```
String[] findRestaurant(String[] list1, string[] list2) {
if (list2.length < list1.length) {
    return fR(list2, list1);
}
    Map<String, Integer> hashmap = new HashMap();
    for(int i=0; i< list1.length(); i+=1) {
        hashmap.put(list1[i], i);
    }

    List<String> result = new Array list();

    int minSum = Integer.MAX_VALUE;

    for(int j=0; j< list2.length; j+=1) {
        if j<= minSum
        if(hashmap.containsKey(list2[j])) {
            sum = j+ hashmap.get(list2[j]);

            if (sum < minSum) {
                result.clear();
                result.add(list2[j]);
                minSum = sum;
            } else if (sum == minSum) {
                result.add(list2[j]);
            }
    }

    return result.toArray(new String[result.size()]);
}
```

T: $O(l_1 + l_2)$
S: $O(\min(l_1, l_2) * x)$

x → average length of string

(2) → Aggregate by key-

→ Aggregate all the information by key

Example-

Given a string, find the first non-repeating character in it and return its index.
If it doesn't exist, return -1

→ Count the occurence of each character first
→ And then go through the results to find out the first unique character

Hashmap
Key → Character
Value → Count

→ Idea - decide your strategy when you encounter an existing key

Strategy → count the occurence

Template-

```
ReturnType solve(List<Type> keys) {
    Map<Type, InfoType> hashmap = new HashMap<>;
    for (Type key : keys) {
```

```
            if ( hashmap.containskey(key)) {
                hashmap.put (key, updated-
                                information);
            }
            hashmap.put (key, value);
        }
        return needed-information;
}
```

① Find unique character in a String

s = leetcode
→ 0

s = loveleetcode
→ 2

s = aabb
→ -1

```
int firstUniqueChar( String s) {
    Map< Character, Integer> hashmap =
                        new HashMap();
    for (int i=0; i< s.length(); i+=1) {
        char c = s.charAt(i);
        hashmap.put(c, hashmap.getOf
                        Default(c, 0)+1);
    }
    for(int i=0; i< s.length(); i+=1){
```

```
            if (hashmap.get(s.charAt(i))==1){
                return i;
            }
        }
        return -1;
    }
```

T = O(N)
S = O(1)

② → Intersection of two arrays –

nums1 = [1,2,2,1]
nums2 = [2,2]
→ [2,2]

nums1 = [4,9,5]
nums2 = [9,4,9,8,4]
→ [4,9]

```
int[] intersect(int[] nums1, int[] nums2) {

    if(nums2.length < nums1.length) {
        return intersect(nums2, nums1);
    }

    Map(Int, Int) hashmap = new HashMap();

    for(int num : nums1) {
        hash map.put(num, harhmap.getOrDefault
                          (num, 0) +1);
    }

    int k=0;
    int[] result = new int[nums1.length];
```

```
for(int num: nums2) {
    int count= hashmap.getOrDefault(num,0);
    if (count>0) {
        result[k]= num;
        k++;
        hashmap.put(num, count-1);
    }
}
return Arrays.copyOfRange(result, 0, k);
}
```

③ → Contains Duplicate II –

nums = [1, 2, 3, 1] k = 3 (0, 3)

→ true

i, j → distinct
nums[i] = nums[j]
abs(i-j) <= k

nums = [1, 0, 1, 1] k = 1

→ true

nums = [1, 2, 3, 1, 2, 3], k = 2

→ false

```
boolean containsNearbyDuplicate(int[] nums, int k) {
    Map<Integer, Integer> hashmap = new Hashmap();
    for(int i=0; i< nums.length; i++) {
        if( hashmap.containsKey(nums[i])) {
            int i1 = hashmap.get(nums[i]);
            if (Math.abs(i1-i) <= k) {
                return true;
            }
```

```
        }
        hashmap.put(nums[i], i);
    }
}
return false;
}
```

