

ChatGPT

This is a copy of a conversation between ChatGPT & Anonymous.



How to understand that a problem should be solved using binary search



Understanding when to apply binary search involves recognizing a few key characteristics of the problem:

1. **Sorted Data:** Binary search is most effective on sorted data (either ascending or descending order). If the problem involves searching or finding an element in a sorted array, list, or any data structure where order is maintained, binary search can be highly efficient.
2. **Divide and Conquer:** Binary search follows a divide-and-conquer approach. It repeatedly divides the search interval in half until the target element is found or the interval is empty. Problems that can be framed as recursively dividing the search space into halves are suitable candidates for binary search.
3. **Efficiency Requirements:** Binary search is very efficient with a time complexity of $O(\log n)$, where n is the number of elements in the data structure. If the problem demands efficient searching, especially with large datasets, binary search is a good choice.
4. **Single Point of Interest:** Binary search works best when there is a single point of interest (like finding a specific element or determining a boundary), rather than needing to access multiple elements simultaneously.
5. **Index-based Access:** Binary search is ideal for data structures that allow fast random access by index, such as arrays or lists. It's less suitable for linked lists where sequential access is required.
6. **Examples:** Common applications include searching for an element in a sorted array, finding the smallest or largest element that satisfies a condition (e.g., minimum or maximum in a rotated sorted array), and determining the existence of an element (e.g., checking if a number exists in a sorted list).

By identifying these characteristics in a problem, you can determine whether binary search is appropriate. It's crucial to ensure the data is sorted and that the problem's nature aligns with binary search's divide-and-conquer methodology for optimal efficiency.



in the context of competitive programming



In competitive programming, recognizing when to use binary search can be crucial for solving problems efficiently. Here are some tips to help identify such scenarios:

1. Searching in a Sorted Array:

- **Direct Search:** If the problem asks to find a specific element in a sorted array or determine its position, binary search is the go-to method.
- **Boundaries:** Tasks like finding the first or last occurrence of an element, or the lower and upper bounds, are typical scenarios where binary search applies.

2. Search on the Answer Space:

- Often, a problem asks to find the maximum or minimum value that satisfies certain conditions. If you can evaluate whether a given value meets the criteria efficiently, you can use binary search over this "answer space."
- **Example:** Finding the minimum length of a rope that can tie all given knots, or the maximum size of a subarray meeting certain conditions.

3. Monotonic Functions or Decisions:

- Binary search is effective when the problem involves a function or decision that is monotonic, i.e., increasing or decreasing. This means if a certain value meets the criteria, all larger (or smaller) values will also meet it (or vice versa).
- **Example:** Determining the maximum weight that can be carried without breaking, finding the minimum speed required to reach a destination on time.

4. Optimization Problems:

- Problems involving minimizing or maximizing a parameter, especially under constraints, can often be solved using binary search.
- **Example:** Minimizing the maximum difference between two subsets, or optimizing the cost under given conditions.

5. Unimodal Functions:

- Binary search can be adapted to find the maximum or minimum of unimodal functions (functions that increase and then decrease, or vice versa) using techniques like ternary search.
- **Example:** Finding the peak element in a bitonic array, where elements first increase and then decrease.

6. Special Structures:

- If the problem involves a data structure like a balanced tree or other pre-ordered data, binary search might be a natural fit.
- **Example:** Binary search on a segment tree for range queries, or finding the k-th smallest element using binary search on the frequency table.

7. Implicit Binary Search:

- Sometimes, the array is not explicitly given, but you can compute values on the fly, treating it as if it were sorted.
- **Example:** Finding the smallest distance between two points in a geometric problem where the distances can be checked using some function.

Recognizing Patterns:

To quickly decide if binary search is applicable, consider these patterns:

- **Is there a sorted property, either explicitly or implicitly?**
- **Can you define a condition that is true for all values on one side of a midpoint and false for the other?**
- **Does the problem ask to find an extreme value (maximum/minimum) or an element's position under constraints?**

Example Problems:

1. **Classic Problems:** Searching in a sorted array, finding square roots, and finding an element in a rotated sorted array.
2. **Advanced:** Minimizing maximum page allocation to students, finding the k-th smallest element in a sorted matrix, or binary searching for the optimal meeting point in a coordinate system.

