Table of Contents *generated with DocToc*

Hard (91)

0. Count of Smaller Number before itself.java** Level: Hard Tags: []

Very similar to Count of Smaller Number. The actual value is used to form the segment tree, and the leaf is stored (count of smaller number).

Trick: Query first, then modify.
Each time Query, A [i] has not been added to the Segment Tree, and A [i + 1, ... etc] has not been added yet.
Then naturally it is coutning smaller number before itself.
Tricky!

Also note:
In modify: Check root.start <= index and index <= root.end. It was ignored in the past. You can also write this later.
(In fact, it is Make sense, which is to check the index and root.left or root.right more strictly)

---

1. Kth Smallest Sum In Two Sorted Arrays.java** Level: Hard Tags: []

Use priority queue. Each time the smallest expansion, shift. X + 1, or y + 1:
Because x and y are the smallest in Min at the moment. So the next smallest is either (x + 1, y), or (x, y + 1).

Just poll () one every time, just put 2 new candidates in. Note that this approach will be repeated, for example the example (7,4) will appear twice. Block it with a HashSet.

Note that the uniqueness of HashSet can be solved by using an "x, y" string.

---

2. LFU Cache.java** Level: Hard Tags: [Design, Hash Table]

#** Hash Table -See thoughts specifically, use map in several different ways

- regular object map : map of <key, item>, where item : {int val; int count}
- Use a Map<frequency count, doubly-linked node> to track the frequency
- Track constant capacity, and minimum frequency
- Every get(): update all frequency map as well
- Every put(): update all frequency map as well, with optional removal (if over capacity)

- Original post: http://www.cnblogs.com/grandyang/p/6258459.html
- TODO: one doubly linked list might be good enough to replace below:
- frequency list map : map of <frequency count, List>, where the list preserves recency
- item location in frequency map : map of <key, int location index in list>:
- index relative to the item in a particular list, not tracking which list here

---

3. Prefix and Suffix Search.java** Level: Hard Tags: [Trie]

---

4. Remove Node in Binary Search Tree.java** Level: Hard Tags: [BST]

Method 1: Brutle a little. Find the target and target's parent.
When removing the target, rearrange the child nodes of the target into a new BST: inorder traversal, build tree based on inorder traversal list.

Method 2: Analyze the rules, first find the target and parent, and then move the children nodes when the target is removed according to the nature, to ensure that it is still BST.

---

5. Subarray Sum II.java** Level: Hard Tags: [Array, Binary Search, Two Pointers]

---

6. k Sum.java** Level: Hard Tags: [DP]

DP. How the formula comes to mind needs to be re-understood.

dp[i][j][m]: # of possibilities such that from j elements, pick m elements and sum up to i. i: [0~target]

dp[i][j][m] = dp[i][j-1][m] + dp[i - A[j - 1]][j-1][m-1] (i not included) (i included)

---

7. Copy Books.java** Level: Hard Tags: [Binary Search, DP, Partition DP]

Give a list of books pages [i], k people, pages [i] represents the number of pages in each book. K people start to copy at the same time from different points.

Q. When can I copy it as soon as possible?

#** Partition DP -The first step is to understand the problem required by the title: the first k individuals copy n books and find the least amount of time; it can also be translated into: n books, let k individuals copy, that is, split into k segments. -Finally, dp [n] [k] is required. On: int [n + 1] [k + 1]. -Principle: -1. Consider the last step: In [0 ~ n-1] book, the last person can choose to copy 1 book, 2 books ..... n books, the results of each cutting method are different -2. Discuss the situation of the kth person, looping at j = [0 ~ i]. And the slowest case when looping j determines the result of the kth person (barrel principle): Math.max (dp [j] [k-1], sum). -3. Among them: dp [j] [k-1] is the result of [k-1] individual reading the j book, which is also known as the previous step. Here the loop considers that the kth person is different j kind of previous step :) -4. The result of the loop is that dp [i] [k] = Math.min (Math.max (dp [j] [k-1], sum [j, i]), loop over i, k, j = [i ~ 0])

- Time: O (kn ^ 2), space O (nk)

Init

-Init: dp [0] [0] = 0, 0 people 0 books -Use of Integer.MAX_VALUE: -When i = 1, k = 1, Expression: dp [i] [k] = Math.min (dp [i] [k], Math.max (dp [j] [k-1], sum)) ; -There is only one case that works: i = 0, k = 0, exactly 0 people copy 0 book, dp [0] [0] = 0. -In other cases, i = 1, k = 0, 0 people read 1 book, it cannot happen: so use Integer.MAX_VALUE to break Math.max, and maintain ridiculous values. -When the case of i = 0, k = 0 is discussed, the above equation will calculate dp [i] [k] according to the actual situation -The init of this question is very important and tricky

Calculation order

-k people, need a for loop; -k people, starting from copy1 book, 2, 3, ... n-1, so i = [1, n], need a second for loop -On each i, the cutting method can be [0 ~ i], we have to calculate each worst time

Scrolling array

-[k] Only related to [k-1]

- Space: O(n)

#** Binary Search -Based on: How much time does each person spend doing binary search: How long does it take each person to complete it in K people with the least amount of time? -The range of time variable is not index or page size. It is [minPage, pageSum] -Pay attention to 3 cases when validating: people are enough to use k> = 0, people are not enough so the ending is reduced to k <0, and there is a time (the time spent by each person) is less than the current page, return -1

- O(nLogM). n = pages.length; m = sum of pages.

8. Scramble String.java** Level: Hard Tags: [DP, Interval DP, String]

-Give two strings S, T. Check if they are scramble string. -scramble string definition: string can be split into binary tree form, that is, cut into substring; -After rotating a node that is not a leaf, a new substring is formed, which is the scramble of the original string.

#** Interval DP Interval -Dimension reduction strike, split, dp by length. -dp [i] [j] [k]: array S starts at index i, T starts at index j, is a substring of length k, is it a scramble string

Break down

-After two halves of everything, look at two cases:, or not rotate the two halves. For these substrings, verify whether they are scrambled. -Two halves without rotation: S [part1] corresponds to T [part1] && S [part2] corresponds to T [part2]. -Rotate the two halves: S [part1] corresponds to T [part2] && S [part2] corresponds to T [part1].

Initialization

-When len == 1, it can't be rotated, that is, to see if the corresponding indexes of S and T are equal. -Initialization is very important. It 's amazing. This initailization lays the foundation for DP, and the result is calculated with a mathematical expression. -input s1, s2 are hardly used in the main content of the entire problem, but only used in the initialization. -More details, see answer

---

9. Interleaving String.java** Level: Hard Tags: [DP, String]

Double-sequence DP, consider from the last point. At the end of the split problem, consider the association with s1, s2 subsequence.

Seeking existence, boolean

---

10. Edit Distance.java** Level: Hard Tags: [DP, Double Sequence DP, Sequence DP, String]

time: O (MN) Space: O(N)

Two strings, A must be B, you can insert / delete / replace to find the smallest change in operation count

#** Double Sequence -Consider the end of two strings, index? S [i], t [j]: If you need to make these two characters the same, you might use the three operations given in the title: insert / delete / replace? -Calculate worst case first, 3 operation count + 1; then compare match case. -Note that when i or j is 0, the steps that become another number can only be fully changed. -First step, space time is O (MN), O (MN) -Rolling array optimization, space O (N)

Detail analysis

- insert: assume insert on s, #ofOperation = (s[0 ~ i] to t[0 ~ j-1]) + 1;
- delete: assume delete on t, #ofOperatoin = (s[0 ~ i - 1] to t[0 ~ j]) + 1;
- replace: replace both s and t, #ofOperatoin = (s[0 ~ i - 1] to t[0 ~ j - 1]) + 1;
- dp [i] [j]? represents the nature of two sequences:? s [0 ~ i]? The minimum operation count required to convert to s [0 ~ j]
- init: When i == 0, dp [0] [j] = j;? + j characters each time; Similarly, when j == 0, dp [i] [0] = i;
- And dp [i] [j] has two cases: s [i] == t [j] or s [i]! = T [j]

When to initialize

-This judgment depends on experience: if you know that initialization can be done together in a double for loop, then you can keep doing that -This belongs to what is needed, initialize what -When doing space optimization afterwards, you can easily do rolling array on 1st dimension

#** Search -It can be done, but it is not recommended: this question needs to find min count, not search / find all solutions, so search will be more complicated, killing chickens.

---

11. Distinct Subsequences.java** Level: Hard Tags: [DP, String]

Double Sequence DP:

0. DP size (n + 1): find the result of the previous nth, then the dp array needs to open n + 1, because the end needs to return dp [n] [m]

1. Initialize dp [0] [j] dp [i] [0] in the for loop

2. Rolling array is optimized to O (N): If dp [i] [j] is in the for loop, it is a good replacement for curr / prev

12. Ones and Zeroes.java** Level: Hard Tags: [DP]

Still Double Sequence, but consider the third state: the amount of string array given. So opened a 3-dimensional array.

If you use a scrolling array to optimize space, you need to put the for loop that you want to scroll to the outermost, not the innermost. Of course, this third bit of definition is not a big problem in dp [] [] [].

Also, pay attention to calcualte zeros and ones outside, saving time and complexity.

---

13. Word Break II.java** Level: Hard Tags: [Backtracking, DFS, DP, Hash Table, Memoization]

Find all word break variations, given dictionary

利用 memoization: Map<prefix, List<suffix variations>>

#** DFS + Memoization

- Realize the input s expands into a tree of possible prefixes.
- We can do top->bottom(add candidate+backtracking) OR bottom->top(find list of candidates from subproblem, and cross-match)
- DFS on string: find a valid word, dfs on the suffix. [NO backtraking in the solution]
- DFS returns List: every for loop takes a prefix substring, and append with all suffix (result of dfs)
- IMPORANT: Memoization: Map<prefix, List<suffix variations>>, which reduces repeated calculation if the substring has been tried.
- Time O(n!). Worst case, permutation of unique letters: s= 'abcdef....', and dict=[a,b,c,d,e,f...]

#** Regular DPs -Two DPs are used together to solve the problem of timeout: when a invalid case 'aaaaaaaaa' occurs, isValid [] stops dfs from occuring -1. Isword [i] [j], subString (i, j) exist in dict? -2. Use isWord to speed up isValid [i]: Can [i ~ end] find a reasonable solution from dict?
-View i from the end: Because we need to test isWord [i] [j], j> i, and we observe the interval [i, j];
-The part of j> i also needs to be considered, we also need to know isValid [0 ～ j + 1]. So isValid [x] is a DP indicating whether [x, end] is valid this time.
-i is from the end to 0, probably because it is considered that isWord [i] [j] is within [0 ~ n], so the numbers are reversed and the coordinates are easier to figure out.
-(Looking back at Word Break I, there is also a practice of coordinate inversion) -3. dfs uses isValid and isWord for ordinary DFS.