④ Logger rate limiter →

Logger → takes a stream of messages → Each unique message should only be printed atmost every 10 seconds

msg → 2, "Newton" → [Logger] → "Newton"

msg → 11, "Newton" → [Logger] → ? No → 11 < 2+10

timestamp, message

```
class Logger {
    Map<String, Integer> hashmap;
    Logger() {
        hashmap = new Hashmap();
    }

    boolean shouldPrintMessage(int timestamp,
                               String message) {
        if(!hashmap.containskey(message)) {
            hashmap.put(message, timestamp);
            return true;
        }
        int oldTimeStamp = hashmap.get(message);
```

```
        if( timeStamp - oldTimeStamp >= 10) {
            hashmap.put(menage, timestamp);
            return true;
        }
        return false;
    }
```

Design the key-

→ In prev questions, choosing the key was easy. But its not always the case. Sometimes, we have to think it over to design a suitable key when using a hashtable.

Example-

Given an array of stings, group anagrams together.

[eat, ate, cat, tac]

→ [[eat, ate], [cat, tac]]

Process→

Designing a key → Build a mapping relationship by yourself b/w original information and actual key used by hash map

- → All values belong to same group will be mapped in same group
- → Values which need to be separated into different groups will not be mapped into same group

Mapping strategy for above example

↳ sort the string and use it as key

eat → aet
ate → aet
↳ Tricky.

① → Group anagrams →

strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
→ [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

strs = [""]
→ [[""]]

strs = ["a"]
→ [["a"]]

```
List<List<String>> groupAnagrams(String[] strs){
    if(strs.length == 0){
        return new ArrayList();
    }
    Map<String, List<String>> hashmap =
        new HashMap();
    int[] count = new int[26];
    for(String s: strs){
        Arrays.fill(count, 0);
        for(char c: s.toCharArray()){
            count[c-'a']+=1;
```

}

```
StringBuilder sB = new StringBuilder();
for(int i=0; i< 26; i+=1) {
    sB.append('#');
    sB.append(count[i]);
}

String key = sB.toString();

if(!hashmap.containsKey(key)) {
    hashmap.put(key, new ArrayList<String>());
}
hashmap.get(key).add(s);
}

return new ArrayList(ans.values());
}
```

T: O(NK)
S: O(NK)

② Group shifted strings –

strings = ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]

→ [["acef"], ["a", "z"], ["abc", "bcd", "xyz"], ["az", "ba"]]

strings = ["a"]

→ [["a"]]

```
char shiftLetter(char letter, int shift) {
    return (char)((letter - shift + 26) % 26 + 'a');
}

String getHash(String s) {
    char[] chars = s.toCharArray();

    int shift = chars[0];
    for (int i = 0; i < chars.length; i += 1) {
        chars[i] = shiftLetter(chars[i],
                               shift);
    }

    String hashKey = String.valueOf(chars);

    return hashKey;
}
```

99 97
c d e a → a b c
100 101

((<99 - 99> % 26) + 'a' → a
((<100 - 99> % 26) + 'a' → b

((<97 - 99> % 26) + 'a' → 'y'?

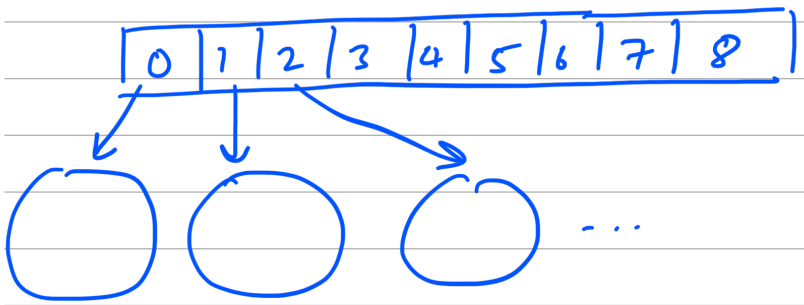
((97 - 99 + 26) % 26) + 'a'

