Branch: master OOP-Design / ******AllOtherQuestions******.java Find file Copy path

huyilong SUMMARY - ALLLL **All the Remaining Problems Summary** Problems 4384e87 just now

1 contributor

924 lines (885 sloc) | 32.7 KB Raw Blame History

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *next;}
Populate each next pointer to point to its next right node.
If there is no next right node, the next pointer should be set to NULL.
public class Solution {
    public void connect(TreeLinkNode root) {
        //The basic idea is have 4 pointers to move towards right on two levels (see comments in the code).
        if(root == null)
        return;
        TreeLinkNode lastHead = root;//prevous level's head  -- linkedlist head
        TreeLinkNode lastCurrent = null;//previous level's pointer -- linkedlist scanner
        TreeLinkNode currentHead = null;//currnet level's head  -- linkedlist head
        TreeLinkNode current = null;//current level's pointer -- linkedlist scanner
        while(lastHead!=null){
            //make sure the last linkedlist is not null
            //let the scanner to point to this head at first to scan
            lastCurrent = lastHead;
            while(lastCurrent!=null){
            if(currentHead == null){
                currentHead = lastCurrent.left;
                current = lastCurrent.left;
            }else{
                current.next = lastCurrent.left;
                current = current.next;
            }
            if(currentHead != null){
                current.next = lastCurrent.right;
                current = current.next;
            }
            lastCurrent = lastCurrent.next;
        }
        //update last head
        lastHead = currentHead;
        currentHead = null;
    }
    }
}
int[] m = new int[26] ---- ++m[s.charAt(i) - 'a']; ---  --m[t.charAt(i) - 'a']
-----------------create a array of 26 int to be the hashmap of characters--------
public class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;
        int[] m = new int[26];

        for (int i = 0; i < s.length(); ++i)
            ++m[s.charAt(i) - 'a'];

        for (int i = 0; i < t.length(); ++i) {
            if (--m[t.charAt(i) - 'a'] < 0)
                return false;
        }
        return true;
    }
}
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int res = 0;
        //32 bit we already know
        for(int i=0; i<32;i++){
```

```
            if((n&1) == 1){
                res = (res<<1) +1;//fill 1 to the right
            }else{
                res = res<<1;//automatically fill in the 0 to the right
            }
            //search for next bit in the 32-bit int
            n = n>>1;
        }

        return res;
    }
}
Palindrome Permutation
Given a string, determine if a permutation of the string could form a palindrome.
For example, "code" -> False, "aab" -> True, "carerac" -> True.
Consider the palindromes of odd vs even length. What difference do you notice?
Count the frequency of each character. If each character occurs even number of times, then it must be a palindrome.
How about character which occurs odd number of times? -- must <=1 such kind of num
本题也可以用一个HashSet，第偶数个字符可以抵消Set中的字符，最后判断Set的大小是否小于等于1就行了。
public class Solution {
    public boolean canPermutePalindrome(String s) {
        Set<Character> set = new HashSet<Character>();
        for(int i = 0; i < s.length(); i++){
            // 出现的第偶数次，将其从Set中移出
            if(set.contains(s.charAt(i))){
                set.remove(s.charAt(i));
            } else {
            // 出现的第奇数次，将其加入Set中
                set.add(s.charAt(i));
            }
        }
        // 最多只能有一个奇数次字符
        return set.size() <= 1;
    }
}
public class Solution {
    public boolean canPermutePalindrome(String s) {
        Map<Character, Integer> map = new HashMap<Character, Integer>();
        // 统计每个字符的个数
        for(int i = 0; i < s.length(); i++){
            char c = s.charAt(i);
            Integer cnt = map.get(c);
            if(cnt == null){
                cnt = new Integer(0);
            }
            map.put(c, ++cnt);
        }
        // 判断是否只有不大于一个的奇数次字符
        boolean hasOdd = false;
        for(Character c : map.keySet()){
            if(map.get(c) % 2 == 1){
                if(!hasOdd){
                    hasOdd = true;
                } else {
                    return false;
                }
            }
        }
        return true;
    }
}
Given a string s, return all the palindromic permutations (without duplicates) of it.
Return an empty list if no palindromic permutation could be form.
For example:
Given s = "aabb", return ["abba", "baab"].
Given s = "abc", return [].
public class Solution {
    public List<String> generatePalindromes(String s) {
        List<String> results = new ArrayList<>();
        if(s.length() == 0)
            return results;