#** Timeout Note -Regarding regular solution: Without memoization or dp, 'aaaaa .... aaa' will repeatedly calculate the same substring -Regarding double DP solution: Set.contains (...) is used in

Word Break, i is 0 in isValid. However, contains () itself is O (n); intead, use an isWord [i] [j], judge whether i ~ j exists in dictionary based on O (1)

---

14. Minimum Window Substring.java** Level: Hard Tags: [Hash Table, String, Two Pointers]

Basic idea: use a char [] to store the frequency of the string. Then 2pointer, end go to the end, and continue to validate. If it meets the process as result candidate.

HashMap is a bit more complicated to write than char [], but more generic

---

15. Longest Substring with At Most K Distinct Characters.java** Level: Hard Tags: [Hash Table, Sliding Window, String]

Large cleaning O (nk)
Once map.size> k, erase the char at the beginning of the longest string (marked by pointer: start)
Once a char is to be cleared, the char between 1st and last appearance of this char must be cleaned from map

---

16. Find Minimum in Rotated Sorted Array II.java** Level: Hard Tags: [Array, Binary Search]

A topic that requires rigorous thinking. Because duplicates cause constant translation, the time complexity is ultimately O (n) So it is better to scan it directly and give the answer.

But still write a Binary Search, but the worst result is O (n)

---

17. Number of Islands II.java** Level: Hard Tags: [Union Find]

给一个island grid[][], and list of operations to fill a particualr (x,y) position.

count # of remaining island after each operation.

#** Union Find, model with int[] -After converting the board into a 1D array, you can use union-find to determine it. -Use int [] father's unionFind, need to convert 2D position into 1D index. This is relatively clean -When judging, one step is taken in each of the four directions to determine whether it is the

same Land.-Each time the operator walks, it will count ++. If it is found to be the same island, count---The count addition and subtraction are all placed in UnionFind's own function, which is convenient for tracking. Just give a few helper functions.

- Time: O (k * log (mn))

#** Union Find, model with Hashmap -Union-find with HashMap.

#** Note:

- Proof of UnionFind log(n) time: https://en.wikipedia.org/wiki/Proof_of_O(log*n)_time_complexity_of_union%E2%80%93find

---

18. Word Search II.java** Level: Hard Tags: [Backtracking, DFS, Trie]

Give a string of words, and a 2D character matrix. Find all the words that can be formed. Condition: 2D matrix can only be positioned next to each other.

#** Trie, DFS -Compared with the previous implementation, there are some places that can be optimized: -1. During Backtracking, you can mark on board [] [] instead of opening a visited [] [] -2. You don't need to implement all the equations of the trie, you don't need it: only insert is needed here.-Common trie questions will let you search for a word, but here is a board, see if each letter of the board can come out of a word.-That is: the search here is written by hand, not the traditional trie search () funcombination -3. When there is end in TrieNode, the string word is stored, which means the end. When the word = null is used up, the problem of repeated search is just truncated.

About Trie

- Build Trie with target words: insert, search, startWith. Sometimes, just: `buildTree(words)` and return root.
- Still do DFS on the board matrix.
- no for loop on words. Directly to board DFS:
- Each layer will have an up-to-this-point string. Check if it exists in the Trie. Use this to judge.
- If it does not exist, you do not need to continue DFS.
- Trie solution time complexity, much better:
- build Trie: n * wordMaxLength
- search: boardWidth * boardHeight * (4^wordMaxLength + wordMaxLength[Trie Search])

#** Regular DFS

- for loop on words: inside, do board DFS based on each word.
- Time cpmplexity: word[].length * boardWidth * boardHeight * (4^wordMaxLength)

#** Previous Notes

- Big improvement: use boolean visited on TrieNode!

- Don't use rst.contains (...), because this is O (n) and timeout in leetcode (lintcode can pass)!
- In the Trie search () method, mark any visits.

---

19. Word Squares.java** Level: Hard Tags: [Backtracking, Trie]

Can open Trie class, which uses TrieNode. Open Trie (words) can be directly initalized with for loop A TrieNode can have a List startWith: record all strings that can reach this point: a bit like a tree, ancestor-shaped storage.

God operates: According to the nature of square, if list of words is selected, set int prefixIndex = list.size (). Take all the words [prefixedIndex] in the list and add them together to make the prefix of the next word candidate.

Image a bit: list = ["ball", "area"]; prefixIndex = list.size(); ball[prefixIndex] = 'l'; area[prefixIndex] = 'e'; //then candidatePrefix = ball[prefixIndex] + area[prefixIndex] = "le"; Here you can use the findByPrefix function of Trie. At each point, all the dates that can be generated by this point are stored. At this point, try all candidate: dfs

I can think of this inverted structure to store prefix candidates in Trie, this idea is very worth thinking about.

---

20. Trapping Rain Water.java** Level: Hard Tags: [Array, Stack, Two Pointers]

There are many ways to solve this problem. #** method 1 Array, maintaining a left-hand highest wall array, and a right-hand highest strength array. For each index, the maximum water column that can be stored vertically is determined by the left and right highest walls: min(leftHighestWall, rightHighestWall) - currHeight.

#** Method 2 The optimization above method 1, two pointers, still find the highest left and highest right. O (1) space. The idea used in method 3 is the same: the entire structure is divided by the highest bar in the middle: The left is calculated as maxLeft, and the right is calculated as maxRight.

#** Method 3

2 Pointers, double-sided pinch:

1. Find the index of the highest bar in the middle

2. Swipe to the center on both sides: add (topBarIndex-currIndex) * (elevation from previous index) each time. That is, add one horizontal bar at a time.

3. Every time you want to subtract the height of the block itself

#** Method 4 The main idea is the same as Method 3: On the basis of the downhill slope, the bottom has been stacked with stacks. Before finally encountering the ascent, at this time the bottom can be used to compare with all the downhill indexes accumulated before the stack, which is the water that is different from their height. The idea of using a stack to record downhill, and then dig to the end with a while loop is great.

---

21. Largest Rectangle in Histogram.java** Level: Hard Tags: [Array, Monotonous Stack, Stack]

Give n bars to form a histogram. Find the rectangle with the largest area that can be found in this row of histograms.

Thinking: Finding the area of a rectangle is nothing more than finding two indices, then the length of the bottom edge * height.

#** Monotonous Stack -The main point is to maintain a monotonically increasing stack according to the nature of the rectangle in the Histogram. -When loop over indexes: -If the height is> = previous peek (), then for that peek, it means, go down, keep going higher, the previous peek can always continue to bottom -When can't I buy a bottom? When is there a downward trend? -At this time, not all previous peeks are calculated, but all previous peeks larger than the current height are considered. -Find all the rectangles from the peek to the current height: stack.pop () -In the process of stack.pop (), the current height is not counted, because it needs to be retained in the next round, and the current index is added to the stack. -Why use stack? Because you need to know the continuously increasing peek, stack.peek () O (1), easy to use In fact, instead of stack, you can record all heights in other ways, but it is inconvenient to find peek by O (n)

#** Knowledge -Understand how monotonous stack is maintained -Maintaining a monotonous stack is required for the problem, not the nature of the stack itself.It is a clever use of stack.peek () O (1).

---

22. Find Peak Element II.java** Level: Hard Tags: [Binary Search, DFS, Divide and Conquer]

2Dmatrix, the value inside has some increasing and decreasing characteristics (the details are longer, see the original question). The goal is to find the peak element

peak: greater than the point value in the surrounding 4 directions

#** DFS

Fundamental

-We can't locate (x, y) accurately at one go, but we can find the peak of 1D array in another row / col. -Based on this point, move in the remaining two directions -1. In the middle line i = midX, find y where peak is. -2. In the middle column j = midY, find the x where the peak is. (It is possible to find the y before the strong override, that is, give up the peak of that line and find the peak on midY) -3. According to the 4 neighbor check (x, y) of (x, y) whether peak (x, y), if not, move one block like a higher position -4. According to the previously calculated midX, midY divide the board into 4 quadrants, and continue to find in each -This question LintCode did not do it, so the idea is correct, but the answer has not been verified again.

Pruning / splitting quadrant

-Just find a peak in row / col every time! -Finding this point is equivalent to cutting the board in half. -Then, compared with the remaining two adjacent positions, I know where the bigger one is, and where to find the peak, that is, I cut the second knife again. -When cutting the second knife, also move (x, y) to the quadrant to be taken. DFS -Cut according to mid row:

- http://www.jiuzhang.com/solution/find-peak-element-ii/#tag-highlight-lang-java
- http://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf

time complexity

-Each level is halved

- T(n) = n + T(n/2) = n + n/2 + n/4 + ... + 1 = n(1 + 1/2 + ... + 1/n) = 2n = O(n)

#** Binary Search

- EVERYTHING
- O (nLogN)

---

23. Palindrome Pairs.java** Level: Hard Tags: [Hash Table, String, Trie]

Obvious's method is to try it all, and judge, it becomes O (n ^ 2) * O (m) = O (mn ^ 2). O (m): isPalindrome () time.

Of course not, then it depends on O (nlogN), or O (n)?

#** Method 1: Properties of Hash Table + Palindrome. Compound. O (mn)

Ideas

-Each word can be split into front + mid + end. If this word + other words can form palindrome, that is to say -Cut off (mid + end), front.reverse () should be in words []. -Cut off (front + mid), end.reverse () should be stored in words []. -We use HashMap to store all <word, index>, and then reverse, just find a match.

Corner case

-If there is an empty string "", then it can be paired with any palendrome word, and it can be converted back and forth according to the position to make 2 distinct indexes. -This has the logic of if (reverseEnd.equals (" ")) {...}. -Note: Although the two for loops that deal with beheading / chopping are repeating records based on the empty string, But because "" itself cannot be used as a starting point, overall will only be recorded once when paired with other palendrome.

#** Method 2: Trie Still have to do that.

---

24. Maximal Rectangle.java** Level: Hard Tags: [Array, DP, Hash Table, Stack]

#** Method 1: monotonous stack Decomposed, it is actually 'Largest Rectangle in Histogram', but here you have to build your own model heights. The rectangle in a 2D array is also finally made with height * width. The clever thing is, treat each line as the bottom edge, and calculate the height of the bottom edge to the top: -If there is a value == 0 on the bottom edge, then it is counted as no height (the rectangle is used as the bottom edge, and the position of value == 0 is the sky tower, it cannot be used) -If value == 1 on the bottom edge, then add the above height to make a histogram

If you look at specific examples, some rows seem to be calculated in vain, but there is no way. This is a search process, and the optimal solution will eventually be compared.

#** Method 2: DP Coordinate DP?

25. Longest Increasing Path in a Matrix.java** Level: Hard Tags: [Coordinate DP, DFS, DP, Memoization, Topological Sort]

mxn's matrix, find the longest increasing sequence length. Here the default continuous sequence.

-It is not possible to make a circle, so visit (x, y) cannot go. -Cannot walk in oblique direction, only walk up, down, left and right -Can't do according to coordinate DP, because the calculation order can go in 4 directions. -In the end, all nodes must be visited, so DFS search is more appropriate.

