <= high) {
        int mid = (low + high) / 2;

        // Check if its a peak
        if (arr[mid] > arr[mid-1] && arr[mid] > arr[mid+1]) {
            return mid;
        }
        // if present on increasing line
        else if (arr[mid] > arr[mid-1]) {
            low = mid + 1;
        }
        // if present on decreasing line
        else if (arr[mid] < arr[mid-1]) {
            high = mid - 1;
        }
        else {
                // if it is a dip, then we can go either of the direction
                // to get one peak
                low = mid + 1;
                // high = mid - 1;
        }
    }
    return -1; // Default return if no peak is found.
}
```

Finding Sqrt of a number using Binary Search

```
int floorSqrt(int n) {
    int left = 1;
    int right = n;
    int ans = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (mid * mid <= n) {
        // Take care when doing mid*mid can cause overflow so use long long
            ans = mid;
            // Update the answer to mid since
            // mid*mid is less than or equal to n.
            left = mid + 1;
            // Move to the right half to search for a larger possible value.
        } else {
```

```
            right = mid - 1;
            // Move to the left half to search for a smaller value.
        }
    }
    return ans;
}
```

- rather than having extra space for ans we can directly return high, as we always break out when high is on left of left which is the answer.

- nth root ?

```
int func(int mid, int n, int m) {
    long long ans = 1; // to avoid overflow
    for(int i = 1; i <= n; i++) {
        ans = ans * mid;
        if(ans > m) return 2; // mid is big left
    }
    if(ans == m) return 1; // found the anwer
    return 0; // mid is small go right
}

int NthRoot(int n, int m) {
    int low = 1, high = m;
    while(low <= high) {
        int mid = (low + high) / 2;
        int midN = func(mid, n, m);
        if(midN == 1) {
            return mid;
        }
        else if(midN == 0) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

- koko eating bananas (A monkey has to finish eating all bananas which are split into several piles)
- koko has to have a certain eating speed k(no.of bananas per hour as guards come back after a fixed time h)
- what is the minimum possible value of k ?
- the max that surely ensure this is max of piles (then no.of hours required would be size of piles)

```
class Solution {
private:
```

```
    int hrs_req(const vector<int>& piles, int k) {
        int ans = 0;
        for (auto pile : piles) {
            ans += ceil(pile / (double)k);
            // int / int would return a int not a float **
        }
        return ans;
    }
public:
    int minEatingSpeed(vector<int>& piles, int h) {
        int max_ans = *max_element(piles.begin(), piles.end());
        int l = 1, r = max_ans;
        int ans = max_ans;
        while (l <= r) {
            int mid = (l + r) / 2;
            if (hrs_req(piles, mid) <= h) {
            // search for a lesser possible speed
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1; // increase k
            }
        }
        return ans;
    }
};
```

Given an array where we are given blooming dates of flowers, we are to make m batches with k adjacent flowers each return the min number of days I have to wait. return -1 if not possible

```
class Solution {
private:
    int f(vector<int>& bloomDay, int k, int day) {
    // we interesting approach
        int counter = 0;
        int batches = 0;
        for(auto bday : bloomDay) {
            if(bday <= day) counter++;
            else {
                batches += counter / k;
                counter = 0;
            }
        }
        batches += counter / k; //**
        return batches;
    }
```

```
public:
    int minDays(vector<int>& bloomDay, int m, int k) {
        int min_ans = *min_element(bloomDay.begin(), bloomDay.end());
        int max_ans = *max_element(bloomDay.begin(), bloomDay.end());

        int l = min_ans;
        int r = max_ans;
        int ans = -1;

        while(l <= r) {
            int mid = (l + r) / 2;
            int possible_bouqets = f(bloomDay, k, mid);

            if(possible_bouqets >= m) {
                r = mid - 1;
                ans = mid;
            }
            else l = mid + 1;
        }

        return ans;
    }
};
```

- we could also do long(k) * m > N to check for not possible case.
- why long ? multiplying two int can cause overflow

Given the weights of some goods in an array, a ship with certain capacity and a delivery deadline.