        HashMap<Character, Integer> d = new HashMap<>();
        for(int i = 0; i < s.length(); i++) {
            if(d.containsKey(s.charAt(i)))
                d.put(s.charAt(i), d.get(s.charAt(i)) + 1);
            else
                d.put(s.charAt(i), 1);
        }
        String candidate = "";
```

```java
        String single = "";
        boolean already = false;
        for(Character c : d.keySet()) {
            int num = d.get(c) / 2;
            for(int i = 0; i < num; i++)
                candidate += c;
            if(d.get(c) % 2 != 0) {
                if(already)
                    return results;
                else {
                    already = true;
                    single += c;
                }
            }
        }
        if(candidate.length() == 0 && single.length() != 0) {
            results.add(single);
            return results;
        }
        recursion("", candidate, single, candidate.length(), results);
        return results;
    }
    private void recursion(String left, String candidate, String single, int l, List<String> results) {
        if(left.length() == l) {
            String right = new StringBuffer(left).reverse().toString();
            results.add(left + single + right);
        }
        for(int i = 0; i < candidate.length(); i++) {
            if(i > 0 && candidate.charAt(i) == candidate.charAt(i - 1))
                continue;
            recursion(left + candidate.charAt(i), candidate.substring(0, i) + candidate.substring(i + 1), single, l, results);
        }
    }
}

public class Solution {
    public List<String> summaryRanges(int[] nums) {
        //两个指针 start, end.  如果nums[end+1] = nums[end]+1, 就移动end指针, 否则, 插入字符串nums[start]->nums[end].
        List<String> res = new ArrayList<>();
        if(nums==null || nums.length<1) return  res;
        int s=0, e=0;  //start means s and end means e
        //s and e will create a window for the range
        while(e<nums.length) {
            //outside e<nums.length
            //but inside we still use e+1<nums.length
            if(e+1<nums.length && nums[e+1]<=nums[e]+1) {
                 e++;
            }else{
                if(s==e) {
                    //only one element within the window
                    //convert integer to string
                    //using Integer.toString
                    //Integer.parseInt
                    res.add(Integer.toString(nums[s]));
                }else{
                    //there are multiple elements within the window
                    //we need to use -> now
                    String str = nums[s] + "->" + nums[e];
                    res.add(str);
                }
                 ++e; s=e;
            }
        }
        return res;
    }
}
pattern = "abba", str = "dog cat cat dog" should return true.
pattern = "abba", str = "dog cat cat fish" should return false.
pattern = "aaaa", str = "dog cat cat dog" should return false.
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        String[] words = str.split(" ");
        char[] patterns = pattern.toCharArray();
        HashMap<Character, String> map = new HashMap<>();
        if(words.length != patterns.length){
             return false;
         }
        for(int i=0; i<patterns.length; i++){
            if(!map.containsKey(patterns[i])){
                if(map.containsValue(words[i])){
```

```
                    //this one is so similar to longest non-repetitive word
                    //no such a key but the value has already existed
                    //must false because each word can only have one key to reach it
                    return false;
                }
                map.put(patterns[i], words[i]);
            }else{
                if(!map.get(patterns[i]).equals(words[i])){
                    return false;
                }
            }
        }
        return true;
    }
}
 Longest Substring Without Repeating Characters
 public class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s.length() == 0 || s ==null){
            return 0;
        }
        HashSet<Character> hs = new HashSet<>();
        int max = 0;
        int len  = s.length();
        //at least O(n^2)
        for(int i=0; i<len; i++){
            int temp = 0;