#** DFS, Memoization

- 简单版: longest path, only allow right/down direction:
- dp[x][y] = Math.max(dp[prevUpX][prevUpY], or dp[prevUpX][prevUpY] + 1) ; and compare the other direction as well
- This problem, just compare the direction from dfs result
- DFS has too many double calculations; memoization (dp [] [], visited [] []) eliminates double calculations
- initialize dp [x] [y] = 1, (x, y) counts itself as a cell in path
- dfs (matrix, x, y): check 4 neighbors (nx, ny) of (x, y) each time, if they are increasing to (x, y), then consider and compare:
- Maht.max(dp[x][y], dp[nx][ny] + 1); where dp[n][ny] = dfs(matrix, nx, ny)
- top level: O (mn), try to start from each (x, y)
- O(m * n * k), where k is the longest path

#** Topological sort Not done yet

---

26. Coins in a Line III.java** Level: Hard Tags: [Array, DP, Game Theory, Interval DP, Memoization]

LeetCode: Predict the Winner

Still 2 people take n coins, and coins can have different values.

But this time the player can take from any side, but not restricted from one side. Will the first mover win?

#** Memoization + Search -Like Coins in a Line II, MaxiMin's idea: Find the maximum of my disadvantages - dp [i] [j] represents the sum of values that players can take in the [i, j] interval - Similarly, sum [i] [j] represents the sum of values between [i] and [j] -Worst case for opponent, best case for first mover:

- dp[i][j] = sum[i][j] - Math.min(dp[i][j - 1], dp[i + 1][j]);

- You need to search here, draw a tree to see how it is segmented according to before and after fetching.

#** Game + Interval DP, Interval DP -Because it looks at the interval [i, j], it can be thought of as the interval DP -This method needs a review, and it is related to the inference of mathematical expressions: S (x) =-S (y) + m. Refer to the following formula for derivation. -dp [i] [j] indicates that from index (i) to index (j), the maximum value that the first player can get is different from the opponent's number. That is S (x). -One of S (x) = dp [i] [j] = a [i]-dp [i + 1] [j] -There are two cases where m is at the beginning and m is at the end:

- dp[i][j] = max{a[i] - dp[i + 1][j], a[j] - dp[i][j - 1]}
- len = 1, integral is values [i]
- Finally judge dp [0] [n]> = 0, the difference between the maximum number sum is greater than 0, and you win.
- Time / space O (n ^ 2)

Formula Derivation

-S (x) = X-Y, find the difference between the largest number and sum, where X and Y are the total score of player X and the total score of player Y. -For player X: If the maximum value of S (x) is greater than 0, you win; if the maximum value is less than 0, you must lose. -Player Y: S (y) to indicate the difference between the largest number and sum for Y. S (y) = Y-X -According to S (x), if you take out a number m from the number and X, that is X = m + Xwithout (m)

- S(x) = m + Xwithout(m) - Y = m + (Xwithout(m) - Y).
- If we simply remove m from the global, then S (y '') = Y-Xwithout (m)
- Then calculate it: S (x) = m + (Xwithout (m)-Y) = m-(Y-Xwithout (m)) = m-S (y '')
- In this question, when we model X and Y, they are actually dp [i] [j], and the difference is first-hand / last-hand.
- Apply the formula, a certain S (x) = a [i]-dp [i + 1] [j], which is m = a [i], and S (y '') = dp [i + 1] [j]

Note

-If you consider calculating the maximum value between the first hand [i, j], then two arrays may be needed, and finally used to compare the score size of the first hand and the opponent => then more dimensions are needed. -The number difference we consider here just happens to make people not need to calculate the total score of the first mover, very clever. -Trick: Using the difference formula, the derivation is a bit difficult to think of.

Interval Dynamic Programming

-Find the properties within the [i, j] interval: dp [i] [j] The subscript indicates the interval range [i, j] -Sub-question: behead, tail, behead -loop should be based on the length of the interval -template: consider len = 1, len = 2; when setting i must be i <= n-len; when setting j, j = len + i-1;

---

27. Burst Balloons.java** Level: Hard Tags: [DP, Divide and Conquer, Interval DP, Memoization]

A volleyball, each ball has a value, each time you break one, you will score: left * middle * right value. Find, how to tie, the maximum?

TODO: Need more thoughts on why using dp[n + 2][n + 2] for memoization, but dp[n][n] for interval DP.

#** Interval DP -Because the regularity of the array changes, it is difficult to find the 'first burst'. On the contrary, which one is the last burst? -The last burst becomes a wall: separate the two sides, consider separately, the principle of addition; finally add the middle.

- dp[i][j] represent max value on range [i, j)
- Need to calculate dp[i][j] incrementally, starting from range size == 3 ---> n
- Use k to divide the range [i, j) and conquer each side.

Interval DP Three Axes:

-Split in the middle -Cut off head or tail -Range as the basis of iteration

Print the calculation process

- use pi[i][j] and print recursively.
- Print k, using pi[i][j]: max value taken at k

#** Memoization -In fact, it will be a DP that I think about afterwards -dp [i] [j] = max between balloons i ~ j. -Then which point to start the burst? Set to x. -For loop all points are taken as x, go to burst. -Each burst is cut into three parts: the left side can be recusive to find the maximum value of the remaining part on the left side + the middle 3 terms are multiplied + the right side is recursive to find the maximum value. -Note: This is Memoization, not pure DP -Because it is recursive, it is still a search, but memorize the requested value, saving processing

---

28. K Edit Distance.java** Level: Hard Tags: [DP, Double Sequence DP, Sequence DP, Trie]

Give a string of String, target string, int k. Find all the dates in the string array: change K times, can become target.

#** Trie EVERYTHING

#** Double Sequence DP

- Edit Distance的follow up.
- In fact, it is to change the function of minEditDistance and bring K for comparison.
- The main logic is exactly the same as Edit Distance.
- But LintCode 86% test case is timeout.
- Time O (mnh), where h = words.length, if n ~ m, Time is almost O (n ^ 2), which is too slow.

---

29. Paint House II.java** Level: Hard Tags: [DP, Sequence DP, Status DP]

time: O (NK ^ 2): space: (NK)

A row of n houses, each house can be painted in k colors, the price of each house is different, expressed by costs [] [].

costs [0] [1] means that the house with index 0 is painted and color 1.

Rule: two adjacent houses cannot be the same color

Seek: the least cost

#** DP -It's almost the same as Paint House I, but with more paint colors: k colors. -First consider simply using dp [i] to represent the minimum cost of the first i houses -But what colors dp [i] and dp [i-1] index will affect each other, it is difficult to discuss, so add state: the sequence DP is added to the state to 2D. -Consider the last bit, and the previous i-1 is limited by the color of the i bit, so when considering min dp [i], there is another layer of iteration. -Do dp [i] [j]: # cost for the first i houses, so you must first pick the cost of the (i-1) house, then find out the cost of the (i-2) house -K colors => O (NK ^ 2) -If not optimized, it is almost the same code as Paint House I

- Time O (NK ^ 2), space (NK)
- Rolling array: reduce space to O(K)

#** Note -The sequence type dp [i] represents the result of 'top i-1'. So dp is best set to int [n + 1] size. -However, the color is the state here, so it remains in j: [0 ~ k) -[[8]] This edge case. Can't run into for loop, so special handle.

#** Optimization Solution

- Team: O (NK)
- If it is known that two different minimum costs must be selected from the cost each time, then the minimum two are selected first, and there is no need to have a third for loop to find min
- Find each time in the series: min value except yourself, use min / min value
- Maintain 2 minimum values: minimum / secondary value.
- When calculating, if the index of the minimum value is not removed, the minimum value is given; if the index of the minimum value is removed, the next smallest value is given.
- Every loop: 1. calculate the two min vlaues for each i; 2. calcualte dp[i][j]

- How to think of optimization: Write the expression and see where you can optimize
- In addition, you can still roll array, reduce space complexity to O (K)

---

30. Best Time to Buy and Sell Stock III.java** Level: Hard Tags: [Array, DP, Sequence DP]

One more restriction than stock II: only 2 sell opportunities.

#** DP plus status -Sell only 2 times, split the sale into 5 status modules. -In state index 0, 2, 4: No stock is held. 1. Always in this state, max profit is unchanged; 2. Just sold, dp [i] [previous state] + profit -At state index 1, 3: Holding the stock. 1. Always in this state, daily profit. 2. Just bought, state changed, but no profit yet: dp [i] [previous state]

Partial profit

-Adding daily partial profit (diff) together, the final overall profit is the same. The only thing that is better is that you don't need to record the time of the middle buy. -When will profit accumulate? -1. The original stock is held. If there is no action, the status will not change and the profit diff will be accumulated. -2. The stock was sold, the status changed, and profit diff accumulated. -Note: Only when the state index: 0, 2, 4, that is, the stock is sold, can you accumulate profit

Rolling Array

-[i] only deals with [i-1], reduce space

- O (1) space, O (n) time

#** Looking for the peak -Find the peak; then look for another peak. -How about Optimize twice? Start looking for Max from both sides at the same time! (Awesome idea) -leftProfit is the maximum Profit at each i point from left to right. -rightProfit is the maximum profit at each point starting from point i to the end. -At point i, it is the leftProfit, and the rightProfit split point. At point i, leftProfit + rightProfit is added to find the maximum value. -Three O (n), or O (n)

---

31. Best Time to Buy and Sell Stock IV.java** Level: Hard Tags: [DP, Sequence DP]

There are int [] price of stock, up to k transactions. Seeking maximum profit.

#** DP -According to StockIII, it is not difficult to find that StockIV is to divide the state into 2k + 1 shares. Then the same code, transplant.

Note 1:

-If k is large and k> n / 2, then there can be at most n / 2 transactions in an array of length n -Then the problem is simplified to stockII, giving n arrays, unlimited transactions. -Note that the number of status is 2k + 1

- Time O(NK), Space O(2k+1) to store the status

Note 2:

-The final status is 'no stock' should be considered, make a for loop to compare max. -Of course, it is also possible to make a profit variable and keep comparing.

#** Method 2 -(previous notes, thinking about the first method is enough) -Remember to understand: Why did you sell and buy on day i-1, and you can make a transaction with the sale on day i?
-Because the price of daily transactions is fixed. So if you sell and buy, you are not selling! That's why you can merge. Be sensitive to prices.

- Inspired from here: http://liangjiabin.com/blog/2015/04/leetcode-best-time-to-buy-and-sell-stock.html

Local optimal solution vs. global optimal solution:

- local[i][j] = max(global[i − 1][j − 1] + diff, local[i − 1][j] + diff)
- global[i][j] = max(global[i − 1][j], local[i][j])
- local [i] [j]: On day i, profit of j-th transaction must be made on that day
- global [i] [j]: On day i, a total of j transactions have been profitable.

-The difference between local [i] [j] and global [i] [j] is that local [i] [j] means that there must be a transaction (sell) on day i.
-When the price on day i is higher than day i-1 (that is, diff> 0), then this transaction (buy on day i-1 and sell on day i) can be traded with day i-1 (Sell) merge into one transaction, that is local [i] [j] = local [i-1] [j] + diff;
-When the price on day i is not higher than day i-1 (that is, diff <= 0), then local [i] [j] = global [i-1] [j-1] + diff, and because diff <= 0, so it can be written as local [i] [j] = global [i-1] [j-1].
-(Note: + diff is not omitted in this solution below)

-global [i] [j] is the maximum profit we can make for a maximum of k transactions in the first i days, which can be divided into two cases:
-If there is no transaction (sell) on day i, then global [i] [j] = global [i-1] [j];
-If there is a transaction (sell) on day i, then global [i] [j] = local [i] [j].