By honing these instincts and practicing problems that involve binary search, you'll become better at identifying suitable scenarios for its application in competitive programming.



give code example in cpp for every examples you have provided me. additionally, first describe how you have decided that the problem should be solved using binary search then give the solution code. add some more examples along with your provided examples.



1. Classic Search in a Sorted Array

Problem:

Given a sorted array, find the index of a specific target element. Return `-1` if the target is not found.

Why Binary Search:

The array is sorted, which means we can leverage the order to reduce the search space by half each time. This is the textbook use case for binary search.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
}
```

```

    return -1; // Target not found
}

int main() {
    vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int target = 5;
    int result = binarySearch(arr, target);
    cout << "Index of " << target << " is: " << result << endl;
    return 0;
}

```

2. Finding the First or Last Occurrence of an Element

Problem:

Given a sorted array, find the first occurrence of a target element.

Why Binary Search:

The array is sorted, and we need to find the boundary (first occurrence), which can be done by modifying binary search to continue searching left even after finding the target.

Code:

cpp

```

#include <iostream>
#include <vector>
using namespace std;

int findFirstOccurrence(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1, result = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            result = mid;
            right = mid - 1; // Continue to search left part
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}

int main() {
    vector<int> arr = {1, 2, 2, 2, 3, 4, 5};
    int target = 2;
    int result = findFirstOccurrence(arr, target);
    cout << "First occurrence of " << target << " is at index: " << result << endl;
    return 0;
}

```

3. Finding an Element in a Rotated Sorted Array

Problem:

Given a rotated sorted array, find the index of a target element.

Why Binary Search:

The array is sorted but rotated, meaning it can be split into two sorted subarrays. Binary search can be adapted by checking which part is sorted and deciding the search direction based on the target's location.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int searchInRotatedArray(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;

        // Check if left side is sorted
        if (arr[left] <= arr[mid]) {
            if (arr[left] <= target && target < arr[mid])
                right = mid - 1;
            else
                left = mid + 1;
        } else { // Right side is sorted
            if (arr[mid] < target && target <= arr[right])
                left = mid + 1;
            else
                right = mid - 1;
        }
    }
    return -1; // Target not found
}

int main() {
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2};
    int target = 0;
    int result = searchInRotatedArray(arr, target);
    cout << "Index of " << target << " is: " << result << endl;
    return 0;
}
```

4. Finding Minimum in Rotated Sorted Array

Problem:

Find the minimum element in a rotated sorted array.

Why Binary Search:

Although the array is rotated, the smallest element will be the point of rotation. We can use binary search to locate this inflection point.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int findMinInRotatedArray(const vector<int>& arr) {
    int left = 0, right = arr.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] > arr[right])
            left = mid + 1;
        else
            right = mid;
    }
    return arr[left];
}

int main() {
    vector<int> arr = {4, 5, 6, 7, 0, 1, 2};
    int result = findMinInRotatedArray(arr);
    cout << "Minimum element is: " << result << endl;
    return 0;
}
```

5. Binary Search on the Answer Space: Minimum Speed to Reach Destination

Problem:

Given a list of distances and a maximum time, find the minimum speed needed to reach the destination within the time.

Why Binary Search:

The speed is the answer space, and we need to find the minimum value. We can check whether a given speed satisfies the condition and adjust accordingly.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

bool canReachOnTime(const vector<int>& distances, double speed, double maxTime) {
    double time = 0;
    for (int i = 0; i < distances.size(); i++) {
        time += ceil(distances[i] / speed);
    }
    return time <= maxTime;
}

int minSpeedToReach(const vector<int>& distances, double maxTime) {
    int left = 1, right = 1e7; // Upper bound based on constraints
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canReachOnTime(distances, mid, maxTime))
            right = mid;
        else
            left = mid + 1;
    }
}
```

```

    }
    return left;
}

int main() {
    vector<int> distances = {1, 3, 2};
    double maxTime = 6;
    int result = minSpeedToReach(distances, maxTime);
    cout << "Minimum speed to reach on time is: " << result << endl;
    return 0;
}

```

6. Finding the k-th Smallest Element in a Sorted Matrix

Problem:

Given a sorted matrix (each row and column is sorted), find the k-th smallest element.

