Question- 1:

```
def recursive_binary_search(arr, target):
    mid = len(arr) // 2
    if len(arr) == 1:
        return mid if arr[mid] == target else None
    elif arr[mid] == target:
        return mid
    else:
        if arr[mid] < target:
            callback_response = recursive_binary_search((arr[mid:]),
target)
            return mid + callback_response if callback_response is not None
else None
        else:
            return recursive_binary_search((arr[:mid]), target)


numbers = [10, 34, 48, 59, 63, 74, 85, 120, 140]
locate_element=120;


numbers.sort()

index_of_cards=recursive_binary_search(numbers,locate_element)
if index_of_cards is not None:
    print("Search Element present at location {}".format(index_of_cards))
else:
  print("Number is not present in the list")
```

Steps

1. Trying to return the INDEX of the target in the original list being passed in, before it is halved. Getting the target is the easy part that means we need to pass an sorted array.
2. In each iteration it checks length of the array is equal to 1 or mid value is equivalent to target value or not if not then checks that search target element is greater then middle value or not, If true it again recursive call to *recursive_binary_search* method.

Question- 2 (a):

```
    public static int binarySearch(int[] a, int key) {
        int low = 0;
        int high = a.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            int midVal = a[mid];

            if (midVal < key)
                low = mid + 1
            else if (midVal > key)
                high = mid - 1;
            else
                return mid; // key found
```

```
        }
        return -(low + 1);  // key not found.
    }
```

Here below in this statement *int mid = (low + high) / 2*; bug present for array index out of bound.(integer overflow problem)

Best way to fix integer overflow problem here:

```
int mid = low + ((high - low) / 2);
```

Question- 2 (b & c):

There are 2 solutions to fix this problem statement
1. As Maximum marble pointed out, using HashMap would be a more appropriate choice than Hashtable here
2. Using count array

You can use a HashMap to count the occurrences of each unique element in your array, and that would:

Run in linear O(n) time, and
Require O(n) space

Code would be something like this:

1. Iterate through all of the elements of your array once: O(n)
2. For each element visited, check to see if its key already exists in the HashMap: O(1)
3. If it does not (first time seeing this element), then add it to your HashMap as [key: this element, value: 1]. O(1)
4. If it does exist, then increment the value corresponding to the key by 1. O(1),
5. Having finished building your HashMap, iterate through the map and find the key with the highest associated value - and that's the element with the highest occurrence. O(n)

Example of Code using Java:

```
class test1 {
    public static void main(String[] args) {

    //numbers return in marble
    int[] a = {1,1,2,1,5,6,6,6,8,5,9,7,1}; //marble array
    // max occurences of an array
    Map<Integer,Integer> map = new HashMap<>();
      int max = 0 ; int chh = 0 ;
      for(int i = 0 ; i < a.length;i++) {
          int ch = a[i];
          map.put(ch, map.getOrDefault(ch, 0) +1);
      }//for
      Set<Entry<Integer,Integer>> entrySet =map.entrySet();

      for(Entry<Integer,Integer> entry : entrySet) {
```

```
          if(entry.getValue() > max) {max = entry.getValue();chh =
entry.getKey();}

      }//for
    System.out.println("max element => " + chh);
    System.out.println("frequency => " + max);
    }
}
```

Question- 2 (d):

I believe that this problem is memory-constrained binding or merging or sorting of Strings. Efficient dictionaries for storing large amounts of strings in little memory is a very well studied problem solving technique.
First of all need to divide 4GB Big files into small fractions and need to arrange in such a way most important context are covering. Depending upon the requirement you may follow below steps for optimize solutions.

Efficient way to solve this problem is:

1. Trim off as many unnecessary characters as possible
2. Only write the largest dictionary as a batch
3. Performance tuning
4. Radix-sort each batch into outputs by first-character (a-z, non-alphabetic that sorts before A, and non-alphabetic that sorts after z)