            for(int j=i; j<len; j++){
                if(!hs.contains(s.charAt(j))){
                    hs.add(s.charAt(j));
                    temp++;
                    if(j == len-1){
                        max = Math.max(max, temp);
                        //return Math.max(max, temp);
                    }
                }else{
                    max = Math.max(max, temp);
                    hs.clear();
                    break;//current loop does not need to continue
                }
            }
        }
        return max;
    }
}
Find The First Non Repeated Character In A String
public static Character firstNonRepeatedCharacter(String str){
        HashMap<Character,Integer>  characterhashtable=
                     new HashMap<Character ,Integer>();
        int i,length ;
        Character c ;
        length= str.length();  // Scan string and build hash table
        for (i=0;i < length;i++){
            c=str.charAt(i);
            if(characterhashtable.containsKey(c))  characterhashtable.put(  c ,  characterhashtable.get(c) +1 );
            else   characterhashtable.put( c , 1 ) ;
        }
        // Search characterhashtable in in order of string str
        for (i =0 ; i < length ; i++ ){
            c= str.charAt(i);
            if( characterhashtable.get(c)  == 1 )
            return c;
        }
        return null ;
}

public class Solution {
    public boolean isUgly(int num) {
        //Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since
        //Note that 1 is typically treated as an ugly number.
        if(num<=0){
            return false;
        }

        while(num!=1){
            if(num%5 == 0){
                //put the larger number at first
                num /= 5;
            }else if(num%3 == 0){
                num /=3;
```

```java

            }else if(num%2==0){
                num /= 2;
            }else{
                return false;
            }
        }
         return true;
    }
}

public class Solution {
    public int nthUglyNumber(int n) {
/*
这道题是之前那道Ugly Number 丑陋数的延伸，这里让我们找到第n个丑陋数，还好题目中给了很多提示，基本上相当于告诉我们解法了，根据提示中的信息，我们知道丑陋数
(1) 1×2, 2×2, 3×2, 4×2, 5×2, …
(2) 1×3, 2×3, 3×3, 4×3, 5×3, …
(3) 1×5, 2×5, 3×5, 4×5, 5×5, …
仔细观察上述三个列表，我们可以发现每个子列表都是一个丑陋数乘以2,3,5，而这些丑陋数的值就是从已经生成的序列中取出来的，我们每次都从三个列表中取出当前最小的
*/
            if(n==1) return n;
            Queue<Long> q = new PriorityQueue<Long>();
            int[] nums = {2,3,5};
            Long result = Long.valueOf(1);
            q.offer(result);
            for(int i=0;i<n;i++){
                // Each time we poll the peak value of q, is the ith number
                result = q.poll();
                for(int num:nums){
                    Long uglyNum = result*num;
                    if(!q.contains(uglyNum)){
                        q.offer(uglyNum);
                    }
                }
            }
            return result.intValue();
    }
}

/*
Given 2*n + 2 numbers, every numbers occurs twice except two, find them.
Example
Given [1,2,2,3,4,4,5,3] return 1 and 5
Thinking Process:
The 2 exception must have this feature: a ^ b != 0, since they are different
Still want to do 2n + 1 problem as in Single Number I,
then we need to split a and b into 2 groups and deal with two 2n+1 problems
Assume c = a^b, there mush be a bit where a and b has the difference, so that bit in c is 1.
Find this bit position and use it to split the group:
ff
*/

public class Solution {
    public List<Integer> singleNumberIII(int[] A) {
        if (A == null || A.length == 0) {
            return null;
        }
        List<Integer> rst = new ArrayList<Integer>();
        int xor = 0;
        for (int i = 0; i < A.length; i++) {
            xor ^= A[i];
        }
        int bitOnePos = 0;
        for (int i = 0; i < 32; i++) {
            if ((xor >> i & 1) == 1) {
                bitOnePos = i;
            }
        }
        int rstA = 0;
        int rstB = 0;
        for (int i = 0; i < A.length; i++) {
            if ((A[i] >> bitOnePos & 1) == 1) {
                //this group has 2n+1 the result will be one of the single number
                rstA ^ = A[i];
            } else {
                //this group is also 2n+1 with the result being another single number
                rstB ^ = A[i];
            }
        }
        rst.add(rstA);
```

```
        rst.add(rstB);
        return rst;
    }
}

Different Ways to Add Parentheses
这题就是分治法- Divide and Conquer的一个例子。
在递归的过程中，根据符号位，不断将一个字符串分成两个子串，然后将两个子串的结果merge起来。
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        List<Integer> result = new ArrayList<Integer>();
        if (input == null || input.length() == 0) {
            return result;
        }