---

32. Russian Doll Envelopes.java** Level: Hard Tags: [Binary Search, Coordinate DP, DP]

Matryoshka, here is represented by envelope. For a string, each [x, y] is the length and width of envelope. [[5,4], [6,4], [6,7], [2,3]].

Look at these sets of dolls, you can set a few at most.

#** DP: 1D Coordinate -Envelopes have no order, they are sorted first (mainly according to the first index) -Then observe: After sorting, it becomes 1D coordinate dynamic programming. -max number depends on the max value of the previous successful Russian doll + 1 -The previous index is unknown, so iterate to find the previous index. -The current state of index i depends on the state of previous index j, so iterate through two indexes.

- O(n^2)的DP, n = envelopes.length;

#** DP: 2D Coordinate -This method came up by myself, but the time complexity is too large, timeout -Mark the envelop on the 2D grid, and then like a robot, find the maximum count max in the bottom right corner. -count how many Russian dolls can be present -Two cases: current coordinate has no target, current coordinate has target -The current coordinate has no target: like robot moves, Math.max (dp [i-1] [j], dp [i] [j-1])

- 当下coordinate 有target: dp[i - 1][j - 1] + dp[i][j]
- timeout: O(n^2), n = largest coordinate.

---

33. Expression Tree Build.java** Level: Hard Tags: [Binary Tree, Expression Tree, Minimum Binary Tree, Stack]

Give a string of characters, which is the formula expression. Turn the formula into an expression tree

#** Monotonous Stack -Like Max-tree, https://leetcode.com/problems/maximum-binary-tree -A bottom-> top incremental stack is used: the lowest root is maintained as the smallest element. -This topic is Min-tree, the smallest on the head, Logic is the same as max-tree

- Space: O(n)
- Time on average: O(n).

#** Features -TreeNode: Use a TreeNode that is not the final result, store the weight, and use it for sorting -Weigh the symbols on the same level with the concept of base weight, numerical order -Each character is a node and has its own weight. Use a TreeNode to store the weight value, and use weight to determine: -1. (while loop) If node.val <= stack.peek (). NodeValue, change the current stack.peek () to left child. -2. (if condition) If the stack has residue, change the current node to stack.peek (). RightChild

---

34. Expression Evaluation.java** Level: Hard Tags: [Binary Tree, DFS, Expression Tree, Minimum Binary Tree, Stack]

Give a formula expression, array of strings, and evaluate the result.

#** DFS on Expression Tree -Calculate the value of expression: 1. Construct expression tree. 2. DFS calculation result

- Expression Tree: Minimum Binary Tree (https://lintcode.com/en/problem/expression-tree-build/)
- After building Min Tree, do PostTraversal.
- Divde and Conquer: first recursively find the size of left and right, then evaluate the symbol in the middle
- Time, Space O(n), n = # expression nodes

#** Note -1. When Handle number, if left && right Child is all Null, it must be our largest number node.
-2. If one child is null, then return another node.
-3. prevent Integer overflow during operation: Use a Long during the process, and the final result is cast back to int.

---

35. Convert Expression to Polish Notation.java** Level: Hard Tags: [Binary Tree, DFS, Expression Tree, Stack]

Give a string of characters to represent the formula expression. Convert this expression to Polish Notation (PN).

#** Expression Tree

- Expression Tree: Minimum Binary Tree (https://lintcode.com/en/problem/expression-tree-build/)
- After the Expression Tree is made according to the intent: After a Pre-order-traversal, you can record the Polish Notation
- This question is not given to 'ExpressionTreeNode', so TreeNode is regarded as the node we need, and it can be expanded to have left / right child.
- Note: label needs to be String. Although Operator is a char with length 1, the number can be multiple digits

---

36. Convert Expression to Reverse Polish Notation.java** Level: Hard Tags: [Binary Tree, DFS, Expression Tree, Stack]

Give a string of characters to represent the formula expression. Convert this expression to Reverse Polish Notation (RPN).

#** Expression Tree

- Expression Tree: Minimum Binary Tree (https://lintcode.com/en/problem/expression-tree-build/)
- After the Expression Tree is made according to the intent: Post-order-traversal can record Reverse Polish Notation
- This question is not given to 'ExpressionTreeNode', so TreeNode is regarded as the node we need, and it can be expanded to have left / right child.

---

37. Decode Ways II.java** Level: Hard Tags: [DP, Enumeration, Partition DP]

Given a string of numbers, you need to decode them into English letters. [1 ~ 26] Corresponding to the corresponding English letters. Find out how many ways you can decode.

The characters may be "*", which can represent [1-9]

DP -Multiplication principle -Same as decode way I, the principle of addition, and the cutting point: whether 1 digit or 2 digits is used to decode -Define dp [i] = how many ways to decode the first i digits. New dp [n + 1]. -Different situations: In each partition, if there is "*"*, it will extend many different possibilities in itself.

- 那么: dp[i] = dp[i - 1] * (#variations of ss[i]) + dp[i - 2] * (#variations of ss[i,i+1])

Features

-Ability to enumerate: specifically analyze where the '*' appears, enumerate numbers, basic skills. -Note !! The title says * in [1, 9]. (It will be harder if 0 ~ 9) -Understand the reason for taking MOD: the number is too large, take the mod to give the final result: In fact, with a mod as large as 10 ^ 9 + 7, most examples can pass. -After enumerating, the writing and thinking process of this topic is not difficult.

---

38. Palindrome Partitioning II.java** Level: Hard Tags: [DP, Partition DP]

Give a String s, find out how many cuts to use, so that each substring that is cut out is palindrome

#** Partition DP -Find minimum cut: Divide DP -dp [i]: How many knives to cut at least, so that the string of length i before the cut is all palindrome -You end up with dp [n], so int [n + 1] -Move the

cutter, see where to cut, index j in [0 ~ i] -Consider whether [j, i-1] is a palindrome string, and if so, then: dp [i] = min (dp [i], d [j] + 1). -note: It is estimated that when traversing j, the reverse traversal is also possible.

#** Computing Palindrome Optimization -Using the properties of palindrome, the situation of boolean palindrome [i, j] can be calculated. -Find an arbitrary mid point: -1. Assuming that the palindrome is an odd length, then mid is a separate character, and the characters [mid-1], [mid + 1] on both sides should be exactly equal. -2. Assuming that the palindrome is of even length, the characters at [mid] and [mid + 1] should be equal. -Do this palindrome [i, j]: whether the substring from character i to character j is palindrome -This will reasonably reduce the dimensionality of our problem, currently it is time: O (n ^ 2). -Otherwise, if we ask for palindrome once, that is n, it will become O (n ^ 3)

#** Previous Notes -Double for loop checks each substring string (i ~ j). If i, j are adjacent or the same point, then isPal; otherwise, (i + 1, j-1) between i and j must be isPal. -It seems that when checking i, j, how can I know (i + 1, j-1) pressed in the middle first? Actually not .. When j grows up slowly, all 0 ~ j substrings are checked. So isPal [i + 1] [j-1] must already know the result. -okay. Then if any of the above is true, that is to say isPal [i] [j] == true. Then we have to judge how many ways to cut to the end point of the loop parameter j in the first layer? -The idea is smooth: we naturally think that it would be better to add the cut before i plus what happened between i ~ j. -Anyway, j is not changed now, let 's see where i is and whether cut [i-1] is smaller / minimum; then +1 on cut [i-1] is over. -Of course, if i == 0, and i ~ j is isPal, then there is nothing to talk about, don't cut, 0 knife. -In the end, brush to cut [s.length ()-1], which is the last point. return is right.

---

39. Backpack III.java** Level: Hard Tags: [Backpack DP, DP]

For n different items, int [] A weight, int [] V value, each item can be used unlimited times

Ask the maximum value can be packed into a package of size m?

#** DP -Items can be used indefinitely, losing the meaning of last i, last unique item: because it can be reused. -So you can convert an angle: -1. Use i kinds of items, spell w, and satisfy the max value. Here, item i can be used unlimited times, so consider how many times K is used. -2. Although K can be infinite, it is also limited by k * A [i]: the maximum cannot exceed the size of the backpack. -dp [i] [w]: For the first i items, fill the w backpack, what is the maximum value?

- dp[i][w] = max {dp[i - 1][w - k*A[i-1]] + kV[i-1]}, k >= 0
- Time O (nmk)
- If k = 0 or 1, it is actually Backpack II: with or without

#** Optimization -Optimize time complexity, draw and find: -Calculated (dp [i-1] [j-k * A [i-1]] + k * V [i-1]) -In fact, the grid of dp [i] [jA [i-1]] on the same line has V [i-1] -So there is no need to loop over k times every time -Simplify: dp [i] [j] One of the possibilities is: dp [i] [j-A [i-1]] + V [i-1]

- Time O (mn)

#** Space optimized to 1-dimensional array -Draw a 2 rows grid according to the previous optimization -Found that dp [i] [j] depends on: 1. dp [i-1] [j], 2. dp [i] [j-A [i-1]] -Among them: dp [i-1] [j] is the settlement result of the previous round (i-1), it must be already calculated, ready to be used -However,

when we have i ++, j ++, and before row = i-1, col <j, all of them are not needed. -Dimension reduction and simplification: We only need to keep the dimension of weigth, and the dimension of i can be omitted: -(i-1) row is just to use the old value calculated before: each round, j = [0 ~ m], then dp [j] itself has the function of recording the old value. -Becomes 1 bit array -Focus of dimensionality reduction optimization: look at the left and right calculation direction of the two lines

- Time(mn). Space(m)

---

40. First Missing Positive.java** Level: Hard Tags: [Array]

Give a string of unordered numbers with negative numbers: find the first missing positive integer in this array

The missing positive integer is actually compared with [1, n].

#** Array analysis, index tricks -With a while loop, keep trying to get the number to the place -If index = nums [i] exceeds nums.length, of course, it will not move -Note: check val! = Nums [val], avoid infinitely loop -Test: Is nums [i] equal to i, if not, then find the result

#** Edge Case -If nums == null, in fact missing positive integer is naturally 1 -During validation, there may be no disconnected integers in the string, but the largest integer is in the first place (because the index exceeds the standard, it cannot be placed in the correct place) -At this time, n is placed at index 0. In fact, the next integer should be n + 1 -In the end, if the array is completely sorted, it is not lacking, and it also meets the conditions of the superscript, then the only next one is the first positive number outside the range of the array: n

---

41. N-Queens.java** Level: Hard Tags: [Backtracking]

N-Queen question, give numbers n, and nxn board, find all answers for N-queens.

