Binary Search - Recursive Implementation

This chapter details the reasoning behind binary search's time complexity

Binary search is a well-known search algorithm that works on an already-sorted input. The basic idea is to successively eliminate half of the remaining search space at each step till either the target is found or there's no more search space left.

Binary search is recursive in nature but can also be implemented iteratively. Below is the recursive implementation of binary search.

Binary Search Implementation

```java
class Demonstration {
    public static void main( S
        int[] input = new int[
        System.out.println(bin
        System.out.println(bin
        System.out.println(bin
        System.out.println(bin
    }

    /**
     * start and end are inclu
     *
     * @param start
     * @param end
     * @param target
     * @param input
     * @return
     */
    public static boolean bina

        if (start > end) {
            return false;
        }

        int mid = (start + end

        if (input[mid] == targ
            return true;
        } else if (input[mid]
            return binarySearc
        } else {
```

Binary Search Recursive

Recurrence Equation

The recurrence equation for binary search is given below:

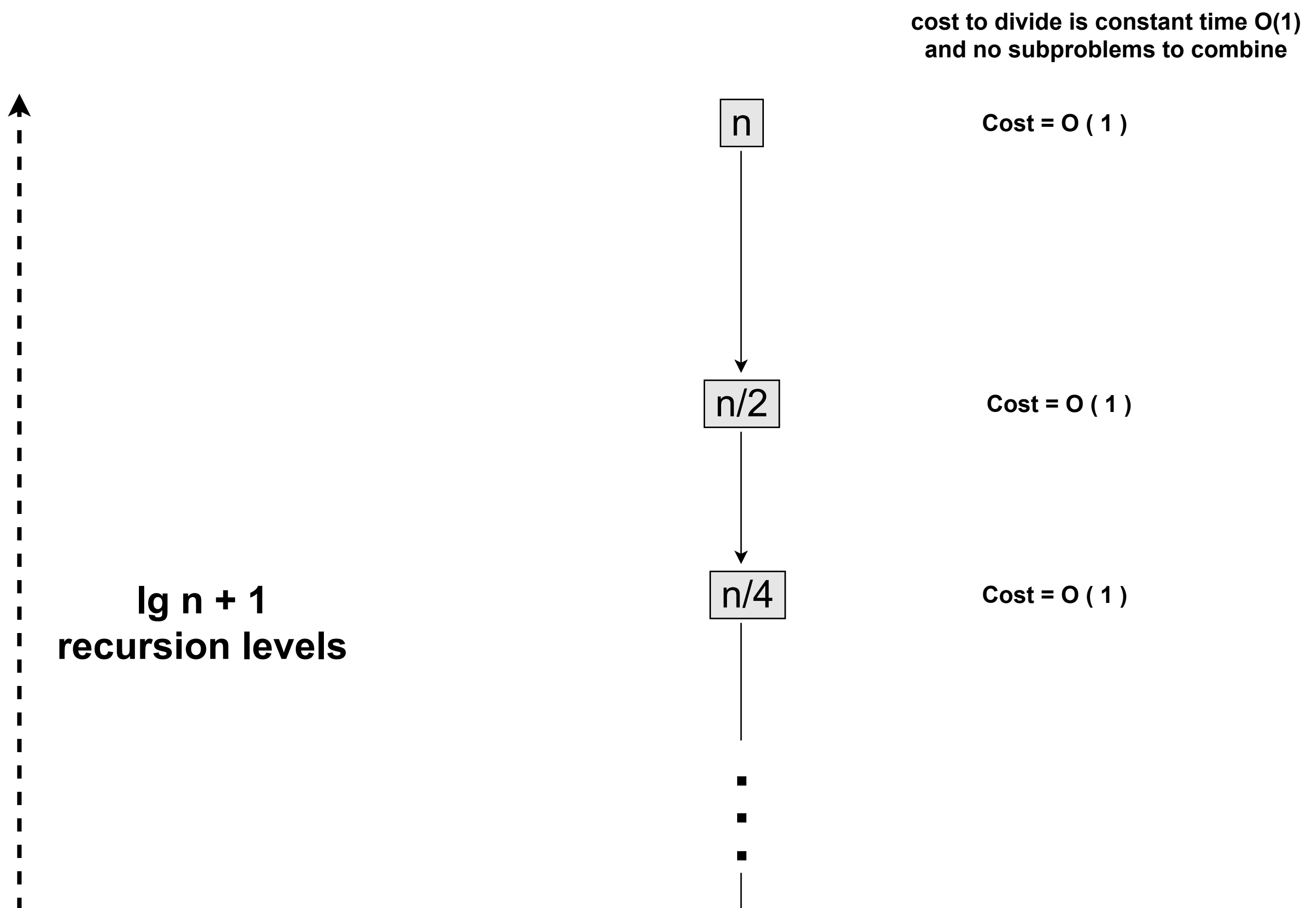
$$Running\ Time = Cost\ to\ divide\ into\ p\ subproblems + p * Cost\ to\ solve\ each\ of\ the\ p\ problems + Cost\ to\ merge\ all\ p\ problems$$