        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);
            if (c != '+' && c != '-' && c != '*') {
                continue;
            }

            List<Integer> part1Result =
                diffWaysToCompute(input.substring(0, i));
            List<Integer> part2Result =
                diffWaysToCompute(input.substring(i + 1, input.length()));

            for (Integer m : part1Result) {
                for (Integer n : part2Result) {
                    if (c == '+') {
                        result.add(m + n);
                    } else if (c == '-') {
                        result.add(m - n);
                    } else if (c == '*') {
                        result.add(m * n);
                    }
                }
            }
        }

        if (result.size() == 0) {
            result.add(Integer.parseInt(input));
        }

        return result;
    }
}

/*
Given an array of meeting time intervals consisting of start and end times
[[s1,e1],[s2,e2],…] (si < ei), determine if a person could attend all meetings.
For example, Given [[0, 30],[5, 10],[15, 20]], return false.
1. sorting by start time
2. sorting by end time
sort interval first by start time and then by end time
traverse the sorted intervals and check if two intervals have intersection with each other
*/
public class Interval{
        //default as private members
        int start;
        int end;
        Interval(){
                start = 0;
                end=0;
        }
        Interval(int start, int end){
                this.start = start;
                this.end = end;
        }
}
public class Solution{
        public boolean canAttendMeetings(Interval[] schedules){
                if(schedules == null || shcedules.length == 0){
                        return false;
                }
                //Arrays.sor(arr, cmp)
                Comparator<Interval> cmp = new Comparator<Interval>(){
                        public int compare(Interval o1, Interval o2){
                                if(o1.start != o2.start){
                                        //we firstly compare the starting time
                                        return o1.start - o2.start;
```

```
                                }else{
                                        //then compare the end time
                                        return o1.end - o2.end;
                                }
                        }
                };
                Arrays.sort(schedules, cmp);
                //we are stating from 1
                for(int i=1;i<schedules.length; i++){
                        if(schedules[i].start < schedules[i-1].end){
                                return false;
                        }
                }

                //after checking all these intervals
                //we sort each interval by starting time and then end time
                return true;
        }
}
/*
Sorting
sort the given intervals first by starting index and then by ending index
1. traverse the given interval and
keep comparing current interval to the *last* interval in the result
2.1 if no intersection, just add current interval into result
2.2 otherwise, update the ending index of last interval
in the result if necessary
*/
public class Interval{
        private int start;
        private int end;

        public Interval(){
                this.start = 0;
                this.end = 0;
        }

        public Interval(int start, int end){
                this.start = start;
                this.end = end;
        }
}
public class Solution{
        public List<Interval> merge(List<Interval> intervals){
                List<Interval> res = new ArrayList<>();
                if(intervals == null || intervals.size() ==0){
                        return res;
                }
                Comparator<Interval> cmp = new Comparator<Interval>(){
                        public int compare(Interval o1, Interval o2){
                                if(o1.start != o2.start){
                                        return o1.start - o2.start;
                                }else{
                                        return o1.end - o2.end;
                                }
                        }
                };
                Collections.sort(intervals, cmp);
                Interval ini = new Interval(intervals.get(0).start, intervals.get(0).end);
                res.add(ini);

                int i=1;//starting from the second interval in the sorted array
                //and always compare to the **last** interval in the res array
                while(i<intervals.size()){
                        //loop until all the intervals are merged
                        Interval cur = intervals.get(i);
                        Interval prev = res.get(res.size()-1);////last interval in the result
                        if(cur.start > prev.end){
                                //no intersection between these two intervals
                                //we just need to add it into the res without merging
                                res.add(new Interval(cur.start, cur.end));
                                //as long as we are adding result into the res
                                //we need to each time new the sub!!!!! and then add
                        }else{
                                //there is a intersection between two intervals
                                //we need to merge them together
                                //and substitute the old one in the arraylist
                                //////THIS IS WRONG!!!!!!
                                //Interval merged = new Interval(prev.start, cur.end);
                                Interval merged = new Interval(prev.start, prev.end > cur.end ? prev.end : cur.end);
```

