**Algorithm Implementation: Binary Search**

|  |
| --- |
| **Part 1 – Implementation**  **public** **static** **int** binarySearch(**int** array[], **int** searchKey){  **int** low = 0;  **int** high = array.length - 1;  **int** mid;  **while** (low <= high){  mid = low + (high-low)/2;  **if** (array[mid] == searchKey){  **return** mid;  }**else** **if** (array[mid] < searchKey){  low = mid + 1;  }**else**{  high = mid - 1;  }  }  **return** -1;  }    **public** **static** **int** binarySearchRecursive(**int**[] array, **int** searchKey){  **return** *binarySearchRecHelper*(array, 0, array.length - 1, searchKey);  }  **public** **static** **int** binarySearchRecHelper(**int** array[], **int** low, **int** high, **int** searchKey){  **if** (high>=low){  **int** mid = low + (high - low)/2;  **if** (array[mid] == searchKey){  **return** mid;  }**else** **if** (array[mid] > searchKey){  **return** *binarySearchRecHelper*(array, low, mid-1, searchKey);  }**else**{  **return** *binarySearchRecHelper*(array, mid+1, high, searchKey);  }  }  **return** -1;  } |

**Part 2 – Overflow Bug:**

* The bug occurs if you use mid = (low + high)/2;. Here you are guaranteed that low <= maximum integer, and high <= maximum integer. However, (low + high) could overflow to a negative number, which means (low + high)/2 could become negative, which means mid could be negative, which means binary search can throw out of bounds exceptions, which means the function that call Binary Search throws exception, etc. The bug could keep propagating up until the user level (program crashes randomly).
* If you use **int** mid = low + (high - low)/2; then low and high are guaranteed <= maximum integerand (high – low) /2 is guranteed less than integer maximum. Also since (high - low)/2 is half the distance between low and high, then low + (high - low)/2 < high <= integer max. So mid is guaranteed not going to overflow ever.

**Part 3 – Time Complexity:**

**Time Complexity:**

* The worst case is we either find the element on the last try (the high = low because the section we are searching has a size equal to one along with the item being a match), or we don’t find the item at all (the high = low because the section we are searching has a size equal to one along with the item not being a match so we return -1).
* In either case, the worst case occurs when we keep search (dividing our search region by 2) to the point where our search region equals one. If our array is of size n, that means we must divide our array by 2 log n times until the search region is one.
* Another way to think of it is that for every time you double your array, you add one extra step to the worst case. This relationship is log n.
* The best case is when you find the element on the first try. The worst case is you find your element on log n. The average case is somewhere in between the two which is still log n.

**Space Complexity:**

* For the iterative verision, the auxillary space is O(1) since no extra space is added.
* For the recursive version, you are creating a constant amount of space (to store recursive case information such as parameters) for ever recursive call.
* For each time your array doubles in the recursive version, you to do one extra call (instead of one extra iteration). This means the space complexity is O( log n).

**Summary:**

|  |  |  |
| --- | --- | --- |
|  | Average and Worst Time | Space Complexity |
| Binary Search Iterative | ***O (log n)*** | ***O (1)*** |
| Binary Search Recursive | ***O (log n)*** | ***O (log n)*** |