Revisions:

2015-01-15

* **~Class 28 Solutions Available.**

### 

### Index

[Index](#h.jb8elqvr1fjc)

[Class 1 - Array and Sorting Algorithms](#h.d6vr01bwy2zy)

[1. Selection Sort](#h.utzx1tfrud6)

[2. Merge Sort](#h.yw5bu9aajbry)

[3. Quick Sort](#h.hlj84p2tts46)

[4. Rainbow Sort](#h.wzc5c6jhj06e)

[Class 2 - Recursion and Binary Search](#h.ypv7hdbgx8kf)

[1. Classical Binary Search](#h.s0gv746vjwln)

[2. Binary Search Variants](#h.b1xxenmu1mpi)

[3. Unknown Size Binary Search](#h.62cnjehmura8)

[4. Shifted Sorted Array Binary Search](#h.gzafv28wgndj)

[5. Occurrence Of Target Number In Sorted Array](#h.b705xws8dnkt)

[Class 3 - Stack and Linked List](#h.e90cmwwfawd)

[1. ListNode class and utilities](#h.zfpzlljo6kj)

[2. Reverse Linked List (Recursive/Iterative)](#h.nrrgrsl73m5o)

[3. Middle Node Of Linked List](#h.dwrvwyflo6di)

[4. Determine If Linked List Has Cycle](#h.i7ow1k91rv3y)

[5. Merge Two Sorted Linked Lists](#h.rw3l01kll2sm)

[6. ReOrder Linked List](#h.1f4x8x70slrh)

[7. Partition Linked List](#h.o7j5fr5f2k89)

[Class 4 - Binary Tree & Binary Search Tree](#h.ihdohjvdh1f)

[1. TreeNode class and utilities](#h.aby0a4d5nfwa)

[2. Binary Tree Pre-order, In-order, Post-order Iterative Traversals](#h.83nobg48s7ie)

[3. Judge Whether A Binary Tree Is Binary Search Tree](#h.gafieqrj59aa)

[4. Judge Whether Two Binary Trees Are Identical](#h.at6li8v2hjws)

[5. Judge Whether A Binary Tree Is Symmetric](#h.f7w6vvs1fc0x)

[6. Judge Whether Two Binary Trees Are Tweaked Identical](#h.jp8wiiap3rw0)

[Class 5 - Heap & Graph + Search Algorithms](#h.f9sca3f5rvd8)

[1. GraphNode class and sample graphs](#h.v99fgzkrt7e8)

[2. Determine Whether An Undirected Graph Is Bipartite](#h.uavpx8d0f4za)

[3. Smallest K Elements In An Unsorted Array](#h.2gp0pffd11pm)

[4. K-th smallest Element In Row Sorted-Column Sorted Matrix](#h.2wq30e2zjrgl)

[5. Print Values Of Nodes Layer By Layer In Binary Tree](#h.l0usog7z7xlp)

[Class 6 - DFS & Hashtable](#h.jg3ob5o67700)

[1. All SubSets Of A Set Without Duplicates](#h.5u70bw3n3fqq)

[2. All Valid Permutations Of Parentheses](#h.e4yv7cs9nj60)

[3. Various Combinations Of Coins](#h.1ukvkktkxgv9)

[4. Top K Frequent Words](#h.c7tibme1olbb)

[5. Missing Number](#h.23145li9rjzu)

[6. Common Numbers Between Two Sorted Arrays](#h.nm4c70s77cvr)

[Class 7 - Midterm 1](#h.3567v3k19pkk)

[1. All Permutations Of A String With/WithOut Duplicates](#h.m43qqyiq8pjr)

[Class 8 - Bit representation of a number and bit operation](#h.spuw2wsvc1k8)

[1. Determine If An Integer Is Power Of 2](#h.8ixqv7htbk4e)

[2. Determine The Number Of Different Bits Between 2 Integers](#h.cbxotm6z396z)

[3. Determine Whether A ASCII Word Contains All Unique Characters](#h.7xh2h06n1ue9)

[4. Get The Hexadecimal Representation String Of A Non-negative Integer](#h.q063mhg0phxj)

[Class 11 - Dynamic Programming I](#h.ozsg66cgkame)

[1. Fibonacci Number](#h.ooxl186hxuxy)

[2. Longest Ascending Subarray](#h.pzd9d4297oym)