```
                                res.set(res.size()-1, merged);
                        }
                        i++;
                }
                return res;
        }
}
class MinStack {
    //比较容易想到就是要追溯这个最小值，在push的时候维护最小值，但是如果pop出最小值的时候该如何处理呢，如何获得第二小的值呢？
    //如果要去寻找又不是常量时间了。解决的方案是再维护一个栈，我们称为最小栈，如果遇到更小的值则插入最小栈，否则就不需要插入最小栈（注意这里正常栈是怎么
    //这里的正确性在于，如果后来得到的值是大于当前最小栈顶的值的，那么接下来pop都会先出去，而最小栈顶的值会一直在，而当pop到最小栈顶的值时，一起出去后接
    //如此push时最多插入两个栈一个元素，是O(1)，top是取正常栈顶，自然是O(1)，而pop时也是最多抛出两个栈的栈顶元素，O(1)。最后getMin只需要peek最小栈顶
    //实现了所有操作的常量操作，空间复杂度是O(n)，最小栈的大小。代码如下：
    //space complexity we just need to take consideration of the additional space
    //here we need an additional stack -- O(N)
    List<Integer> stack = new ArrayList<>();
    List<Integer> minStack = new ArrayList<>();
    public void push(int x) {
        //anyway we need to add into the original stack
        stack.add(x);
        //when push we need to take care of minStack after pushing into original stack anyhow
        if(minStack.isEmpty() || minStack.get(minStack.size()-1)>=x){
            //min stack always need to be small heap!!!
            minStack.add(x);
        }
    }
    public void pop() {
        if(stack.isEmpty()){
             return;
        }else{
            //stack has sth in it
            //remove the last one of the arraylist
            //!!!! for arraylist or hashmap we all could use map.remove(key/index)
            int elem = stack.remove(stack.size()-1);
            //after remove the original stack
            //we need to also check that whether minStack also needs to be updated as well
            //O(1)
            if(!minStack.isEmpty() && elem == minStack.get(minStack.size()-1)){
                minStack.remove(minStack.size()-1);
            }
        }

    }
    public int top() {
        if(stack.isEmpty()){
            //nothing here
            return -1;
        }else{
            return stack.get(stack.size()-1);
        }
    }
    public int getMin() {
        if(minStack.isEmpty()){
            return -1;
        }else{
            return minStack.get(minStack.size()-1);
        }
    }
}
public class Solution {
    public String simplifyPath(String path) {
        if(path.length() == 0 || path == null){
            return path;
        }

        String[] splits = path.split("/");
        //what if a/b//c?
        //then the arr should contain a,b, ,c  4 elements with one is empty
        //we need to get rid of empty
        //if(s.length() == 0 || s.equals(".") continue
        for(String s:splits){
            System.out.println(s);
        }
        LinkedList<String> stack = new LinkedList<String>();
        for (String s : splits) {
            if(s.length()==0 || s.equals(".")){
                //stay in current dir
                continue;
            }else if(s.equals("..")){
                //go to the previous dir
```

```
                //each time when you want to pop from the stack
                //you must check whether it is empty
                if(!stack.isEmpty()){
                    stack.pop();
                }
            }else{
                //build the path further
                stack.push(s);
            }
        }

        if(stack.isEmpty()){
            //if parse to the end still empty
            stack.push("");
        }
        StringBuilder res = new StringBuilder();
        while(!stack.isEmpty()){
            //stack.removeLast()
            //stack is LIFO
            //so if we want to remove the first
            //we need to use stack.removeLast()
            //res.append("/").append(stack.pop());
            res.append("/").append(stack.removeLast());
        }
        return res.toString();
    }
}
/*
public class StackData{
        public int start;
        public int size = 0;
        public int capacity =100;
        //we must use public for each var to define their scopes
}*/
public class Stack{
        public stackSize = 100;
        int[] buffer = new int[stackSize * 3];
        //use a single array of buffer to implement 3 stacks
        //tops is an array of 3 numbers for each top index of the stack
        int[] tops = {-1, -1, -1};//here we are initializing the array directly
        public void push(int stackNum, int value) throws Exception{
                if(tops[stackNum] >= stackSize){
                        throw new FullStackException();
                }
                //update the pointer for checking capacity
                tops[stackNum]++;
                //update the value at this position
                int index = stackNum * stackSize + tops[stackNum];
                buffer[index] = value;