#** Backtracking -Use dfs to find all situations, each iteration, pick the right point from the line, dfs - The selected points are added to the candidate list, remember to backtracking. -Each candidate needs validation, check if row, col, 2 diagnal is queen

#** validate n queue at certain (x, y) -1. array cannot have target row # -2. diagnal. Remember the formula:

- row1 - row2 == col1 - col2. Diagnal elelment.fail
- row1 - row2 == - (col1 - col2). Diagnal element. fail
- Draw a 3x3 board to test the 2 scanarios:
- (0,0) and (3,3) are diagnal
- (0,2) and (2,0) are diagnal

---

42. N-Queens II.java** Level: Hard Tags: [Backtracking]

Like N-Queens, not all results, but how many results are counted.

#** Backtracking -When list.size () == n, it means that a solution was found.

- 1. dfs function (List, n)
- 2. validate function

---

43. LRU Cache.java** Level: Hard Tags: [Design, Hash Table, Linked List]

#** Double Linked List -Using a special bidirectional ListNode, with head and tail, this greatly speeds up the process.
-The main thing that speeds up is the process of updating the ranking. Finding the item hashmap O (1), and doing subtraction and transposition are O (1)

- Overall O(1)

Be smart

-1. Head and tail are particularly clever: removing heads and tails, and adding heads and tails, are particularly fast.
-2. Use two-way pointers: pre and next. When you need to remove any node, just know which one to remove.
-Just patiently connect node.pre and node.next, and the node will be disconnected naturally.
-Once you know how to solve it, it is not very special, and it is not difficult to write the algorithm:

- moveToHead()
- insertHead()
- remove()

#** O (n) check for duplicates -timeout method, naive came an O (n) solution, and it turned out timeout.
-A map <key, value> stores the value. A queue to hold the rank.
-Every time there is an update, put the latest one at the end; every time you exceed the capaticity, kill the big head. Very simple, but it took too long to run and failed.

---

44. Binary Tree Maximum Path Sum.java** Level: Hard Tags: [DFS, DP, Tree, Tree DP]

Find max path sum, from any treeNode to any treeNode.

#** Kinda, Tree DP -Two cases: 1. combo sum: left + right + root; 2. single path sum

- Note1: the path needs to be continuous, curr node cannot be skipped
- Note2: what about I want to skip curr node: handled by lower level of dfs(), where child branch max was compared.
- Note3: skip left / right child branch sum, by comparing with 0. Less than 0, no need to record

#** DP Thoughts -tree gives us 2 branches, each branch is similar to dp [i-1], here is similar to dp [left], dp [right] -After finding dp [left], dp [right], combine with curr node. -Because it is looking for max sum, and can skip nodes, the global variable max is required -Each time dfs () returns must be a path that can continue continuously link ', so return one single path sum + curr value`.

#** DFS, PathSum object

- that just solves everything

---

45. Basic Calculator.java** Level: Hard Tags: [Binary Tree, Expression Tree, Math, Minimum Binary Tree, Stack]

Give an expression String to evaluate the value of the expression.

Expression strings include +,-, integers, opening and closing parentheses, and space.

#** Expression Tree -Expression Tree is a weight-based min-tree -Tree based on operation symbol + number: number is always in leaf, then symbol is tree node, brackets do not appear in tree -Use monotonuous stack to build this tree

Thinking points

- Understand Expression Tree
- Use stack to build the expression tree + understand the weight system
- Use post-order traversal to evaluate the tree
- Note that the number in input will not be a single digit, so a buffer is needed to store the number string
- For the practice of the entire topic, please refer to Expression Evaluation

---

46. Longest Consecutive Sequence.java** Level: Hard Tags: [Array, Hash Table, Union Find]

Give a string of numbers, unsorted, find the length of the sequence of consecutive elements in the string of numbers

#** HashSet -To see continuous elements, you must search for num ++, num---1. Need O (1) to find the element -2. Need to find num-1, num + 1.-If you open the array with min, max, it consumes space - Save with HashSet, use set.contains () to find num-1, num + 1 exists

- for loop. O(n)
- The while loop inside will generally not have O (n); once O (n), it also means that the set is cleared, and there will be no more inner while derivatives for the for loop.
- overall O (n) time complexity

#** Union Find -In the end, we need to calculate the total length of the connected elements. In fact, the elements are grouped together, and the connected groups are together, so I think of UnionFind -An `int [] size` is used here to help deal with the question of `parent` when merging: always go to a union with a large group -In the main function, there is a map to track, and each element is processed only once.

- union的内容: current number - 1, current number + 1
- https://www.jianshu.com/p/e6b955ca208f

Features

-Union Find seems easier to do on index

- 其他union find function: `boolean connected(a,b){return find(a) == find(b)}`

---

47. Serialize and Deserialize Binary Tree.java** Level: Hard Tags: [BFS, DFS, Deque, Design, Divide and Conquer, Tree]

Serialize and Deserialize Binary Tree

#** DFS, Divide and Conquer

Serilize

- Divide and conquer: Pre-order traversal to link all nodes together
- build the string data: use '#' to represent null child.

- the preorder string, can be parsed apart by `split(',')`

Deserialize

- Use a list (here we use `Deque` for the ease of get/remove in 1 function: remove())
- to take all parts of the parsed sring data: dfs on the Deque
- first node from the list is always the head
- '#' will be a null child: this should break dfs
- Deque is a global variable, so dfs(right child) will happen after dfs(left child) completes

#** DFS, Recursive [previous note]

- serilize: divide and conquer, pre-order traversal
- deserialize: slightly more complicated, use dfs. truncate input string each time:
- Keep dfs looking for left child, then right child until leaf is found.
- Use a StringBuffer to hold the string, because string is primitive, we need a pass reference here

#** BFS, Non-recursive -using queue. The idea is intuitive. level-order traversal. Save to a string. -When encountering a null child, instead of ignoring it directly, you assign an Integer.MIN_VALUE, and then mark as '#' -BFS requires track queue size, each time only a specific number of nodes

---

48. Count of Smaller Numbers After Self.java** Level: Hard Tags: [BST, Binary Indexed Tree, Binary Search, Divide and Conquer, Segment Tree]

Give a string of numbers nums [], find a new array result, where result [i] = # of smaller items on right of nums [i]

#** Binary Search -sort and insert into a new list, the new list is sorted -Traverse nums [] from the end i = n-1 -Each time insert nums [i] enters the list, it is # of smaller items on right side of nums [i] -Record result [i] every time -** Question : The binary search here is done with `end = list.size (); while (start <end) {...}`, can it be replaced with `end = list.size ()- 1` ?

#** Segment Tree based on actual value

- Build segment tree based on min/max values of array: set each possible value into leaf
- query(min, target - 1): return count # of smaller items within range [min, target - 1]
- Very similar to `Count of Smaller Number`, where segment tree is built on actual value!!
- IMPORTANT: goal is to find elements on right -> elements processed from left-hand-side can be removed from segment tree
- Use `modify(root, target, -1)` to remove element count from segment tree. Reuse function
- time: `n * log(m)`, where m = Math.abs(max-min). log(m) is used to modify() the leaf element

Segment Tree solution - tricky part:

- negative nubmer works oddly with mid and generates endless loop in build(): [-2, -1] use case
- build entire segment tree based on [min, max], where min must be >= 0.
- we can do this by adding Math.abs(min) onto both min/max, as well as +diff during accessing nums[i]

#** Binary Indexed Tree

- TODO, have code

---

49. Remove Duplicate Letters.java** Level: Hard Tags: [Greedy, Hash Table, Stack]

#** Hash Table, Greedy -count [] = int [256], no c-'a' required -boolean visited []: Once a letter has fixed its position, when it encounters it again, it skips the used character directly -If the tail letter can be made smaller, delete the tail and reconnect with the new letter (prerequisite: the removed letter will appear again after the set letter, set visited [tail] = false)

- Space: O(1) count[], visited[].
- Time: Go through all letters O(n)

#** Stack

- Use stack instead of stringBuffer: keep append/remove last added item
- However, stringBuffer appears to be faster than stack.

---

50. Expression Add Operators.java** Level: Hard Tags: [Backtracking, DFS, Divide and Conquer, String]

Give a number String, the numbers come from 0-9, give 3 operators +, -, *, see how to piece together, you can make the result target.

output 所有 expression

#** string dfs, use list to track steps (backtracking) -Related to string, it may be a little tedious to write -Numbers have dfs ([1,2,3 ...]) combination method -operator has [+, -, *] 3 combinations -Note 1: The multiplication sign must be treated specially, the numbers along the multiplication are passed, and when calculating the next product, sum-preProduct + product -Note 2: '01' is a skip number -Note 3: The first selected number does not need to be added, it is added directly

- Time: O(4^n), Space: O(4^n)

- T(n) = 3 * T(n-1) + 3 * T(n-2) + 3 * T(n-3) + ... + 3 *T(1);
- T(n-1) = 3 * T(n-2) + 3 * T(n-3) + ... 3 * T(1);
- Thus T(n) = 4T(n-1) = 4^2 * T(n - 1) = ... O(4^n)

#** String dfs, use string as buffer -The logic is the same, the code is shorter, but instead of doing a list, pass `buffer +" + "+ curr` directly -It is slightly slower because new strings are created each time. Same as Time complexity

---

51. Insert Interval.java** Level: Hard Tags: [Array, PriorityQueue, Sort]

#** Sweep Line -Interval teardown point, PriorityQueue rank point -Merge with count == 0 as the judgment point -Note, be sure to compare curr `px == queue.peek (). X` to ensure that all coincident points are processed by process:`count + = px` -PriorityQueue: O (logN). Scan n points, total: O (nLogn)

#** Basic Implementation -** sorted intervals by start point have been given here. -Directly find the seat where insert newInterval can be inserted. Insert

- 然后loop to merge entire interval array
- Because it is a list, it is convenient for `intervals.remove (i)`
- pre.end will be reasserted before remove to ensure that the removed node.end is captured
- O (n)

#** Also -Because interval has been sorted, I wanted to use Binary Search O (logn). -But to find the interval insert position, the final merge still uses O (n), so it is not necessary to use binary search

---

52. Shortest Palindrome.java** Level: Hard Tags: [KMP, String]

#** Divide by mid point, Brutle

- check (mid, mid+1), or (mid-1, mid+1).
- If the two position matches, that is a palindrome candidate
- Compare whether front string is a substring of end string
- O (n ^ 2)
- timeout on last case: ["aaaaaa....aaaacdaaa...aaaaaa"]

#** KMP

- EVERYTHING

---

53. K Empty Slots.java** Level: Hard Tags: [Array, BST, TreeSet]

题目解析后: find 2 number, that: 1. k slots between the 2 number, 2. no slots taken between the two number.

#** BST

- BST structure not given, use TreeSet to build BST with each node
- Every time find last/next inorder element
- `treeSet.lower(x)`, `treeSet.higher(x)`
- Once the positions are separated (k + 1), the subject conditions are met
- O(nlogn), good enough

#** Track slots of days

- Reverse the array, save days index into days[], where the new index is slot.
- days[i]: at slot i, which day a flower will be planted
- O (n)
- Needs to understand: http://www.cnblogs.com/grandyang/p/8415880.html

---

54. Count of Range Sum.java** Level: Hard Tags: [BST, Divide and Conquer, Merge Sort, PreSum]

TODO: Write the code + merge function

#** Divide and Conquer + PreSum + MergeSort -The algorithm is very powerful: presum [], then sum range [i, j] is equal to preSum [j + 1]-preSum [i] -Divide and conquer: Consider the results in [start, mid] range, and then the results in [mid, end] range. (MergeSort separately) -Finally consider [low, high] overall results -Tip: PreSum is made (n + 1) length, then the range sum [i, j] can be reduced to preSum [j]-preSum [i]

- NOTE: should write merge() function, but that is minor, just use `Arrays.sort(nums, start, end)`, OJ passed
- Every mergeSort() has a for loop => O(n log n)

How to count range?