Why Binary Search:

The answer lies in a sorted sequence. We can binary search on the possible values and count the number of elements less than or equal to the midpoint, adjusting the range based on this count.

Code:

cpp

```

#include <iostream>
#include <vector>
using namespace std;

int countLessEqual(const vector<vector<int>>& matrix, int mid) {
    int count = 0, n = matrix.size(), i = n - 1, j = 0;
    while (i >= 0 && j < n) {
        if (matrix[i][j] <= mid) {
            count += i + 1; // All elements in column are less than or equal
            j++;
        } else {
            i--;
        }
    }
    return count;
}

int kthSmallest(const vector<vector<int>>& matrix, int k) {
    int n = matrix.size(), left = matrix[0][0], right = matrix[n - 1][n - 1];
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (countLessEqual(matrix, mid) < k)
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 5, 9},
        {10, 11, 13},
        {12, 13, 15}
    };
    int k = 8;
    int result = kthSmallest(matrix, k);
}

```

```
cout << k << "-th smallest element is: " << result << endl;
return 0;
}
```

7. Peak Element in Unimodal Array

Problem:

Given an array that first increases and then decreases, find the peak element.

Why Binary Search:

The peak can be found by comparing midpoints, since the function changes direction at the peak. Binary search can find this peak by comparing the mid and mid+1 elements.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int findPeakElement(const vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[mid + 1]) // Go right
            left = mid + 1;
        else
            right = mid; // Go left
    }
    return left; // Peak element index
}

int main() {
    vector<int> nums = {1, 3, 8, 12, 4, 2};
    int peak = findPeakElement(nums);
    cout << "Peak element is at index: " << peak << endl;
    return 0;
}
```

Additional Example: Search Insert Position

Problem:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be inserted in order.

Why Binary Search:

The array is sorted, and we're looking for a position, making binary search ideal.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int searchInsert(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return left;
}

int main() {
    vector<int> nums = {1, 3, 5, 6};
    int target = 5;
    int result = searchInsert(nums, target);
    cout << "Insert position of " << target << " is: " << result << endl;
    return 0;
}
```

These examples illustrate the various applications of binary search in competitive programming, leveraging its efficiency in different problem settings.



give 10 more examples in the same way



8. Finding the Smallest Missing Positive Number

Problem:

Given an unsorted array of integers, find the smallest missing positive integer.

Why Binary Search:

After sorting the array, we need to find the smallest missing positive, which can be found using binary search to efficiently locate the first missing integer.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int firstMissingPositive(vector<int>& nums) {
    sort(nums.begin(), nums.end()); // Sort the array first
    int left = 0, right = nums.size() - 1;
```

```
int missing = 1; // Start with the smallest positive integer
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (nums[mid] == missing) {
        missing++;
        left = mid + 1;
    } else if (nums[mid] < missing) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
return missing;
}

int main() {
    vector<int> nums = {3, 4, -1, 1};
    int result = firstMissingPositive(nums);
    cout << "Smallest missing positive number is: " << result << endl;
    return 0;
}
```

9. Maximum Length of Subarray with Sum $\leq k$

Problem:

Given an array of integers, find the maximum length of a contiguous subarray with a sum less than or equal to a given value k .

Why Binary Search:

We can use binary search to find the longest subarray by calculating prefix sums and using binary search to find the maximum index such that the subarray sum stays within the limit.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int maxLengthSubarray(vector<int>& nums, int k) {
    int n = nums.size();
    vector<int> prefixSum(n + 1, 0);
    for (int i = 1; i <= n; i++) {
        prefixSum[i] = prefixSum[i - 1] + nums[i - 1];
    }
    int maxLength = 0;
    for (int i = 1; i <= n; i++) {
        int target = prefixSum[i] - k;
        auto it = lower_bound(prefixSum.begin(), prefixSum.begin() + i, target);
        int index = distance(prefixSum.begin(), it);
        if (prefixSum[index] <= target)
            maxLength = max(maxLength, i - index);
    }
    return maxLength;
}

int main() {
    vector<int> nums = {1, 2, 3, 4, 5};
    int k = 11;
}
```

```
int result = maxLengthSubarray(nums, k);
cout << "Maximum length of subarray with sum ≤ " << k << " is: " << result << endl;
return 0;
}
```

