**URL:** [**https://leetcode.com/discuss/general-discussion/1089533/An-approach-to-writing-bug-free-Binary-Search-code**](https://leetcode.com/discuss/general-discussion/1089533/An-approach-to-writing-bug-free-Binary-Search-code)

**Pre-requisites:**  
An idea about what binary search is, as this post is focused on how to write bug-free code for this.

**Problem statement:**  
There are multiple variants of binary search, I will cover these three:

1. Search the target element in a sorted array
2. Search the smallest element in a sorted array greater than the target element
3. Search the largest element in a sorted array lesser than the target element

**Format of this post:**  
In this post,

* First, I will write standard code I have seen for these problems
* Then, I will mention what problems I faced while using these standard codes and how I fell into bug traps,
* Lastly, I will provide an approach how I write code such that I don't think twice about these general bugs.
* This post will be finished with a few examples/applications.

**Standard codes:**  
The function will return the index as the answer instead of element, and -1 when there is no answer.

**1. Search the target element in a sorted array**

Standard code :

int search(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1;

while(lo <= hi) { // observe "<=" operator here

int mid = lo + (hi-lo)/2;

if(arr[mid] == target) return mid;

if(arr[mid] < target) lo = mid+1;

else hi = mid-1;

}

return -1;

}

You may notice that I choose mid = lo + (hi-lo)/2 instead of mid = (lo + hi) / 2. That is because the latter one is prone to overflow. Consider the case when lo = 10 and hi = INT\_MAX and check the output for both the expressions. 😊

**2. Search the smallest element in a sorted array greater than the target element**

A standard way to write this would be:

int greaterThan(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1;

while(lo < hi) { // observe this time it is "<" operator, not "<="

int mid = lo + (hi-lo)/2;

if(arr[mid] <= target) lo = mid+1;

else hi = mid;

}

return arr[lo] > target ? lo : -1;

}

**3. Search the largest element in a sorted array lesser than the target element**  
A standard way for this would be:

int smallerThan(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1;

while(lo < hi) {

int mid = lo + (hi-lo+1)/2; // Note that I used ceil here otherwise it could go in infinite loop

if(arr[mid] < target) lo = mid;

else hi = mid-1;

}

return arr[lo] < target ? lo : -1;

}

There are few other implementations to solve the problem of infinite loop but using ceil looked cleaner way to me.

**Introduction of bugs:**  
As you can see, while writing Binary search (or its variants), you need to

* think about the while loop condition,
* infinite loop case,
* choose between lo = mid vs lo = mid + 1,
* choose between hi = mid vs hi = mid - 1

These are the main culprits to introduce bugs that I have experienced.

**A bug-free approach:**  
My approach to writing Binary-search related code is to:

1. **Always** keep an ans variable to store/update the answer. You have to initialize it correctly though. E.g. if you want to maximize something, start with the minimum that can be the answer (we will see this in the end of this post while solving a problem). In the end, we would be returning this ans variable *without thinking twice*.
2. **Always** make while loop condition to be while(lo <= hi), note the <= sign, and we will either make lo = mid+1 or hi = mid-1 (no more choosing here 😊), which would **always** avoid infinite loop condition, which is not the case when we do lo = mid or hi = mid and that introduces the infinite loop bugs.

That creates a pretty determinstic loop for us and we can play inside it.

**1. Search the target element in a sorted array**

int search(vector<int> arr, int target) {

int lo = 0, hi = arr.size(), ans = -1;

while(lo <= hi) {

int mid = lo + (hi-lo)/2;

if(arr[mid] == target) {

ans = mid;

break;

} else if(arr[mid] < target) lo = mid+1;

else hi = mid-1;

}

return ans;

}

**2. Search the smallest element in a sorted array greater than the target element**

int greaterThan(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1, ans = -1;

while(lo <= hi) {

int mid = lo + (hi-lo)/2;

if(arr[mid] > target) {

ans = mid; // So far this could be the answer

hi = mid-1; // But we can find better answer in left

} else {

lo = mid+1; // Try right

}

}

return ans;

}

**3. Search the largest element in a sorted array lesser than the target element**

int smallerThan(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1, ans = -1;

while(lo <= hi) {

int mid = lo + (hi-lo)/2; // Note that we don't have to think twice whether we should use ceil here or not, this will always work

if(arr[mid] < target) {

ans = mid; // So far this could be the answer

lo = mid+1; // But we can find better answer in right

} else {

hi = mid-1; // Try left

}

}

return ans;

}

If you notice,

* Last two codes are almost similar,
* We didn't have to think about infinite loops,
* We didn't have to think about while loop condition,
* All we have to think about is the if-condition, and that would always depend on the problem statement.

**Applications/Examples:**  
**1. Searching first and last occurrence of the target element in a sorted array**  
To find the first occurrence, whenever we find the element, we update our answer, and search for better answer in left (why?)  
Similarly for last occurrence, we update our answer whenever we find the element, and then search in right.

int firstOccurrence(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1, ans = -1; // these should always be initialized carefully

while(lo <= hi) { // this will never change

int mid = lo + (hi-lo)/2; // this will never change

if(arr[mid] == target) {

ans = mid; // Update the answer

hi = mid - 1; // Search in left for better answer

} else if(arr[mid] < target) lo = mid + 1;

else hi = mid - 1;

}

return ans; // this will never change

}

int lastOccurrence(vector<int> arr, int target) {

int lo = 0, hi = arr.size() - 1, ans = -1;

while(lo <= hi) {

int mid = lo + (hi-lo)/2;

if(arr[mid] == target) {

ans = mid; // Update the answer

lo = mid + 1; // Search in right for better answer

} else if(arr[mid] < target) lo = mid + 1;

else hi = mid - 1;

}

return ans;

}

**2. Binary search over monotonic functions:**  
There are many problems which can be solved using binary search, not on some array, but on some monotonic function (a function that is either increasing or decreasing e.g. f(x) = x+1, as we increase x, the value of f(x) will always increase).

Let's solve [1011. Capacity To Ship Packages Within D Days](https://leetcode.com/problems/capacity-to-ship-packages-within-d-days/) to understand:

We can observe that if we keep the capacity very high, we can always ship, and with very low capacity, we can't ship, so we need to find the lowest possible ans such that it is shippable for capacity >= ans

int shipWithinDays(vector<int>& weights, int D) {

// Lets first define our feasible range

int lo = weights[0];

for(int w : weights) lo = max(lo, w); // we would at least need the capacity such that the largest weight can be shipped, so that would be lower bound

int hi = 0;

for(int w : weights) hi += w; // we would not need the capacity more than sum of all weights, so that would be upper bound

int ans = hi; // since we need to minimize the capacity, start with hi, later we will update/minimize it

while(lo <= hi) { // No need to think here :D

int mid = lo + (hi-lo)/2;

if(isPossible(weights, mid, D)) {

ans = mid; // Update answer

hi = mid-1; // Try smaller capacities

} else {

lo = mid+1;

}

}

return ans;

}

// Code to check whether it is possible to ship with this capacity

bool isPossible(vector<int>& weights, int capacity, int D) {

int cur = 0;

for(int w : weights) {

if(cur + w <= capacity) {

cur += w;

} else {

D--;

if(D == 0) return false;

cur = w;

}

}

return true;

}

**UPDATE**: Now that we have avoided infinite loop bug in binary search, there is one more caveat I have experienced. That is when we initialize lo, hi, and ans incorrectly. We can't initialize lo to 0 and hi to INT\_MAX blindly. So we have to be careful there.

Thanks for reaching to the end, hope this post was worth your time :)