-A special method here: find a preSum after i and mid in [low, mid] for comparison (explained from: https://blog.csdn.net/qq508618087/article/details/51435944) -Find two boundaries in the right array, set to `m`, `n`, -Where m is the first in the array on the right such that `sum [m]-sum [i]> = lower`, -n is the first position that makes `sum [n]-sum [i]> upper`, -In this way, `nm` is the number of intervals in the `[lower, upper]` range formed by the element i on the left.

The magic point: why merge and sort

-The borders [lower, higher] are compared in the sorted array. Once the national borders, the calculation can be stopped and unnecessary calculations can be reduced. -The premise of n, m above is feasible: preSum [] has two ranges [low, mid], [mid, high] before and after sorted -In other words, when recursively mergeSort (), you really need to merge sorted 2 partitions -You may ask: Can you sort? Sort will disrupt the order soon? Yes, the order of preSum [] is disrupted. -But it doesn't matter: very clever, when dividing and conquering, the first half / the second half are separated and the original process is separated, and the merge is finally completed. -When doing the range of m, n, the principle is as follows, for example, preSum is divided into two sections: [A, B, C], [D, E, F] -When comparing each preSum value A with preSum [i], A-preSum <lower, it is a single comparison, not involving B, C -Therefore, it does not matter whether [A, B, C] retains the order of the initial preSum - The most important thing at this time is that [A, B, C] and sorting are good, then when the lower boundary is compared, once the boundary is crossed, the calculation can be stopped (reduce unnecessary calculations)

#** BST

- EVERYTHING?

---

55. Max Sum of Rectangle No Larger Than K.java** Level: Hard Tags: [Array, BST, Binary Search, DP, Queue, TreeSet]

Given a non-empty two-dimensional matrix matrix and an integer k, find the largest rectangular sum of sums not greater than k within the matrix.

#** BST, Array, preSum -Reduce the problem to: row of values, find 1st value> = target.

- 1. loop over startingRow; 2. loop over [startingRow, m - 1]; 3. Use TreeSet to track areas and find boundary defined by k.
- When building more rows/cols the rectangle, total sum could be over k:
- when it happens, just need to find a new starting row or col,
- where the rectangle area can reduce/remain <= k
- Find the starting point of the extra area: extraArea = treeSet.ceiling (totalSum-k). That is, find the starting / left area after subtracting k.
- Remove these left starting areas, the rest is <= k. (Num-extraArea)
- Why use TreeSet: the size of the area is irregular, and find the first value of> = any value. Give a string of non-sorted numbers, find the number of> = target, if you do not write binary search, then BST is the most suitable
- O (m ^ 2 * nlogn)

#** Thought -Considering the most basic O (m ^ 2 * n ^ 2): traversing startingRow / startingCol - rectangle? layer by layer? You can think of the idea of Presum. When it is larger than the required sum, subtract the extra part. -How to find extra areas? Then search: Save the content you need to search, you can think of using BST (TreeSet), or write Binary Search yourself.

---

56. Perfect Rectangle.java** Level: Hard Tags: [Design, Geometry, Hash Table]

See if the list of coordinates can form a perfect rectangle and does not allow overlap area.

#** Drawing features -Feature 1: All the given points (find out the diagonal points without specifying), if the perfect rectangle is formed at the end, they should be eliminated from each other, and finally there are 4 corners left -Feature 2: Find the min / max (x, y) in all points, and the final maxArea should be equal to the area accumulate in the process -Feature 1 Make sure there is no hollow part in the middle, ensure that all coincident points will be eliminated from each other, and finally there are 4 vertices left -Feature 2 ensures no coincidence: coincident areas will eventually exceed maxArea

---

57. Max Points on a Line.java** Level: Hard Tags: [Array, Geometry, Hash Table, Math]

给list of (x,y) coordinates. Determine # of points on the same line

#** Observation

- If given n points, we can calculate all possible slopes. O(n^2) times
- For the two dots that generates the same slope, these dots could be on parallel** slopes
- figure out how to prune the parallel dots

#** Trick: prune parallel dots using greatest common divider

- GCD: greatest common divider
- Devide the x and y by their greatest common divider, such that x and y can be reduced to minimum value
- All other x and y can be reduced to such condition as well
- track the final reduced (x,y) in a map: they are the key to the count
- No need to use Map<Integer, Map<Integer, Integer>> to perform 2 level mapping; just map<String, Integer>, where the key is "x@y"

---

58. Number of Digit One.java** Level: Hard Tags: [Math]

Pure math problem, not quite representative

Explanation https://leetcode.com/problems/number-of-digit-one/discuss/64381/4+-lines-O(log-n)-C++JavaPython

---

59. Binary Representation.java** Level: Hard Tags: [Bit Manipulation, String]

#** String -First we need to solve in two halves, the breakpoint is '.': Str.split ("\."); -The half of Integer is easy to handle, in the while loop: num% 2, num / 2. Make a `parseInteger ()` function -Decimal is more complicated. Make a `parseDecimal ()` function: -bit == 1 Mathematical condition: Now num * 2> = 1. Update: num = num * 2-1; -Math condition for bit == 0: num * 2 <1. Update: num = num * 2

#** Note -num is double, decimals may loop infinitely under the formula `num = num * 2-1` -Therefore check: num repeatability, and binary code <32 bit. -So the title also has a 32BIT requirement!

---

60. Recover Binary Search Tree.java** Level: Hard Tags: [BST, DFS, Tree]

There are 2 node misplaces in BST, which are classified as: Requirement: O (1) extra space

#** Observation

- BST inorder traversal should give small -> large sequence
- misplaced means: a large->small item would occur, and later a large>small** would occur.
- The first large && second small item are the 2 candidates. Example
- [1, 5, 7, 10, 12, 15, 18]
- [1, 5, 15, 10, 12, 7, 18]

#** dfs, O(1) extra space

- traverse, and take note of the candidate
- at the end, swap value of the 2 candidates

#** O(n) space

- inorder traversal the nodes and save in array, find the 2 items misplanced and swap them
- But O(n) space should not be allowed

---

61. Jump Game II.java** Level: Hard Tags: [Array, Coordinate DP, DP, Greedy]

Giving a string of numbers is the distance that can be jumped. Goal: the minimum number of jumps to the last index possible.

#** Greedy

- always aiming for the `farest can go`
- if the `farest can go` breaches the end, return steps
- otherwise, send `start=end+1`, `end=farest` and keep stepping from here
- though trying with 2 loops, worst case [1,1,1,...1,1] could have O(n^2)
- But on average should be jumpping through the array without looking back
- time: average O(n)

#** Previous Notes, Greedy -Maintain a range, the farthest we can go. -index / i is step by step, each time when i <= range, make a while loop, find the farthest reach in it maxRange -Then update range = maxRange -Where step is the same as index, step by step. -The final check condition is that the range you can walk the farthest is> = nums.length-1, indicating that the focus has been reached with the least steps. Good.

#** Even simpler Greedy -Graphic http://www.cnblogs.com/lichen782/p/leetcode_Jump_Game_II.html

- track the farest point
- whenver curr index reachest the farest point, that means we are making a nother move, so count++
- there seems to have one assumption: must have a solution. Otherwise, count will be wrong number.
- In fact, it is exactly the same thinking pattern as the first greedy.

#** DP -DP [i]: record at point i, the minimum number of jumps to point i

- dp[i] = Math.min(dp[i], dp[j] + 1);
- condition (j + nums[j] >= i)
- Note the use of dp [i] = Integer.MAX_VALUE as the starting value to find min
- time: O(n^2), slow, and timesout

---

62. Longest Valid Parentheses.java** Level: Hard Tags: [Coordinate DP, Stack, String]

Give a string with only `(`, `)` in it. Find the length of the longest valid parentheses.

#** 1D Coordinate DP

- use dp[i] track local max, maintain global max
- int[] dp. dp[i]: longest valid string that ends on i.
- The ending is ')', 2 cases: 1. s [i-1] is just '('; 2. s [i] 's' is the beginning of a '('
- Note, if the ending is '(' is unreasonable, ignore it.
- init: dp [0] = 0, single char cannot be formed.
- Calculation order: from left to right, find local max, maintain global max
- O(n) space, O(n) runtime

#** Stack -Store all open / close parentheses in the Stack. -If stack.top () happens to open and close, just stack.pop (). -The rest are unreasonable elements. -A bit like negatively looking for solution: `endIndex-The last failedIndex (stack.pop ())-1`, it should be the length of the last succeded string -

Update endIndex to stack.top () each time, and then continue to find the next failedIndex from the stack -Compare all lengths to find the longest length -O (n) stack space, O (n) runtime. It should be slower than dp, because O (n) is done twice

---

63. Rearrange String k Distance Apart.java** Level: Hard Tags: [Greedy, Hash Table, Heap]

Give a string, all lowercase letters, and ask for rearrangement: then each unique character must have k distance apart.

It's similar to Task Scheduler, except that you can use other methods to find the count in Task. This problem requires the ranking results.

#** PriorityQueue + HashTable -A classic usage of PriorityQueue sorting + distribution sorting.

- Count frequency and store in pq.
- Consume element of pq for k rounds, each time pick one element from queue.
- Exception: if k still has content but queue is consumed: cannot complete valid string, return "";
- space, O(n) extra space in sb, O(26) constant space with pq.
- time: O(n) to add all items

---

64. Valid Number.java** Level: Hard Tags: [Enumeration, Math, String]

team: O (n)

Analyze edge cases, and various situations, and then determine whether it is a valid number

#** Situation summary -Several different cases of `.`, `e`, `+/-`, `int` -The results are different when the order of encounter is different. -This question is more about analyzing the situation, and then enumerating the edge case, the algorithm has less significance.

---

65. Bricks Falling When Hit.java** Level: Hard Tags: [Union Find]

Give a matrix of 1 and 0, `1` stands for brick. The brick connected to ceiling will not drop. Give a string of coordinate hits [] [], record how many drops each time you take down 1 brick.

#** UnionFind -1. We know that most bricks may be connected to ceiling, so every forward check is traverse all and timeout -2. Can I use union to install the connectors together, and then drop the connected ones when I drop the brick? Difficult: Because I still have to check all the current status of

the brick. -Inspired by other people's answers, since it is counting counts, we can think backwards: -Mark all hit-bricks = 2 (just ignore them), observe the last step of the whole situation, first calculate the total number of all bricks still connected to ceiling, and count all the counts in unionFind in count [0]. -The remaining ones that are not connected to ceiling are isolated islands -Method: Add the hit-bricks one by one, and then make a union again to see how many are eventually connected to ceiling. The increased count is the number of dropped bricks in forward thinking!