10. Allocate Minimum Number of Pages

Problem:

Given an array of books where each element represents the number of pages in a book, allocate the books to students so that the maximum number of pages assigned to a student is minimized.

Why Binary Search:

The solution involves binary search on the minimum maximum pages that can be assigned, adjusting based on whether a given maximum is feasible.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

bool isFeasible(const vector<int>& books, int students, int maxPages) {
    int count = 1, sum = 0;
    for (int pages : books) {
        if (sum + pages > maxPages) {
            count++;
            sum = pages;
            if (count > students) return false;
        } else {
            sum += pages;
        }
    }
    return true;
}

int allocateMinPages(const vector<int>& books, int students) {
    if (books.size() < students) return -1;
    int left = *max_element(books.begin(), books.end());
    int right = accumulate(books.begin(), books.end(), 0);
    int result = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (isFeasible(books, students, mid)) {
            result = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return result;
}

int main() {
    vector<int> books = {12, 34, 67, 90};
    int students = 2;
    int result = allocateMinPages(books, students);
    cout << "Minimum maximum pages allocated: " << result << endl;
}
```

```
    return 0;
}
```

11. Searching in a Matrix

Problem:

Given a matrix where every row and column is sorted in ascending order, find if a target exists.

Why Binary Search:

Each row and column is sorted, so binary search can be used to eliminate half the rows or columns in each step.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

bool searchMatrix(const vector<vector<int>>& matrix, int target) {
    if (matrix.empty()) return false;
    int rows = matrix.size();
    int cols = matrix[0].size();
    int left = 0, right = rows * cols - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int midValue = matrix[mid / cols][mid % cols];
        if (midValue == target) return true;
        else if (midValue < target) left = mid + 1;
        else right = mid - 1;
    }
    return false;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 3, 5, 7},
        {10, 11, 16, 20},
        {23, 30, 34, 60}
    };
    int target = 3;
    bool result = searchMatrix(matrix, target);
    cout << "Element " << target << " found: " << boolalpha << result << endl;
    return 0;
}
```

12. Median of Two Sorted Arrays

Problem:

Given two sorted arrays, find the median of the combined sorted array.

Why Binary Search:

We can use binary search to partition the two arrays in such a way that all elements on the left side are less than those on the right, thus finding the median.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    if (nums1.size() > nums2.size()) return findMedianSortedArrays(nums2, nums1);
    int x = nums1.size(), y = nums2.size();
    int low = 0, high = x;

    while (low <= high) {
        int partitionX = (low + high) / 2;
        int partitionY = (x + y + 1) / 2 - partitionX;

        int maxX = (partitionX == 0) ? INT_MIN : nums1[partitionX - 1];
        int minX = (partitionX == x) ? INT_MAX : nums1[partitionX];

        int maxY = (partitionY == 0) ? INT_MIN : nums2[partitionY - 1];
        int minY = (partitionY == y) ? INT_MAX : nums2[partitionY];

        if (maxX <= minY && maxY <= minX) {
            if ((x + y) % 2 == 0)
                return (double)(max(maxX, maxY) + min(minX, minY)) / 2;
            else
                return (double)max(maxX, maxY);
        } else if (maxX > minY) {
            high = partitionX - 1;
        } else {
            low = partitionX + 1;
        }
    }
    return -1; // Should never reach here
}