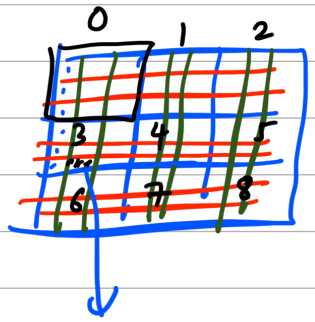
```
List<List<String>> groupStrings(String[] strings) {
    Map<String, List<String>> hashmap = new
                                HashMap();

    for (String s: strs) {
        String hashkey = getHash(s);

        if (!hashmap.containsKey(hashKey)) {
            hashmap.put(hashkey, new
                        ArrayList<>());
```

```
        }
        hashmap.get(hashKey).add(s);
    }
    return new ArrayList(hashmap.values());
}
```

③ → Valid Sudoku →

Check if a 9 x 9 sudoku board is valid or not





r = 5
c = 1

boxes $\langle (r/3) \times 3 + (c/3) \rangle$?

T → $[\frac{6 \ldots 8}{3}, \frac{3 \ldots 5}{3}]$

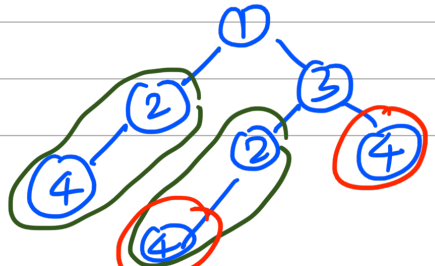
$2 \times 3 + 1 \Rightarrow 7$

```
boolean isValidSudoku(char[][] board) {
    int N = 9;

    HashSet<Character>[] rows = new HashSet[N];
    HashSet<Character>[] cols = new HashSet[N];
    HashSet<Character>[] boxes = new HashSet[N];

    for (int r=0; r<N; r+=1) {
        for (int c=0; c<N; c+=1) {
```

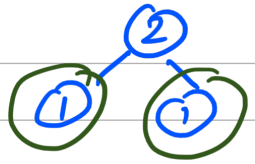
```
char val = board[r][c];

if (val == '.') {
        continue;
}

if(rows[r].contains(val)){
        return false;
}
rows[r].add(val);

if(cols[c].contains(val)){
        return false;
}
cols[c].add(val);

int boxIdx = (r/3)*3 + (c/3);

if (boxes[boxIdx].contains(val)) {
        return false;
}
boxes[boxIdx].add(val);

        }
    }

    return true;
}
```

$T = O(N^2) \rightarrow O(1)$
$S = O(N^2)$

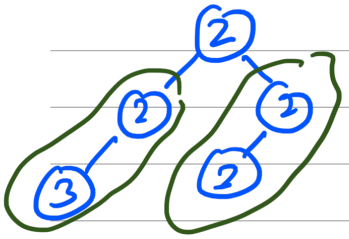
④ → find duplicate subtrees -



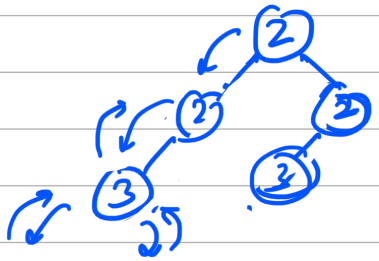
→ [[2,4], [4]]



→ [[1]]



→ [[2,3],[3]]



3(,) → 1

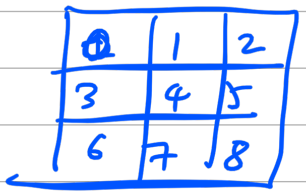
2(3(,),) → X 2

2(2(3(,),),2(3(,),)) → 1 L → [3], [2]

```
List<TreeNode> findDuplicateSubtrees(TreeNode root){
    Map<String,Integer> hashmap2 new HashMap();
    List<TreeNode> result2 new LinkedList<>();
    postOrder(root, hashmap, result);
}

String postOrder(TreeNode cur, Map<String,Integer>
                 hashmap, List<TreeNode> result){
    if(cur == null){
        return "n";
    }

    String serial2 cur.val + "(" +
        postOrder(cur.left, hashmap,
        result) + "," +
        postOrder(cur.right, hashmap,
        result);

    map.put(serial, map.getOrDefault(serial,0)+1);
    if(map.get(serial)==2){
        result.add(cur);
    }
    return serial;
}
```