Union Find variants

-I still use the number index to do the union find, but I increase each index by +1, shift it to the right, and [0] is reserved for special purposes: -Use union at 0 to count the total remaining count of ceiling-connected bricks, where x = 0. -If you are on other topics, the condition may not be x = 0, but you can also use this union index = 0 to make a root statistic -The key: add the last hit brick back, and then re-union around this hit-brick: the increase in count is not the number of drops when the hit-brick is missing

#** DFS (timeout) -Consider the surroundings of each hit, all traverse, all without the ceiling: -For example, a matrix of 200 x 200 all 1s, the traverse must reach the top every time; the calculation is repeated, so timeout -The algorithm is correct, but it is not efficient. -I want to reduce repeated calculations, but I can't calculate in advance: the grid is constantly changing. So can you group all connected ceilings, can O (1) quickly check?

---

66. Interval Sum II.java** Level: Hard Tags: [Binary Search, Lint, Segment Tree]

SegmentTree large collection. Methods: build, query, modify. Not difficult. Just remember to make no mistakes.

#** Segment Tree

- build: recursively build children based on index-mid and link to curr node
- query: recursively try to find node.start == targetStart && node.end == targetEnd. Compare with node.mid
- modify: recursively try to find node.start == targetStart && node.end == targetEnd; modify and recursively assign upper interval with updated interval property.

---

67. HashHeap.java** Level: Hard Tags: [HashHeap, Heap]

No. It is a HashHeap implementation found from Chapter 9.

---

68. Trapping Rain Water II.java** Level: Hard Tags: [BFS, Heap, MinHeap, PriorityQueue]

Give a 2Dmap, each position has height. Find Trapping water sum.

#** Min Heap -Use PriorityQueue to sort the selected height for the position, create class Cell (x, y, height).

Note a few theories

-1. Start thinking around the matrix and find that the water that the matrix can hold depends on the low-height block -2. It must be considered from the outside, because the water is wrapped inside, and at least one layer must be existing outside -The above two points prompted us to use min-heap: the PriorityQueue of natural order.

Steps

-1. During the process, you can draw a picture to make it clear: that is to walk in all four directions, subtract the height of the surrounding cells from the height of the curr cell. -2. If it is greater than zero, then the surrounding cells will have standing water: because the cell is already the lowest on the periphery, so the inside is lower, there must be standing water. -3. Every visited cell must be marked, avoid revisit -4. Create a new cell and add it to the queue according to the movements of (mX, mY) `in 4 directions: cell (mX, mY) After the stagnant water has been calculated, the outer wall hours,? It becomes a wall. -5. Because what is done is to shrink a new fence, height = Math.max (cell.h, neighbor.h); -Same idea as trapping water I. Just from the periphery, it can only be added to the horizontal plane with the same height as the peripheral cell. Going inside, it is likely that the cell height changes.
-Attach the maximum height of curr cell and move-to cell here.

Why use Heap (min-heap-priorityQueue)

-To find the shortest board of a bucket -The shortest one needs to be processed first (on top)

Why traverse from outside to inside

-Wooden barrel theory -Peel onions and discard

---

69. Find Median from Data Stream.java** Level: Hard Tags: [Design, Heap, MaxHeap, MinHeap]

#** Principle -Think of the input stream as an upward slope. The middle point of the slope is naturally median. -In the first half, as maxHeap, the focus is on the peak of PriorityQueue, which is actually median.
-The second half, as minHeap, normal PriorityQueue. The beginning is minimal.

#** Note -First set here, which queue stores one more element. Here select maxHeap: maxHeap.size () == minHeap.size () + 1 || minHeap.size () -You must first maintain an element in maxHeap, otherwise it will cause problems when comparing sizes.

---

70. Sliding Window Median.java** Level: Hard Tags: [Design, Heap, MaxHeap, MinHeap, Sliding Window]

The same problem of Data Stream Median: not only increasing sequence, but also removing item (maintain a window size)

#** MaxHeap, MinHeap -Median still uses min-heap and max-heap. Time (logN) -Add / subtract: prioirtyQueue, log (n) -findMedian: O (1) -Add a number, subtract a number. -It is optimistic when adding or subtracting, whether it is drawn from the maxheap in the front or the minHeap behind. -Finish the balance.

#** Note -When using maxHeap, minHeap, it is customary to choose one more number for maxHeap: -The maxHeap on the left always has x + 1 or x numbers -MinHeap behind should always have x numbers

---

71. Design Search Autocomplete System.java** Level: Hard Tags: [Design, Hash Table, MinHeap, PriorityQueue, Trie]

time: input: O(x), where x = possible words, constructor: O(mn) m = max length, n = # of words space: O(n^2), n = # of possible words, n = # of trie levels; mainlay saving the Map<S, freq>

Description is long, but in short: 做 search auto complete.

Best problem to review Trie (prefix search), Top K frequent elements (Hash Map), and MinHeap (PriorityQueue)

Easier to revisit https://leetcode.com/problems/design-search-autocomplete-system/description/

#** Thinking direction -For text search, there is no doubt to use Prefix tree, trie.

Find all possible word / leaf, two options:

-? After Trie is done, do prefix search, then DFS / BFS return all leaf items. [High runtime complexity] -Store all possible words in TrieNode. [High space usage] -Shouldn't it be in memory space? Big question, so we can choose store all? possible words

Given k words, find top k frequent items. MinHeap is definitely used, but there are two solutions:

-? Store MinHeap with TrieNode: Because it will continue to search for new articles, the same prefix (especially at higher level) will be searched multiple times.

- [complexity: need to update heaps across all visited TrieNodes once new sentence is completed]
- Compute MinHeap on the fly: Of course we can't come to DFS every time? Otherwise it will be slow, so we must store list of possible candidates in TrieNode.
- Here we use `Map <String, freq>` in `Top K Frequent Words`, so O (m) is actually very convenient to build min-heap.

Train the system

-Each time a `#` is marked, an entry is added to the search history. Then `insert it into trie`.-This one can be done at the end when `#` is met, very concise

#** Trie, PriorityQueue, HashMap

- Trie Prefix Search + maintain top k frequent items
-

---

72. Integer to English Words.java** Level: Hard Tags: [Enumeration, Math, String]

Give a number less than Integer.MAX_VALUE (2 ^ 31-1), convert to English. (No need to add 'and')

#** String -Basic implementation - `Classification Discussion`: thounsand, million, billion. `3 numbers per division`.-Enumerate tokens with array -Use% and / to find English translation for each segment -3-digit part, you can use a helper funtion to find the result, the processing method of each segment is the same

#** Note -StringBuffer is more efficient! `Sb.insert (0, xxx)` append before sb -Note that when adding "", if it is unnecessary, try "trim ()" -Note that numbers less than 20 have their own special writing and require additional handles -This question is to be careful and patient. There is almost no algorithm. It is to write efficiently and correctly. You need to be careful.

73. Alien Dictionary.java** Level: Hard Tags: [BFS, Backtracking, DFS, Graph, Topological Sort]

Give an array of strings: If the array is sorted according to a new alphabet dictionary, you need to find the alphabet.

It is possible to have multiple sorting methods, just give one.

#** Graph -Essence: two strings above and below, corresponding to the same index, if the letters are different, the letter on the first line is more advanced in the alphabet -Turn string array into graph of topological sort: map <char, list <char >> -Also List [26] edges (Course Schedule problem)

- Build edges: find char diff between two row, and store the order indication into graph
- Note: indegree is always reversed (established in the opposite way to node to neighbors)

#** BFS -topological sort itself is well written, but first understand the nature of alphabetical sorting in the title -In fact, the nature of the above sorting is very imaginary, but it will be a bit difficult to think of it as the code for constructing the graph. -Calculate indegree, then use BFS to find nodes with inDegree == 0 -The first node with inDegree == 0 is placed before the alphabet. -The following solution uses Graph: map <Character, List > instead of List [26], in fact, try a dictionary with more than 26 letters.

- 如果 inDegree.size() != result.length() , there is nodes that did not make it into result.
- ex: cycle nodes from input, where inDegree of a one node would never reduce to 0, and will not be added to result
- In this case, it will be treated as invalid input, and return ""

#** DFS -It is exactly the same as the process of setting up grpah by BFS -The difference in DFS is: use visited map to mark the places you pass -When you get to leaf, add to result: but only add because you have reached the end, the final order should be reversed (or, sb.insert (0, x) directly add in reverse order)

---

74. Word Ladder II.java** Level: Hard Tags: [Array, BFS, Backtracking, DFS, Hash Table, String]

Give a string of string, start word, end word. Find all shortest path list from startWord-> endWord.

Variation: mutate 1 letter at a time.

#** BFS + Reverse Search -Find the shortest path with BFS.

- 问题: how to effectively store the path, if the number of paths are really large?
- If we store Queue<List>: all possibilities will very large and not maintainable

- Make a reverse structure with BFS, and then reverse search

BFS Prep Step

-BFS finds all the places where the start string can go and put it in an overall structure: Note, the way to store Map <s, list of sources> -BFS changes by 1 step each time, so recording the distance is actually the shortest path candidate (stop here)

- 1. 反向mutation map: destination/end string -> all source candidates using queue: Mutation Map
- Mutation Map<s, List>: list possible source strings to mutate into target key string.
- 2. 反向distance map: destination/end string -> shortest distance to reach dest
- Distance Map<s, possible/shortest distance>: shortest distance from to mutate into target key string.
- BFS prep step didn't solve the problem, even did not use end string. We need to use the reverse mapping structure built by BFS for search

Search using DFS

-Scan from end string, find all candidate dates && only visit candidate that is 1 step away -dfs until start string is found.

Bi-directional BFS: Search using BFS

-The reversed structure is ready, now you can search: you can also use bfs.

- Queue<List<String>> to store candidates, searching from end-> start

---

75. Text Justification.java** Level: Hard Tags: [Enumeration, String]

Adjust the text according to the rules. It is in Word: there is a line too long, adjust the space in the middle of the word, and then ensure the total width of each line.

There are some detailed rules, see the original question

#** String

- Summing space = width + (size-1). maintain: 1. list of candidates, 2. width of actual words
- calculate space in between: remain/(size - 1)
- overall for loop; clean up list: 1. over size; 2. last item

- It's not difficult at all, but be careful: when dealing with list of string, pay attention to the clean sum size of list .
- Clean processing space : only (n-1) items are processed, then the last one is taken out of the for loop, special processing.

#** Notes

- Clarification, observation:
- can start with greedy approach to stack as many words as possible
- once exceed the length, pop the top, and justify the added words (untouched words tracked by index)
- left justify: given list/stack of words with size t, overall remaining space length m,
- deal with last line with special care: just fill one space, and fill the rest of the row with space
- Does not seem very complicated, but need additional care of calculating the amount of space needed.
- Overall runtime: O(n) to go over all space
- Overall space O(maxWidth) for maxWidth amount of strings

---

76. Read N Characters Given Read4 II - Call multiple times.java** Level: Hard Tags: [Enumeration, String]

Read N Character using Read4 (char [] buf) enhanced version: can read continuously read (buf, n)

#** String -Pay attention to the index handle of String, slowly write edge case -Understand the meaning of the title: read4 (char [] buf) like populate input object has slightly less functionality. -When you come across, carefully understand the function usage, don't panic. In fact, the way of thinking is very simple, just carefully handle the string and edge case.