int main() {
    vector<int> nums1 = {1, 3};
    vector<int> nums2 = {2};
    double result = findMedianSortedArrays(nums1, nums2);
    cout << "Median of two sorted arrays is: " << result << endl;
    return 0;
}
```

13. Minimize the Maximum Distance to Gas Station

Problem:

Given positions of gas stations and additional gas stations to be built, minimize the maximum distance between adjacent gas stations.

Why Binary Search:

Binary search on the distance, checking feasibility by simulating the placement of new stations, helps find the minimal maximum distance.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool canPlaceStations(const vector<int>& stations, int K, double dist) {
    int count = 0;
    for (int i = 1; i < stations.size(); i++) {
        count += (int)((stations[i] - stations[i - 1]) / dist);
    }
    return count <= K;
}

double minimizeMaxDistance(vector<int>& stations, int K) {
    double left = 0.0, right = 1e9, result = 0;
    while (right - left > 1e-6) {
        double mid = left + (right - left) / 2.0;
        if (canPlaceStations(stations, K, mid)) {
            result = mid;
            right = mid;
        } else {
            left = mid;
        }
    }
    return result;
}

int main() {
    vector<int> stations = {0, 2, 4, 7, 10};
    int K = 3;
    double result = minimizeMaxDistance(stations, K);
    cout << "Minimum maximum distance is: " << result << endl;
    return 0;
}
```

14. Finding Minimum in Rotated Sorted Array

Problem:

Given a rotated sorted array, find the minimum element.

Why Binary Search:

Since the array is rotated, using binary search on the rotated structure helps find the inflection point where the minimum lies.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int findMin(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[right]) left = mid + 1;
        else right = mid;
    }
    return nums[left];
}
```

```

    }
    return nums[left];
}

int main() {
    vector<int> nums = {4, 5, 6, 7, 0, 1, 2};
    int result = findMin(nums);
    cout << "Minimum element is: " << result << endl;
    return 0;
}

```

15. Kth Smallest Element in a Sorted Matrix

Problem:

Given an $n \times n$ matrix where each row and column is sorted, find the k -th smallest element.

Why Binary Search:

Use binary search on the range of elements, counting how many elements are smaller or equal to the mid-point.

Code:

cpp

```

#include <iostream>
#include <vector>
using namespace std;

int countLessEqual(vector<vector<int>>& matrix, int target) {
    int count = 0, n = matrix.size();
    int row = n - 1, col = 0;
    while (row >= 0 && col < n) {
        if (matrix[row][col] <= target) {
            count += row + 1;
            col++;
        } else {
            row--;
        }
    }
    return count;
}

int kthSmallest(vector<vector<int>>& matrix, int k) {
    int left = matrix[0][0], right = matrix.back().back();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (countLessEqual(matrix, mid) < k) left = mid + 1;
        else right = mid;
    }
    return left;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 5, 9},
        {10, 11, 13},
        {12, 13, 15}
    };
    int k = 8;
    int result = kthSmallest(matrix, k);
    cout << "The " << k << "-th smallest element is: " << result << endl;
}

```

```
    return 0;
}
```

16. Split Array Largest Sum

Problem:

Given an array, split it into m subarrays so that the largest sum of any subarray is minimized.

Why Binary Search:

Binary search on the range of possible largest sums allows us to find the minimum maximum sum efficiently.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

bool canSplit(vector<int>& nums, int m, int maxSum) {
    int currentSum = 0, count = 1;
    for (int num : nums) {
        if (currentSum + num > maxSum) {
            count++;
            currentSum = num;
            if (count > m) return false;
        } else {
            currentSum += num;
        }
    }
    return true;
}

int splitArray(vector<int>& nums, int m) {
    int left = *max_element(nums.begin(), nums.end());
    int right = accumulate(nums.begin(), nums.end(), 0);
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canSplit(nums, m, mid)) right = mid;
        else left = mid + 1;
    }
    return left;
}

int main() {
    vector<int> nums = {7, 2, 5, 10, 8};
    int m = 2;
    int result = splitArray(nums, m);
    cout << "Minimum largest sum: " << result << endl;
    return 0;
}
```

17. Longest Substring with At Most K Distinct Characters

Problem:

Given a string, find the length of the longest substring with at most K distinct characters.