        }
        //for pop we just need to specify which stack to pop without value
        public void pop(int stackNum) throws Exception{
                if(isEmpty(stackNum)){
                        throw new EmptyStackException;
                }
                int top_index = stackNum * stackSize + tops[stackNum];
                //update the tops index
                tops[stackNum]--;
                //before we clear the value at this position we must get the value at first
                int value = buffer[top_index];

                /////here we forget to do one thing which is clear the value in the buffer
                buffer[top_index] = 0;
                return value;
        }
        public int peek(int stackNum) throws Exception{
                if(isEmpty(stackNum)){
                        throw new EmptyStackException();
                }
                int top_index = stackNum * stackSize + tops[stackNum];
                return buffer[top_index];
        }
        public boolean isEmpty(int stackNum){
                return tops[stackNum] == -1;
        }
}
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<Integer>();
```

```
        if(matrix == null || matrix.length == 0) return result;
        //the row and col could be different we need
        //to make sure this point
        int row = matrix.length;
        int col = matrix[0].length;
        int x=0;
        int y=0;

        while(row>0 && col>0){
            //if the row/column left out, no circle --
            //we just iterate through this
            if(row == 1){
                for(int i=0; i<col; ++i){
                    result.add(matrix[x][y++]);
                }
                break;

            }else if(col==1){
                //only one column left out
                for(int i=0;i<row;++i){
                    result.add(matrix[x++][y]);
                }
                break;
            }
            //below, process a circle
            //top ->>>>> right
            for(int i=0; i<col-1;i++){
                result.add(matrix[x][y++]);
            }
            //right ->>>>>>>> down
            for(int i=0; i<row-1;++i){
                result.add(matrix[x++][y]);

            }
            //down ->>>>>>>>left
            for(int i=0; i<col-1; ++i){
                //index one is for nth row
                //index two is for nth col
                result.add(matrix[x][y--]);
            }
            //left->>>>>>>>>>>>>up
            for(int i=0;i<row-1;++i){
                //do not be silly this is matrix
                //where the x and y is different from x and y in
                //mathmatic  here x is just the y!!!!
                result.add(matrix[x--][y]);
            }
            x++;
            y++;
            row=row-2;
            col=col-2;
        }
        return result;
    }
}
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length == 0){
            return;
        }
        int m = matrix.length;
        int[][] result = new int[m][m];

        for(int i=0; i<m; i++)
            for(int j=0; j<m; j++)
                result[j][m-1-i] = matrix[i][j];
       for(int i=0; i<m; i++)
            for(int j=0; j<m; j++)
                matrix[i][j] = result[i][j];
    }
}
public class Solution {
    public String multiply(String num1, String num2) {
        if(num1.isEmpty() || num2.isEmpty()){
            return "";
        }
1. 首先要注意num1[i] * num2[j]的结果应该加到ret[i+j]的位置上。
2. 其次要注意ln 17不能遗漏最高位的进位，由于此时ret中该位为0，所以只需要将carry转为字符即可。
3. 最容易遗漏的corner case是ln 22-24。如999*0 = 0000，此时需要去掉开始的0，但又需要保留最后一个0。
/*
```

```
直接乘会溢出，所以每次都要两个single digit相乘，最大81，不会溢出。
比如385 * 97，就是个位=5 * 7，十位=8 * 7 + 5 * 9 ，百位=3 * 7 + 8 * 9 …可以每一位用一个Int表示，存在一个int[]里面。
这个数组最大长度是num1.len + num2.len，比如99 * 99，最大不会超过10000，所以4位就够了。
这种个位在后面的，不好做（10的0次方，可惜对应位的数组index不是0而是n-1），
所以干脆先把string reverse了代码就清晰好多。最后结果前面的0要清掉。
*/
        //* or + we need to reverse the string at first
        String n1 = new StringBuilder(num1).reverse().toString();
        String n2 = new StringBuilder(num2).reverse().toString();

        int[] d = new int[n1.length() + n2.length()];
        //we just simply
        for(int i=0; i<n1.length(); i++){
            int a = n1.charAt(i) - '0';
            for(int j = 0; j<n2.length();j++){
                int b=n2.charAt(j) - '0';
                //the whole value is stored such as 9*9=81 is stored here
                //store the whole result here we need !!!!
                //d[i+j] = a*b;////!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!not good!!!!1
                d[i+j] = d[i+j] + a*b;
            }
        }