---

77. Frog Jump.java** Level: Hard Tags: [DP, Hash Table]

The question of Frog jump needs a little understanding: each grid can jump k-1, k, k + 1 steps, and k depends on the number of steps jumped in the previous step. By default, 0-> 1 must be a step.

Note: int [] stones is the unit where stone is located (not the number of steps that can be skipped, don't understand it wrong).

#** DP -I originally wanted to do it according to the corrdiante dp, but found many problems, and needed to track different possible previous starting spots. -According to jiuzhang answer: By definition, use a map of <stone, Set <possible # steps to reach stone >> -Each time a stone is processed, it takes three steps according to its own set of : k-1, k, or k + 1 steps. -Take a step each time to see if stone + step exists; if it exists, add it to the hash set of next position: stone + step

Note init

- dp.put (stone, new HashSet <> ()) mark the existence of each stone - dp.get (0) .add (0) init condition, used for dp.put (1, 1)

Thought

-In the end, think mode, more like BFS mode: starting from (0,0), add all possible ways

- 然后again, try next stone with all possible future ways ... etc

---

78. Longest Substring with At Most Two Distinct Characters.java** Level: Hard Tags: [Hash Table, Sliding Window, String, Two Pointers]

As the title.

#** Two Pointer + HashMap -Originally wanted to use DP, but the idea of practical sliding window - Cutting of sliding window: use hashmap to store last occurrance of char index; -After map.remove (c), that section is completely cut off; then map.get (c) + 1 is the new left window border

---

79. Shortest Distance from All Buildings.java** Level: Hard Tags: [BFS]

It is very similar to Walls and Gates, except that this question is to choose a coordinate, having shortest sum distance to all buildings (marked as 1).

#** BFS

- BFS 可以 mark shortest distance from bulding -> any possible spot.
- Try each building (marked as 1) -> BFS cover all 0.
- time: O(n^2) * # of building; use new visited[][] to mark visited for each building.
- O(n^2) find smallest point/aggregation value.
- Note, this question we update grid [] [] sum up with shortest path value from building.
- Find a min value at the end, even without return coordinates.
- Analyzed, not written yet.

---

80. Sliding Window Maximum.java** Level: Hard Tags: [Deque, Heap, Sliding Window]

#** Deque, Monotonous tail

- 维持monotonuous queue: one end is always at max and the other end is min. Always need to return the max end of queue.
- when adding new elements x: start from small-end of the queue, drop all smaller elements and append to first element larger than x.
- when sliding window: queue curr window max-end, remove it if needed.
- Wonderful: Use a deque data structure (actually in the form of LinkedList) to make a decreasing queue.
- Every time the smaller than the current node is removed, the rest is naturally: the largest> the second largest> the third largest ... ETC.
- We only care about the existence of the maximum value; any value that is less than the current value (which is to be added when new) cannot reach the maximum value anyway, so throw it away!

---

81. Median of Two Sorted Arrays.java** Level: Hard Tags: [Array, Binary Search, DFS, Divide and Conquer]

Famously find the median of two sorted arrays. Definition of median: if the total length of two arrays is even, take the average. The problem requires to be solved in log (m + n) time

-When you see log (m + n), I think of binary search, or recursive. -The two sorted arrays are jagged, and certainly not a simple binary search

#** Divide and Conquer, recursive -Here is a mathematical exclusion idea: consider the intermediate points of A and B respectively. -If A [mid] <B [mid], then A [0 ~ mid-1] is not in the range of median, which can be excluded. This is how divide / conquer comes. -See the code for the specific logic, which roughly means: compare the positions of A and B [x + k / 2-1] each time, and then do the range exclusion method

- end cases:
-
    1. If we find that the start index of A or B in dfs () overflows, then this is the simplest case: midian must be in another array
-
    2. If k == 1: find 1st item in A / B, then make Math.max (A [startA], B [startB])
- The total number length is (m + n) and every time the general content is deleted, then time is O (log (m + n))

---

82. Bus Routes.java** Level: Hard Tags: [BFS]

---

83. Sliding Puzzle.java** Level: Hard Tags: [BFS, Graph]

---

84. Cracking the Safe.java** Level: Hard Tags: [DFS, Greedy, Math]

#** Greedy, Iterative

- For 2 passwords, the shortest situation is both passwords overlap for n-1 chars.
- We can use a window to cut out last (n-1) substring and append with new candidate char from [k-1 ~ 0]
- Track the newly formed string; if new, add the new char to overall result
- Note: this operation will run for k^n times: for all spots of [0 ~ n - 1] each spot tries all k values [k-1 ~ 0]
- Same concept as dfs

#** DFS

- Same concept: use window to cut out tail, and append with new candidate
- do this for k^n = Math.pow(k, n) times

---

85. Redundant Connection II.java** Level: Hard Tags: [DFS, Graph, Tree, Union Find]

#** Union Find -Discuss 3 situations

- http://www.cnblogs.com/grandyang/p/8445733.html

---

86. The Maze III.java** Level: Hard Tags: [BFS, DFS, PriorityQueue]

#** BFS -Similar to Maze I, II, use a Node [] [] to store each (x, y) state.

- Different from traditional BFS(shortest path): it terminates BFS when good solution exists (distance), but will finish all possible routes
-
    1. Termination condition : if we already have a good/better solution on nodeMap[x][y], no need to add a new one
-
    2. Always cache the node if passed the test in step1
-
    3. Always offer the moved position as a new node to queue (as long as it fits condition)

- 4. Finally the item at nodeMap[target.x][target.y] will have the best solution.

---

87. Regular Expression Matching.java** Level: Hard Tags: [Backtracking, DP, Double Sequence DP, Sequence DP, String]

As with WildCard Matching, discuss the situation clearly with string p last char is *'' and not ''*

The difference here is that '*' needs to have a preceding element, then:

- repeat 0 times
- repeat 1 times: need s[i-1] match with prior char p[i-2]

---

88. Wildcard Matching.java** Level: Hard Tags: [Backtracking, DP, Double Sequence DP, Greedy, Sequence DP, String]

Double sequence DP. Much like regular expression.

Double Sequence DP -Analyze the true meaning of the characters?, * And write the expression. -Dp [i] [0] should always be false when initializing. When p is an empty string, it cannot be matched anyway (unless s = "" as well) -At the same time dp [0] [j] is not necessarily false. For example, s = "", p = ""* is a matching.

- A. p[j] != '*'
  1. last index match => dp[i - 1][j - 1]
  2. last index == ? => dp[i - 1][j - 1]
- B. p[j] == "*"
  1.
     - is empty => dp[i][j - 1]
  2.
     - match 1 or more chars => dp[i - 1][j]

---

89. Robot Room Cleaner.java** Level: Hard Tags: [Backtracking, DFS]

#** DFS

- Different from regular dfs to visit all, the robot move() function need to be called, backtrack needs to move() manually and backtracking path shold not be blocked by visited positions
- IMPORTANT: Mark on the way in using set, but backtrack directly without re-check against set
- Mark coordinate 'x@y'
- Backtrack: turn 2 times to revert, move 1 step, and turn 2 times to revert back.

- Direction has to be up, right, down, left.
- you [] dx = {-1, 0, 1, 0};, you [] dy = {0, 1, 0, -1};

---

90. Maximum Vacation Days.java** Level: Hard Tags: [DP]

Review (5)

0. Maximum Subarray III.java** Level: Review Tags: []

---

1. Valid Perfect Square.java** Level: Review Tags: [Binary Search, Math]

Binary looks for sqrt. Basic mid + 1, mid-1 template. 注意: define index as long.

---

2. Maximum Average Subarray II.java** Level: Review Tags: [Array, Binary Search, PreSum]

Give int [] nums and window min size k. The window size can be greater than K. Find the largest continuous series of average value.

-The idea of Binary Search, used on the average sum you are looking for. The size is in [min, max] -When looking for k, it can be> = k, use the concept of min (preSum). -When looking for k, draw a picture. It can be seen that what is actually required is the sum [x, i] in k window, so sum [0, i]-sum [0, x]

Need to read the notes below carefully.

---

3. The Skyline Problem.java** Level: Review Tags: [Binary Indexed Tree, Divide and Conquer, Heap, PriorityQueue, Segment Tree, Sweep Line]

Also called skyline. O (nLogN) with Sweep Line, but it seems that there are many ways: segement tree, hashheap, treeSet?

#** Sweep Line, Time O(nLogN), Space O(n)

- original reference http://codechen.blogspot.com/2015/06/leetcode-skyline-problem.html?_sm_au_=isVmHvFmFs40TWRt
- 画图分析: 需要找到 non-overlaping height point at current index; also height needs to be different than prev height peek to be visible.
- Divide all points, each point has index x, plus a height.
- Sort on this list, according to index and height. Note that building start point height is marked with a negative number, so that start is guaranteed to end
- Mark the start with a negative height: When comparing startPoint.height and endPoint.height with the same x-pos in the priority queue, because the end height is an integer, the start point is automatically placed before the end point when compare
- Of course, if the two start points are compared, if the negative value of the second point is too large (that is, the height is very high), it will naturally return a positive number by comparison, which will form an inverted position.
- Use max-heap (reversed priorityqueue) in processes, and then iterate heightPoints to save the maximum height. In the case of peek, it is a reasonable solution.
- Add 0 to heightQueue to close it at the end

#** Segment Tree -After reading some practices, the segment tree is complicated to write. It is estimated that it is difficult to write segment tree in the interview: https://www.cnblogs.com/tiezhibieek/p/5021202.html

#** HashHeap -HashHeap template can be considered: https://www.jiuzhang.com/solution/building-outline/#tag-highlight-lang-java

Binary Indexed Tree?

---

4. Remove Invalid Parentheses.java** Level: Review Tags: [BFS, DFS, DP]

Give a string with parentheses and other characters. Cut out the valid string with the least amount of knife, find all such strings.

There are multiple solutions to this problem, the strongest is O (n) space and time

#** DFS and reduce input string

- in dfs: remove the incorrect parentheses one at a time
- detect the incorrect parentheses by tracking/counting (similar to validation of the parentheses string): if(count<0)
- once detected, remove the char from middle of s, and dfs on the rest of the s that has not been tested yet.

Core concept: reverse test

- if a parenthese string is valid, the reverse of it should also be valid
- Test s with open='(', close=')' first; reverse s, and test it with open=')', close='('

Minor details

- only procceed to remove invalid parenthese when `count<0`, and also break && return dfs after the recursive calls.
- The above 2 facts eliminates all the redundant results.
- Reverse string before alternating open and close parentheses, so when returning final result, it will return the correct order.
- Open questions: how does it guarantee minimum removals?

Backtracking

-If you use a stringbuffer, you will not create a new string every time, but you need to maintain the string buffer, and you will backtracking

Complexity

- Seems to be O(n), but need to derive

#** BFS EVERYTHING

#** DP