Why Binary Search:

Using binary search on the length, check whether a substring of that length exists with at most K distinct characters.

Code:

cpp

```
#include <iostream>
#include <unordered_map>
using namespace std;

bool isValid(const string& s, int k, int length) {
    unordered_map<char, int> freq;
    int distinct = 0;
    for (int i = 0; i < length; i++) {
        if (freq[s[i]]++ == 0) distinct++;
    }
    if (distinct <= k) return true;
    for (int i = length; i < s.size(); i++) {
        if (--freq[s[i - length]] == 0) distinct--;
        if (freq[s[i]]++ == 0) distinct++;
        if (distinct <= k) return true;
    }
    return false;
}

int lengthOfLongestSubstringKDistinct(string s, int k) {
    int left = 0, right = s.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (isValid(s, k, mid)) left = mid + 1;
        else right = mid;
    }
    return right - 1;
}

int main() {
    string s = "eceba";
    int k = 2;
    int result = lengthOfLongestSubstringKDistinct(s, k);
    cout << "Length of longest substring with at most " << k << " distinct characters: "
    << result << endl;
    return 0;
}
```

These 10 additional examples show various contexts where binary search can be effectively applied to solve problems in competitive programming.



now give 10 more classic and advanced examples



18. Aggressive Cows

Problem:

Given positions of stalls and a number of cows, place the cows in the stalls such that the minimum distance between any two cows is maximized.

Why Binary Search:

The solution involves binary search on the minimum possible distance between cows. We adjust the distance based on whether it's feasible to place all cows with at least that distance apart.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool canPlaceCows(vector<int>& stalls, int cows, int minDist) {
    int count = 1, lastPosition = stalls[0];
    for (int i = 1; i < stalls.size(); i++) {
        if (stalls[i] - lastPosition >= minDist) {
            count++;
            lastPosition = stalls[i];
            if (count == cows) return true;
        }
    }
    return false;
}

int maxMinDistance(vector<int>& stalls, int cows) {
    sort(stalls.begin(), stalls.end());
    int left = 1, right = stalls.back() - stalls[0], result = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (canPlaceCows(stalls, cows, mid)) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}

int main() {
    vector<int> stalls = {1, 2, 4, 8, 9};
    int cows = 3;
    int result = maxMinDistance(stalls, cows);
    cout << "Maximum minimum distance is: " << result << endl;
    return 0;
}
```

19. Find Smallest Divisor Given a Threshold

Problem:

Given an array of integers and a threshold, find the smallest divisor such that the sum of each element divided by this divisor is less than or equal to the threshold.

Why Binary Search:

We binary search on the divisor values, checking if they meet the threshold condition.

Code:

cpp

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int sumByDivisor(vector<int>& nums, int divisor) {
    int sum = 0;
    for (int num : nums) {
        sum += (num + divisor - 1) / divisor; // Equivalent to ceil(num / divisor)
    }
    return sum;
}

int smallestDivisor(vector<int>& nums, int threshold) {
    int left = 1, right = *max_element(nums.begin(), nums.end());
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sumByDivisor(nums, mid) > threshold) left = mid + 1;
        else right = mid;
    }
    return left;
}

int main() {
    vector<int> nums = {1, 2, 5, 9};
    int threshold = 6;
    int result = smallestDivisor(nums, threshold);
    cout << "Smallest divisor: " << result << endl;
    return 0;
}
```

20. Split Array into Minimum Number of Subarrays with Target Sum

Problem:

Given an array and a target sum, split the array into the minimum number of non-empty subarrays with each subarray sum equal to the target.