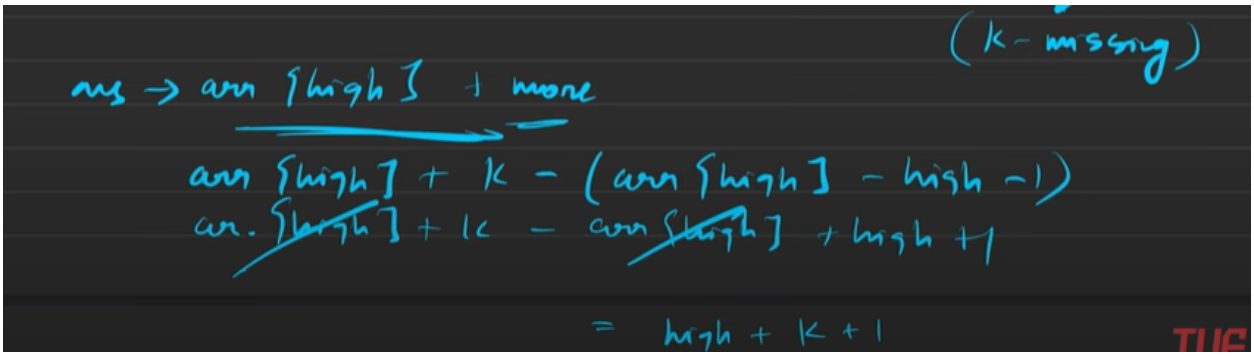
return the min capacity need to deliver on or before time

```
class Solution {
private:
    int f(vector<int> &weights, int cap) {
        int days = 1; // **
        int curr_cap = 0;
        for(auto w : weights) {
            if(cap - curr_cap >= w) curr_cap += w;
            else {
                curr_cap = w; // **
                days++;
            }
        }

        return days;
    }
public:
```

```
    int shipWithinDays(vector<int>& weights, int days) {
        int max_capacity = accumulate(weights.begin(), weights.end(), 0);
        int min_capacity = *max_element(weights.begin(), weights.end());

        int l = min_capacity;
        int r = max_capacity;
        int ans = 0;

        while(l <= r) {
            int mid = l + (r - l)/ 2;
            if(f(weights, mid) <= days) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }

        return ans;
    }
};
```

Kth missing element

```
int findKthMissing(const std::vector<int>& arr, int k) {
    int low = 0, high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        int missing = arr[mid] - (mid + 1);

        if (missing < k) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    // After the loop, low is the position where the
    // kth missing number should be.
    return low + k; // in the one that missing values left boundary exists
    // we add the more that is required to make it the kth missing
}
```



- low = high + 1
- why are we doing this ? it may be possible that high is -1 and arr[-1] you know!

Aggresive Cows :

- Given an array which denotes the stalls distance from some reference point. we need to place the cow in stalls such that if we find the min distance between any two pairs of cows, it should be maximum and return that max dis.

```
bool canWePlace(vector<int> &stalls, int dist, int cows) {
    int n = stalls.size(); //size of array
    int cntCows = 1; //no. of cows placed
    int last = stalls[0]; //position of last placed cow.
    // GREEDY APPROACH
    for (int i = 1; i < n; i++) {
        if (stalls[i] - last >= dist) {
            cntCows++; //place next cow.
            last = stalls[i]; //update the last location.
        }
        if (cntCows >= cows) return true;
    }
    return false;
}
int aggressiveCows(vector<int> &stalls, int k) {
    int n = stalls.size(); //size of array
    //sort the stalls[]:
    sort(stalls.begin(), stalls.end());

    int low = 1, high = stalls[n - 1] - stalls[0];
    // if we can place two cows then the max dis between
    // them would be extreme stalls
    //apply binary search:
    while (low <= high) {
        int mid = (low + high) / 2;
        if (canWePlace(stalls, mid, k) == true) {
            low = mid + 1;
        }
```

```
        else high = mid - 1;
    }
    return high;
}
```

Given b books, m students. Allocate all books in such a way each student has atleast one book. return the min of max no.of pages a student has (Allocate books, Painters partiton,)

```
int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.
    int students = 1;
    int pagesStudent = 0;
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) { // contiguous
            //add pages to current student
            pagesStudent += arr[i];
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i]; //**
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    // baki books atleast dhe sakte he
    int high = accumulate(arr.begin(), arr.end(), 0);
    while (low <= high) {
        int mid = (low + high) / 2;
        int students = countStudents(arr, mid);
        if (students > m) {
        // esa nhi hona chahiye, saare books inhhi m students me baatna he
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
```

```
    return low;
}
```

Median of two sorted arrays :

