179. Largest Number

We will use merge sort to sort all numbers according to the rule: str(a) +str(b) > str(b) + str(a)

TC is O(nlogn) SC is O(n).

```
class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        nums = list(map(str, nums))
        nums = self.mergeSort(nums, 0, len(nums) - 1)
        return str(int(''.join(nums)))

    def mergeSort(self, nums, l, r):
        if l > r:
            return
        if l == r:
            return [nums[l]]
        mid = l + (r - l) // 2
        left = self.mergeSort(nums, l, mid)
        right = self.mergeSort(nums, mid + 1, r)
        return self.merge(left, right)

    def merge(self, l1, l2):
        res, i, j = [], 0, 0
        while i < len(l1) and j < len(l2):
            if not self.compare(l1[i], l2[j]):
                res.append(l2[j])
                j += 1
            else:
                res.append(l1[i])
                i += 1
        res.extend(l1[i:] or l2[j:])
        return res

    def compare(self, n1, n2):
        return n1 + n2 > n2 + n1
```

676. Implement Magic Dictionary

We will group all words in dict by length, then every time search word, we will iterate through all words with that length and return whether it exists. TC is O(n), O(n)

```
from collections import defaultdict
class MagicDictionary:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = defaultdict(list)

    def buildDict(self, dict: List[str]) -> None:
        """
        Build a dictionary through a list of words
        """
        for word in dict:
            self.trie[len(word)].append(word)


    def search(self, word: str) -> bool:
        """
        Returns if there is any word in the trie that equals to the given word after modifying exactly one character
        """
        for w in self.trie[len(word)]:
            i, count = 0, 0
            for i in range(len(word)):
                if w[i] != word[i]:
                    count += 1
                    if count > 2:
                        break
            if count == 1:
                return True
        return False
```

677. Map Sum Pairs

We will use trie to record all values and then use bfs when searching. TC is O(n * len(w)), O(n * len(w))

```
class MapSum:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = {}

    def insert(self, key: str, val: int) -> None:
        node = self.trie
        for i in key:
            if i not in node:
                node[i] = {}
            node = node[i]
        node['val'] = val

    def sum(self, prefix: str) -> int:
        node = self.trie
        res = 0
        for i in prefix:
            if i not in node:
                return 0
            node = node[i]
        cur = [node]

        while cur:
            next_ite = []
            for n in cur:
                for key, val in n.items():
                    if key == 'val':
                        res += val
                    else:
                        next_ite.append(val)
            cur = next_ite
        return res
```

745. Prefix and Suffix Search

We will use two trie to store all words by prefix and suffix and store all index to the associated array. TC is O(LN), O(L + N)

```
from collections import defaultdict
class WordFilter:

    def __init__(self, words: List[str]):
        self.prefix = {}
        self.suffix = {}
        for idx, word in enumerate(words):
            i, length = 0, len(word)
            node_pre, node_suf = self.prefix, self.suffix
            for i in range(length):
                if word[i] not in node_pre:
                    node_pre[word[i]] = {}
                if '#' not in node_pre[word[i]]:
                    node_pre[word[i]]['#'] = [idx]
                else:
                    node_pre[word[i]]['#'].append(idx)
                node_pre = node_pre[word[i]]

                if word[length - i - 1] not in node_suf:
                    node_suf[word[length - i - 1]] = {}
                if '#' not in node_suf[word[length - i - 1]]:
                    node_suf[word[length - i - 1]]['#'] = [idx]
                else:
                    node_suf[word[length - i - 1]]['#'].append(idx)

                node_suf = node_suf[word[length - i - 1]]
        self.prefix['#'] = list(range(len(words)))
        self.suffix['#'] = list(range(len(words)))


    def f(self, prefix: str, suffix: str) -> int:
        node_pre, node_suf = self.prefix, self.suffix
        for i in prefix:
            if i not in node_pre:
                return -1
            node_pre = node_pre[i]

        for i in reversed(suffix):
            if i not in node_suf:
                return -1
            node_suf = node_suf[i]
        if '#' in node_pre and '#' in node_suf:
            l1, l2 = node_pre['#'], node_suf['#']
        else:
            return -1
        length1, length2 = len(l1), len(l2)
        i, j = length1 - 1, length2 - 1
        while i >= 0 and j >= 0:
            if l1[i] > l2[j]:
                i -= 1
            elif l1[i] < l2[j]:
                j -= 1
            else:
                return l1[i]
        return -1
```

1019. Next Greater Node In Linked List

We will iterate through our array from tail to head and use a stack to store all numbers larger than current one. In the end, we will return our reverted res. TC is O(nlogn)

```
from bisect import *
class Solution:
    def nextLargerNodes(self, head: ListNode) -> List[int]:
        res = []
        arr = []
        stack = []
        while head:
            arr.append(head.val)
            head = head.next

        while arr:
            val = arr.pop()
            idx = bisect(stack, val)
            if idx == len(stack):
                res.append(0)
                stack = [val]
            else:
                res.append(stack[idx])
                stack = [val] + stack[idx:]
        return reversed(res)
```