Why Binary Search:

Use binary search to find the smallest value that can be split into subarrays of sum equal to the target.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

bool canSplit(vector<int>& nums, int maxSubarrays, int target) {
```

```

int currentSum = 0, count = 1;
for (int num : nums) {
    if (currentSum + num > target) {
        count++;
        currentSum = num;
        if (count > maxSubarrays) return false;
    } else {
        currentSum += num;
    }
}
return true;
}

int splitArrayMinSubarrays(vector<int>& nums, int target) {
    int left = *max_element(nums.begin(), nums.end());
    int right = accumulate(nums.begin(), nums.end(), 0);
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canSplit(nums, target, mid)) right = mid;
        else left = mid + 1;
    }
    return left;
}

int main() {
    vector<int> nums = {7, 2, 5, 10, 8};
    int target = 15;
    int result = splitArrayMinSubarrays(nums, target);
    cout << "Minimum number of subarrays with target sum: " << result << endl;
    return 0;
}

```

21. Cutting a Chocolate Bar

Problem:

Given a chocolate bar of size $n \times m$ and a maximum number of vertical and horizontal cuts, find the maximum size of a piece.

Why Binary Search:

We binary search on the size of the pieces, using a check function to see if we can make the cuts.

Code:

cpp

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool canMakeCuts(vector<int>& cuts, int maxCuts, int minSize) {
    int cutsMade = 0, lastCut = 0;
    for (int i = 1; i < cuts.size(); i++) {
        if (cuts[i] - lastCut >= minSize) {
            cutsMade++;
            lastCut = cuts[i];
        }
    }
    return cutsMade >= maxCuts;
}

int maxChocolatePiece(vector<int>& verticalCuts, vector<int>& horizontalCuts, int m, int

```

```
n, int maxVCuts, int maxHCuts) {
    sort(verticalCuts.begin(), verticalCuts.end());
    sort(horizontalCuts.begin(), horizontalCuts.end());
    int left = 1, right = max(verticalCuts.back(), horizontalCuts.back());
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canMakeCuts(verticalCuts, maxVCuts, mid) && canMakeCuts(horizontalCuts,
maxHCuts, mid)) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left - 1;
}

int main() {
    vector<int> verticalCuts = {1, 3, 4, 5};
    vector<int> horizontalCuts = {1, 2, 3};
    int m = 5, n = 5, maxVCuts = 2, maxHCuts = 1;
    int result = maxChocolatePiece(verticalCuts, horizontalCuts, m, n, maxVCuts,
maxHCuts);
    cout << "Maximum piece size: " << result << endl;
    return 0;
}
```

22. Matrix Median

Problem:

Find the median in a matrix where each row is sorted.

Why Binary Search:

Binary search on the median value, counting how many elements are less than or equal to it.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int countLessEqual(vector<vector<int>>& matrix, int mid) {
    int count = 0;
    for (auto& row : matrix) {
        count += upper_bound(row.begin(), row.end(), mid) - row.begin();
    }
    return count;
}

int matrixMedian(vector<vector<int>>& matrix) {
    int left = matrix[0][0], right = matrix[0][0];
    for (auto& row : matrix) {
        left = min(left, row[0]);
        right = max(right, row.back());
    }
    int n = matrix.size(), m = matrix[0].size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (countLessEqual(matrix, mid) < (n * m + 1) / 2) left = mid + 1;
        else right = mid;
    }
    return left;
}
```

```

}

int main() {
    vector<vector<int>> matrix = {
        {1, 3, 5},
        {2, 6, 9},
        {3, 6, 9}
    };
    int result = matrixMedian(matrix);
    cout << "Median of the matrix is: " << result << endl;
    return 0;
}

```

23. Minimum Number of Days to Make m Bouquets

Problem:

Given an array of bloom days and a number of bouquets and flowers per bouquet, find the minimum number of days to have m bouquets.

Why Binary Search:

Binary search on the number of days, checking if it's possible to make the required number of bouquets.

Code:

cpp

```

#include <iostream>
#include <vector>
using namespace std;

bool canMakeBouquets(vector<int>& bloomDays, int days, int m, int k) {
    int count = 0, flowers = 0;
    for (int day : bloomDays) {
        if (day <= days) {
            flowers++;
            if (flowers == k) {
                count++;
                flowers = 0;
            }
        } else {
            flowers = 0;
        }
    }
    return count >= m;
}

int minDaysBouquets(vector<int>& bloomDays, int m, int k) {
    if (m * k > bloomDays.size()) return -1;
    int left = *min_element(bloomDays.begin(), bloomDays.end());
    int right = *max_element(bloomDays.begin(), bloomDays.end());
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canMakeBouquets(bloomDays, mid, m, k)) right = mid;
        else left = mid + 1;
    }
    return left;
}

int main() {
    vector<int> bloomDays = {1, 10, 3, 10, 2};
    int m = 3, k = 1;
}

```

