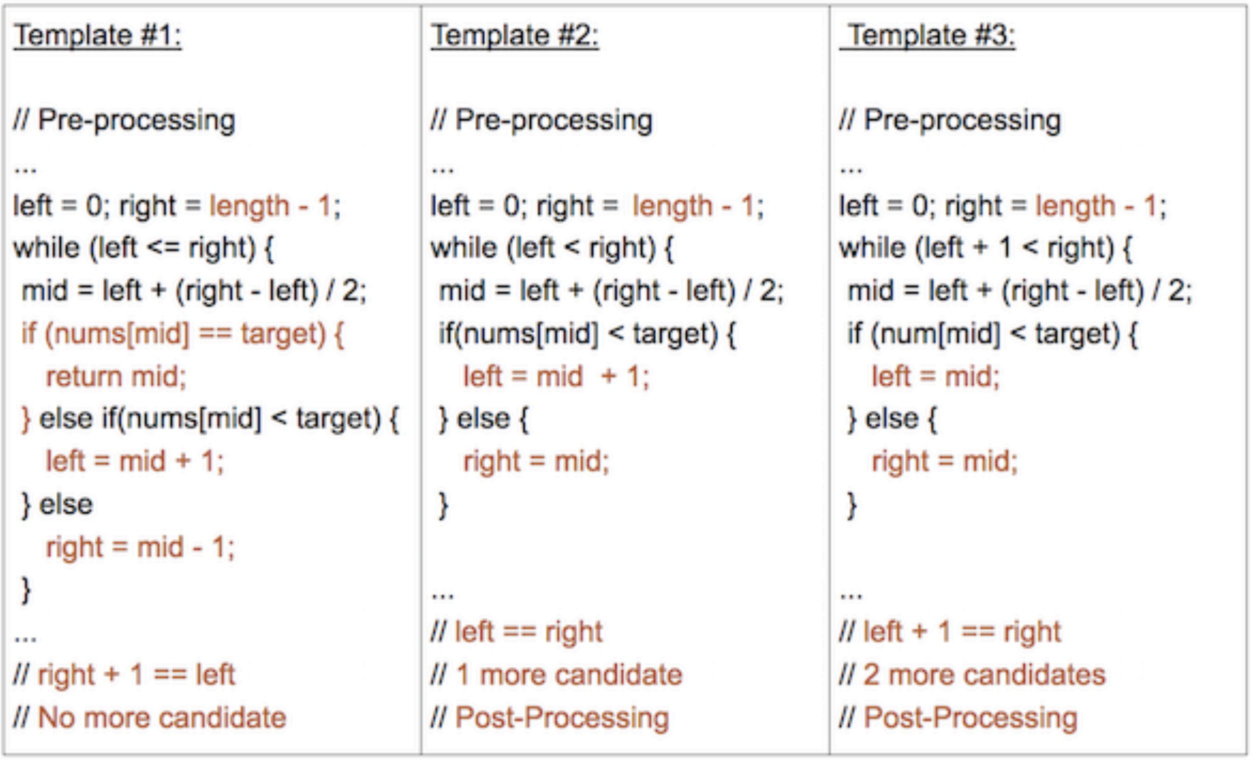
Explore

Template Explanation:

99% of binary search problems that you see online will fall into 1 of these 3 templates. Some problems can be implemented using multiple templates, but as you practice more, you will notice that some templates are more suited for certain problems than others.

Note: The templates and their differences have been colored coded below.



Template #1:	Template #2:	Template #3:
// Pre-processing	// Pre-processing	// Pre-processing
...	...	...
left = 0; right = length - 1;	left = 0; right = length - 1;	left = 0; right = length - 1;
while (left <= right) {	while (left < right) {	while (left + 1 < right) {
mid = left + (right - left) / 2;	mid = left + (right - left) / 2;	mid = left + (right - left) / 2;
if (nums[mid] == target) {	if(nums[mid] < target) {	if (num[mid] < target) {
return mid;	left = mid + 1;	left = mid;
} else if(nums[mid] < target) {	} else {	} else {
left = mid + 1;	right = mid;	right = mid;
} else	}	}
right = mid - 1;		
}	...	...
...	// left == right	// left + 1 == right
// right + 1 == left	// 1 more candidate	// 2 more candidates
// No more candidate	// Post-Processing	// Post-Processing

These 3 templates differ by their:

- left, mid, right index assignments
- loop or recursive termination condition
- necessity of post-processing

Templates 1 and 3 are the most commonly used and almost all binary search problems can be easily implemented in one of them. Template 2 is a bit more advanced and used for certain types of problems.

Each of these 3 provided templates provides a specific use case:

Template #1 `(left <= right)`:

- Most basic and elementary form of Binary Search
- Search Condition can be determined without comparing to the element's neighbors (or use specific elements around it)
- No post-processing required because at each step, you are checking to see if the element has been found. If you reach the end, then you know the element is not found

Template #2 `(left < right)`:

- An advanced way to implement Binary Search.
- Search Condition needs to access the element's immediate right neighbor
- Use the element's right neighbor to determine if the condition is met and decide whether to go left or right
- Guarantees Search Space is at least 2 in size at each step
- Post-processing required. Loop/Recursion ends when you have 1 element left. Need to assess if the remaining element meets the condition.

Template #3 `(left + 1 < right)`:

- An alternative way to implement Binary Search
- Search Condition needs to access element's immediate left and right neighbors
- Use element's neighbors to determine if the condition is met and decide whether to go left or right
- Guarantees Search Space is at least 3 in size at each step
- Post-processing required. Loop/Recursion ends when you have 2 elements left. Need to assess if the remaining elements meet the condition.

Time and Space Complexity:

Runtime: `O(log n)` -- Logarithmic Time

Because Binary Search operates by applying a condition to the value in the middle of our search space and thus cutting the search space in half, in the worse case, we will have to make O(log n) comparisons, where n is the number of elements in our collection.

> Why `log n`?
>
> - Binary search is performed by dividing the existing array in half.
> - So every time you a call the subroutine (or complete one iteration) the size reduced to half of the existing part.
> - First `N` become `N/2`, then it become `N/4` and go on till it find the element or size become 1.
> - The maximum no of iterations is `log N` (base 2).

Space: `O(1)` -- Constant Space

Although Binary Search does require keeping track of 3 indices, the iterative solution does not typically require any other additional space and can be applied directly to the collection itself, therefore warrants `O(1)` or constant space.

Other Types of Binary Search:

Below, we have provided another type of Binary Search for practice.

Binary Search With 2 Arrays -- In this problem, we need to compare values from 2 arrays to determine our search space: LC #4: Median of Two Sorted Arrays