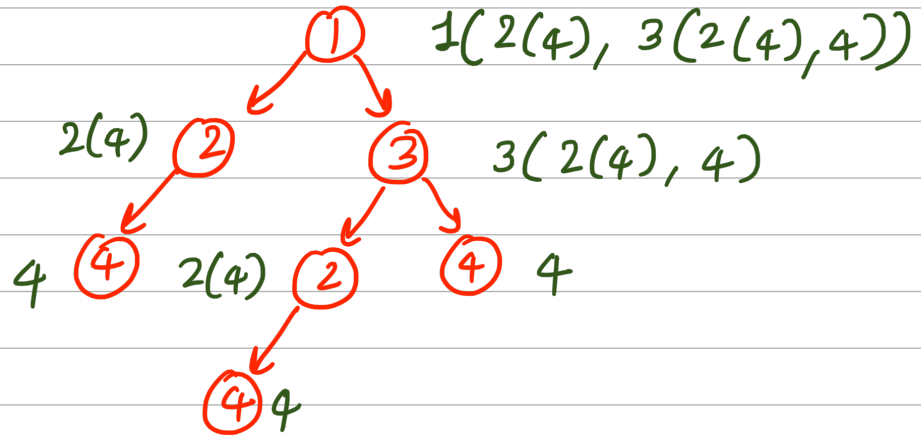
Design the Key: Summary

- Sorted string/array as the key
- offset
- matrix: row index (or) column index as key
- sudoku: box

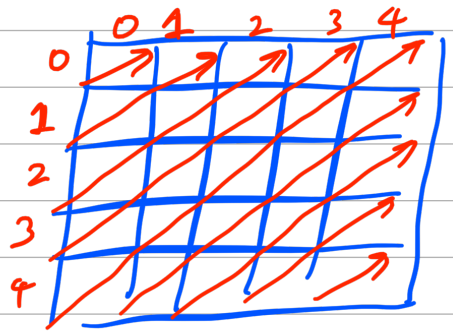


$(r/3)\times 3 + (c/3)$

- In trees, you might want to directly use the TreeNode as key sometimes. But in most cases, the serialization of subtree might be a better idea



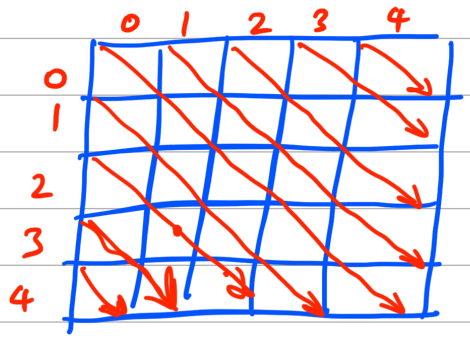
- Sometimes, in a matrix, you might want to aggregate the values in same diagonal line



0, 1, 2, ..., 7

Anti-diagonal order

(i, j) → i+j



Diagonal order
(i,j) → i-j

Homework-

① → Jewels and stones <#771>

② → Longest Substring without repeating characters <#3>

③ → Two Sum III - Data structure Design <#170>

④ → 4Sum II <#454>

⑤ → Top K frequent elements <#347>

⑥ → Unique word abbreviation <#288>

⑦ → Insert Delete GetRandom O(1) <#380>

Connect & Share with me

Thanks

CoDe Bug

/anvj-kumar-9a188968

* Connect and ping me here. Will share today's resources
* Endorse me in teaching and data structures

/mr.anvj.brandy

Q & A