```
int result = minDaysBouquets(bloomDays, m, k);
cout << "Minimum number of days: " << result << endl;
return 0;
}
```

24. Find Peak Element

Problem:

Given an array, find a peak element (an element greater than its neighbors).

Why Binary Search:

We can reduce the search space by choosing the side where a peak must exist.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int findPeakElement(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[mid + 1]) right = mid;
        else left = mid + 1;
    }
    return left;
}

int main() {
    vector<int> nums = {1, 2, 1, 3, 5, 6, 4};
    int result = findPeakElement(nums);
    cout << "Peak element index is: " << result << endl;
    return 0;
}
```

25. H-Index II

Problem:

Given a sorted array of citations, find the H-index.

Why Binary Search:

Binary search on the index values to find where citations are equal to or greater than the H-index value.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

int hIndex(vector<int>& citations) {
    int left = 0, right = citations.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (citations[mid] >= citations.size() - mid) right = mid;
        else left = mid + 1;
    }
    return citations.size() - left;
}

int main() {
    vector<int> citations = {0, 1, 3, 5, 6};
    int result = hIndex(citations);
    cout << "H-index is: " << result << endl;
    return 0;
}
```

26. Divide Chocolate

Problem:

Divide a chocolate bar into k pieces such that the minimum sweetness of any piece is maximized.

Why Binary Search:

Binary search on the minimum sweetness value, checking if it can be achieved.

Code:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

bool canDivide(vector<int>& sweetness, int minSweetness, int k) {
    int currentSweetness = 0, pieces = 0;
    for (int s : sweetness) {
        currentSweetness += s;
        if (currentSweetness >= minSweetness) {
            pieces++;
            currentSweetness = 0;
        }
    }
    return pieces >= k;
}

int maximizeSweetness(vector<int>& sweetness, int k) {
    int left = *min_element(sweetness.begin(), sweetness.end());
    int right = accumulate(sweetness.begin(), sweetness.end(), 0);
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (canDivide(sweetness, mid, k)) left = mid + 1;
        else right = mid;
    }
    return left - 1;
}
```



```
int main() {
    vector<int> sweetness = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int k = 5;
    int result = maximizeSweetness(sweetness, k);
    cout << "Maximum minimum sweetness: " << result << endl;
    return 0;
}
```

27. Maximize the Minimum Distance Between Gas Stations

Problem:

Given positions of gas stations and k more to add, find the maximum minimum distance between adjacent stations.

Why Binary Search:

Binary search on the minimum distance, checking how many new stations are needed to maintain that distance.

Code:

cpp


```
#include <iostream>
#include <vector>
using namespace std;

bool canPlaceStations(vector<int>& stations, int k, double minDist) {
    int count = 0;
    for (int i = 1; i < stations.size(); i++) {
        count += (stations[i] - stations[i - 1]) / minDist;
    }
    return count <= k;
}

double maximizeMinDistance(vector<int>& stations, int k) {
    double left = 0, right = stations.back() - stations[0];
    while (right - left > 1e-6) {
        double mid = left + (right - left) / 2;
        if (canPlaceStations(stations, k, mid)) left = mid;
        else right = mid;
    }
    return left;
}

int main() {
    vector<int> stations = {0, 4, 8, 12};
    int k = 3;
    double result = maximizeMinDistance(stations, k);
    cout << "Maximum minimum distance: " << result << endl;
    return 0;
}
```

These additional classic and advanced examples demonstrate the versatility of binary search in competitive programming across a wide variety of problem types.

 Messages beyond this point are only visible to you

ChatGPT can make mistakes. Check important info.