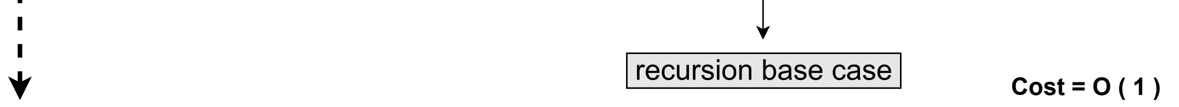
$$T(n) = O(1) + 2 * T\frac{n}{2} + O(1)$$

$$T(n) = T(\frac{n}{2})$$

The running time of binary search is the solution to the above recurrence. Note that we divide the original problem into 2 subproblems. Division is just an arithmetic operation that takes constant time and has no relation to the size of the input. We don't have to combine the solution of the subproblems, as we eliminate one half of the search space at each recursion.

The recursion tree appears below.





Analyzing Cost of Merge Sort

Note that the cost at each recursion level is constant, as all we are doing at each level is a bunch of arithmetic operations, and there are no subproblems to combine. The running time in the worst case occurs when we are unable to find the target value in the sorted array. The running time is then simply the number of recursion levels multiplied by the recursion cost at each level.

$$T(n) = Cost\ at\ each\ recursion\ level * total\ recursion\ levels$$

We already know that the cost at each recursion level is constant or O (1). We need to determine how many recursion levels exist in the worst case. This is akin to asking, "how many times do we divide the input array of size *n* by 2 to get 1?" or, "*2 raised to what power would yield n?*". Assume that the array size is a power of 2 for simplicity.

This brings us back to logarithms. As explained in the previous lesson, the number of recursion levels for an array of size *n* when successively halved will equal:

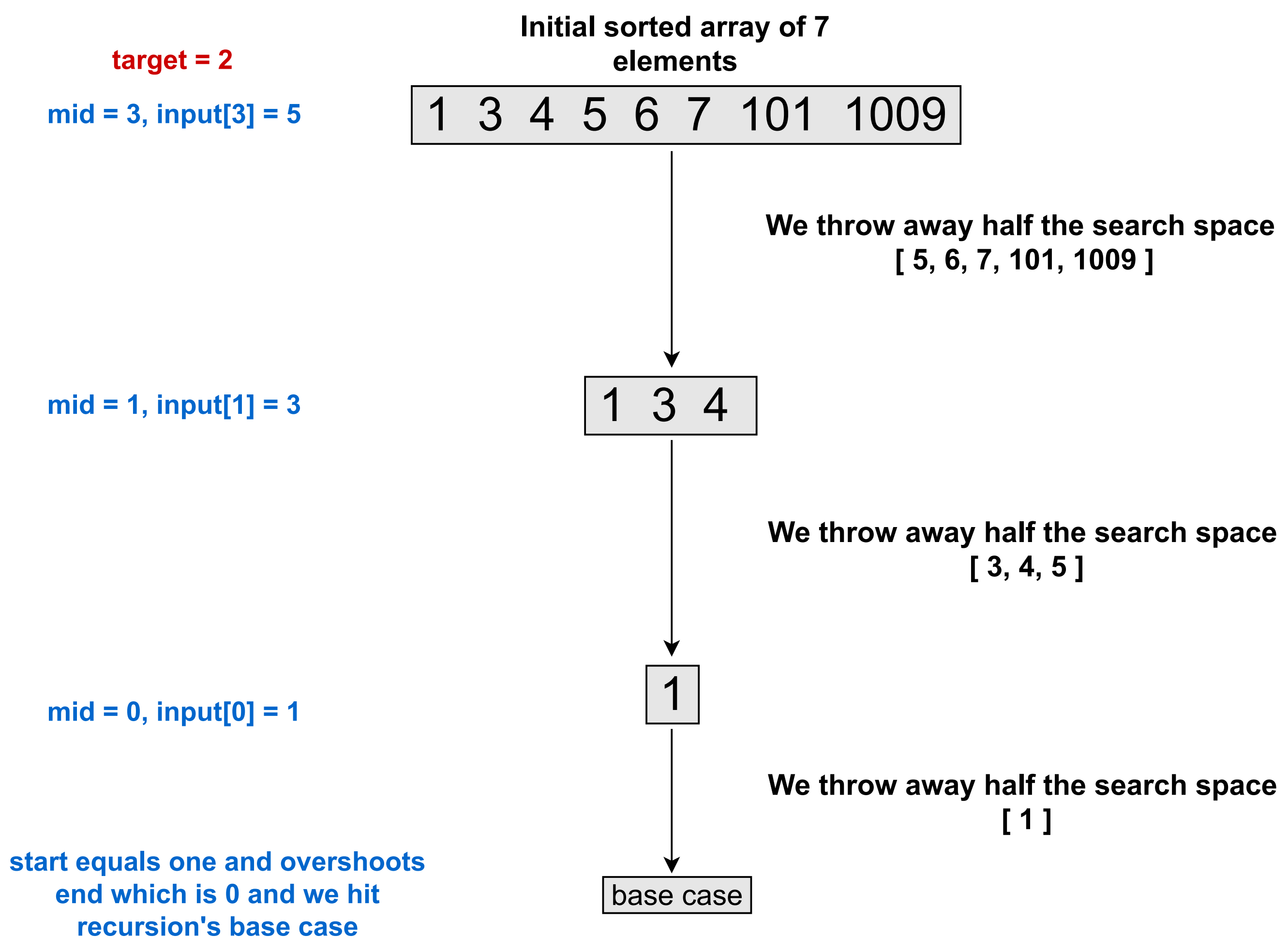
$$log_2 = lgn$$

Note that the way we have implemented our algorithm, we recurse one more level after all the search space is exhausted to detect that there's nothing left to search anymore. The total number of recursion steps would then be **lgn** + **1**

The running time becomes:

$$T(n) = O(1) * (lgn + 1)$$

Below is a pictorial representation of how binary search would proceed using the given algorithm.


target = 2
Initial sorted array of 7 elements
mid = 3, input[3] = 5
1 3 4 5 6 7 101 1009
We throw away half the search space
[5, 6, 7, 101, 1009]
mid = 1, input[1] = 3
1 3 4
We throw away half the search space
[3, 4, 5]
mid = 0, input[0] = 1
1
We throw away half the search space
[1]
start equals one and overshoots end which is 0 and we hit recursion's base case
base case

Working of Binary Search

Recursion Levels' Nuances

The astute reader would notice that even though in the above diagram we claim to be throwing away half the search space, we are actually throwing a little more than that. We also discard the mid value that we just tested against. This strategy results in lgn+1 recursion steps. However, if we throw away exactly half of the search space and unnecessarily keep the mid-index in the search space for the next recursion, the number of recursion steps in the worst case can be lgn+2. That being said, the big O for the algorithm will be unaffected.