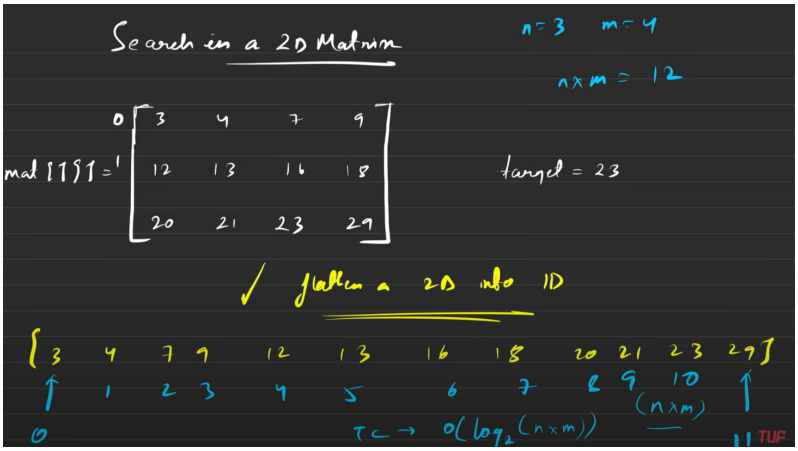
- As they are both sorted we can apply merge sort type of algo, but we would be taking extra space.
- But we need only the middle elements. Take two pointer approach

▼ merge sort extension

```
int findMedian(int a[], int n1, int b[], int n2) {
    int n = n1 + n2;
    int ind1 = n / 2;
    int ind2 = n / 2 - 1;
    int cnt = 0;
    int ind1el = -1, ind2el = -1;
    int i = 0, j = 0;

    while (i < n1 && j < n2) {
        if (a[i] < b[j]) {
            if (cnt == ind1) ind1el = a[i];
            if (cnt == ind2) ind2el = a[i];
            cnt++;
            i++;
        } else {
            if (cnt == ind1) ind1el = b[j];
            if (cnt == ind2) ind2el = b[j];
            cnt++;
            j++;
        }
    }

    while (i < n1) {
        if (cnt == ind1) ind1el = a[i];
        if (cnt == ind2) ind2el = a[i];
        cnt++;
        i++;
    }

    while (j < n2) {
        if (cnt == ind1) ind1el = b[j];
        if (cnt == ind2) ind2el = b[j];
        cnt++;
        j++;
    }

    if (n % 2 == 1) {
```

```
        return ind2el;
    } else {
        return (ind1el + ind2el) / 2;
    }
}
```