        StringBuilder sb = new StringBuilder();
        for(int i=0; i<d.length;i++){
            int digit = d[i] %10;//the value is mod
            int carry = d[i]/10; //the carry is devie
            //here we just use insert to reverse the result string
            sb.insert(0,digit);
            //we need to put the carry in the next bit
            if(i<d.length-1){
                //next bit we need to caryr
                d[i+1] = d[i+1]+carry;
            }
        }

        while(sb.length()>0 && sb.charAt(0) == '0'){
            sb.deleteCharAt(0);
        }
        return sb.length() == 0 ? "0":sb.toString();
    }
}
public class Solution {
        //Two pointers problem -- Container with most water
    public int maxArea(int[] height) {
        //if we are calculating each pair and then get the max pair
        //this is Cn,2 = n(n-1)/2 -> time complexity = O(n^2)
        //however we use two pointers to sandwich left pointer is going to right
        //and right pointer is going to left -- we just traverse the array once -> O(n)
        //space complexity -- auxilary : we just use two pointers O(1)
        if(height == null || height.length ==0){
            return 0;
        }
        int l = 0, r = height.length-1;
        int max=0;//we need to update this continuously
        while(l<r){
            int limit = Math.min(height[l], height[r]);//we are finding the lowest vertical line which is the limit factor
            int cur_area = limit * (r-l);
            max = Math.max(max, cur_area);

            if(height[l] < height[r]){
                l++;//the l is the limiting factor
            }else{
                r--;//r is the limiting factor we find to the left a higher one
                //so that increase our area as possible as we could
            }
        }
       return max;
    }
}
public class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
            if (numRows <= 0)
                    return result;
            ArrayList<Integer> pre = new ArrayList<Integer>();

            pre.add(1);
            result.add(pre);
            for (int i = 2; i <= numRows; i++) {
                    ArrayList<Integer> cur = new ArrayList<Integer>();
```

```
                cur.add(1); //first
                for (int j = 0; j < pre.size() - 1; j++) {
                        cur.add(pre.get(j) + pre.get(j + 1)); //middle
                }
                    cur.add(1);//last
                result.add(cur);
                    pre = cur;
        }
            return result;
    }
}
         [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]，其第i行恰好为 (a + b)^i 的展开系数
public ArrayList<Integer> getRow(int rowIndex) {
    ArrayList<Integer> result = new ArrayList<Integer>(rowIndex + 1);
    for (int i = 0; i <= rowIndex; i++) {
      result.add(0);
    }
    result.set(0, 1);//the head is always unchanged
    for (int i = 1; i <= rowIndex; i++) {
      result.set(i, 1);//always put the tail in /the end of the row postion = rowNumber/
      for (int j = i - 1; j > 0; j--) {
        //we need to add from end to front!!! this is efficient
        result.set(j, result.get(j) + result.get(j - 1));
      }
    }
    return result;
}
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        //这道题是经典题，我在微软和YELP的onsite和电面的时候都遇到了.
        //从右上角开始，比较target 和 matrix[i][j]的值. 如果小于target，则该行不可能有此数，  所以i++; 如果大于target，则该列不可能有此数，所以j--.
        if(matrix.length ==0 || matrix == null){
            return false;
        }

        //search from the right up corner
        //here we just need to traverse o(m+n)
        int row =0;
        int col = matrix[0].length-1;

        //inside the board
        while(row<=matrix.length-1 && col >=0){
            if(matrix[row][col] == target){
                return true;
            }else if(matrix[row][col] < target){
                //could not be on this row
                row++;
            }else{
                //could not be on this col
                col--;
            }
        }

        //out of the search
        return false;
    }
}
```