Given a sorted 2d matrix. return if an element is present in matrix or not

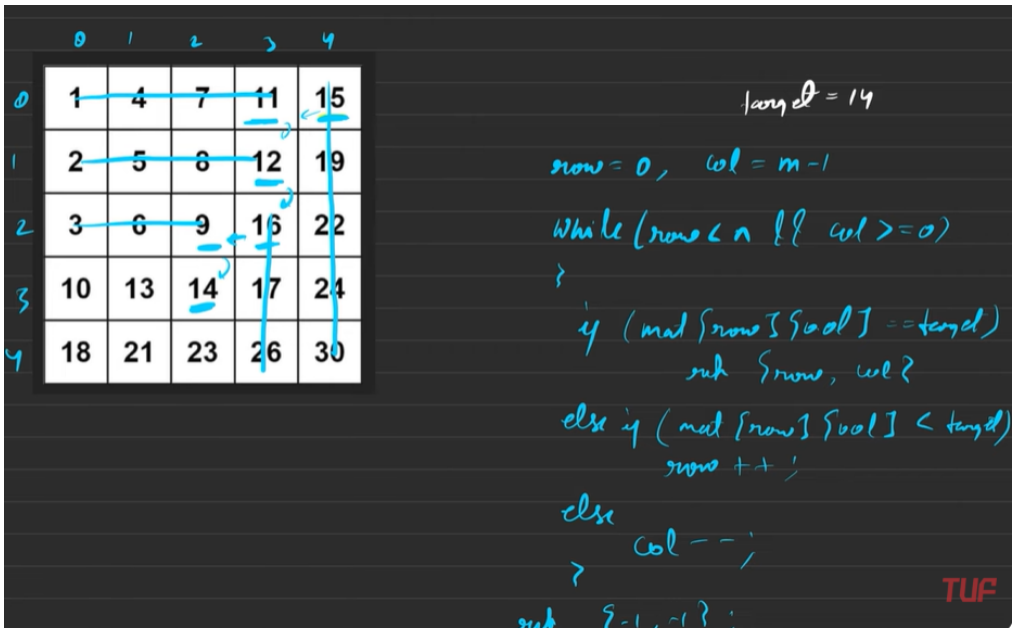


- Similar to Sudoku solver case 3 checking

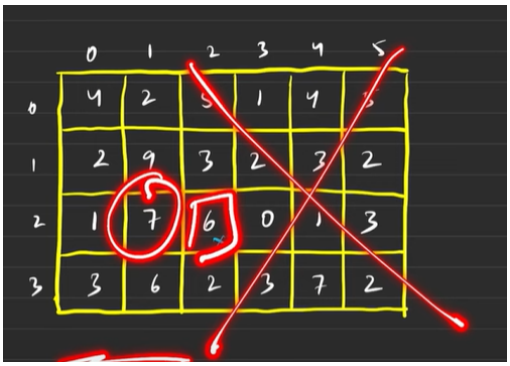
```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int n = matrix.size();
    int m = matrix[0].size();

    //apply binary search:
    int low = 0, high = n * m - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int row = mid / m, col = mid % m; //**
        if (matrix[row][col] == target) return true;
        else if (matrix[row][col] < target) low = mid + 1;
        else high = mid - 1;
    }
    return false;
}
```

Search in a 2D Matrix - II

- Either start from top-right or bottom left and keep moving by eliminating an entire row/column

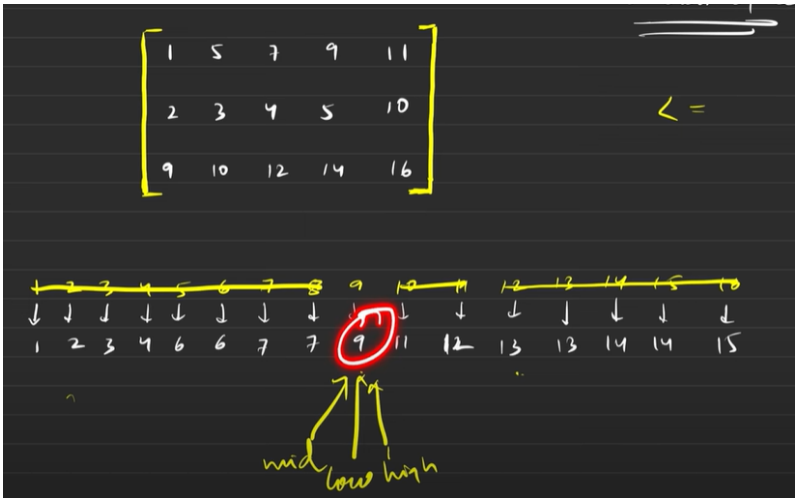


Find Peak Element-II 2d version



```cpp
int findMaxIndex(vector<vector<int>>& mat, int n, int m, int col) {
    int maxValue = -1;
    int index = -1;
    for(int i = 0;i<n;i++) {
        if(mat[i][col] > maxValue) {
            maxValue = mat[i][col];
            index = i;
        }
    }
    return index;
}
vector<int> findPeakGrid(vector<vector<int>> &mat){
    int n = mat.size();
    int m = mat[0].size();
    int low = 0, high = m-1;
    while(low <= high) {
        int mid = (low + high) / 2;
        int maxRowIndex = findMaxIndex(mat, n, m, mid);
        int left = mid - 1 >= 0 ? mat[maxRowIndex][mid - 1] : -1;
        int right = mid + 1 < m  ? mat[maxRowIndex][mid+1] : -1;
        if(mat[maxRowIndex][mid] > left && mat[maxRowIndex][mid] > right) {
            return {maxRowIndex, mid};
        }
        else if (mat[maxRowIndex][mid] < left) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return {-1,-1};
}
```

Median in a Row Wise Sorted Matrix



```cpp
int countSmallEqual(vector<vector<int>> &matrix, int n, int m, int x) {
    int cnt = 0;
    for(int i = 0;i<n;i++) {
        cnt += upperBound(matrix[i], x, m);
    }
    return cnt;
}
int median(vector<vector<int>> &matrix, int m, int n) {
    int low = INT_MAX; int high = INT_MIN;
    n = matrix.size();
    m = matrix[0].size();
    for(int i = 0;i<n;i++) {
        low = min(low, matrix[i][0]);
        high = max(high, matrix[i][m-1]);
    }

    int req = (n * m) / 2;
    while(low <= high) {
        int mid = (low + high) / 2;
        int smallEqual = countSmallEqual(matrix, n, m, mid);
        if(smallEqual <= req) low = mid + 1;
        else high = mid - 1;
    }
    return low;
}
```