[3. Max Product Of Cutting Rope](#h.tk5o0nq7eddo)

[4. Jump Game](#h.6ux5stmhwrhh)

[Class 12 - String](#h.ps9oe1c4stt2)

[1. Remove Adjacent And Duplicated Characters(Leave Only One)](#h.fvjn8e1qhnah)

[2. Replace Space Character With “20%”](#h.dvoc661inch9)

[3. Reverse By Words](#h.3x6uo4pe98bk)

[4. Determine Whether A String Is Substring Of Another](#h.dda5bv5n1mam)

[5. Repeatedly Deduplicate Adjacent Repeated Characters](#h.ocdekhp0uhx8)

[Class 14 - Dynamic Programming II](#h.4pw7nyvgd3eo)

[1. Minimum Number Of Jumps](#h.ljww7mc8r5wq)

[2. Dictionary Word](#h.73jycr1dj4jh)

[3. Edit Distance](#h.oustpurfgt0a)

[4. Largest Square Of 1’s In Binary Matrix](#h.cjmwfvuz1v4v)

[5. Merge Stones](#h.45d8nhdbej1u)

[Class 16 - Dynamic Programming III](#h.o61j2bu7c0yf)

[1. Largest Sum Of A Subarray](#h.7rk41l678f2s)

[2. Longest Consecutive 1’s](#h.2etggg9j11a9)

[3. Largest Cross Of 1’s](#h.cdh58bbg5chd)

[4. Biggest Rectangle In Histogram](#h.xs8z0ry6dc83)

[5. Largest Rectangle Of 1’s](#h.y2n804revv3i)

[6. Largest Sum Of Submatrix](#h.5lpio8h0hkmh)

[Class 18 - Probability, Sampling And Randomization](#h.djo6cz6qwp7s)

[1. Shuffle](#h.au0n6u7j9a50)

[2. Reservoir Sampling](#h.yqeii9vf4nch)

[3. Random7 Using Random5](#h.fvtgbyqx7pll)

[4. Median Tracker](#h.jobcvnk1rsdv)

[5. 95 Percentile Of Urls’ Lengths](#h.gvt5gmcgjqck)

[Class 19 - 强化练习 1](#h.ynfmru34fay0)

[1. Remove Duplicates From Sorted Array](#h.jqvk0s7phmus)

[2. Least Comparisons](#h.669r11rlnnfa)

[3. Generate 2D Array In Spiral Order](#h.jhrjylbddusv)

[4. Rotate 2D Array Clockwise By 90 Degrees](#h.hdm835jl20tn)

[5. Binary Tree Level Order Traversal In Zig-Zag Way](#h.kg1bwn75yk84)

[6. Lowest Common Ancestor](#h.pfziooidb70c)

[Class 20 - Midterm 2](#h.f32vlpw3riex)

[1. Max Path Sum From One Leaf To Another](#h.lmd7oie7npbe)

[2. Move All 0’s To The Right End](#h.exti1hedekgf)

[3. Min Cuts Of Palindrome Partition](#h.cx2zcnpp1oey)

[Class 22 - 强化练习 2](#h.qfqug7ruk5ns)

[1. Deep Copy Skip List](#h.l9b7i3r585k2)

[2. Deep Copy Graph With Possible Cycle](#h.8sb1ckswyjxn)

[3. Merge K Sorted Arrays](#h.ufbo526knsm7)

[4. Merge K Sorted Lists](#h.1pint1dst0k)

[5. Binary Search Tree Closest Node To Target](#h.yqcx2s2gmhti)

[6. Binary Search Tree Largest Node Smaller Than Target](#h.5q5ifw5q30dj)

[7. Binary Search Tree Delete Target](#h.qf8mamn8zt2c)

[8. Binary Search Tree Insert Target](#h.gfwlrmrmxxu9)

[9. Cutting Wood](#h.5e9f36i5zu8g)

[Class 23 - 强化练习 3](#h.5gefxqvs1mi7)

[1. Common Numbers Of Two Sorted Arrays](#h.5cc1jzfpbh7s)

[2. Common Elements In Three Sorted Array](#h.vsevctu87hsb)

[3. String Replace](#h.gowcizmrphfb)

[4. Decompress String II](#h.vuz9hh88339k)

[5. Check If Binary Tree Is Balanced](#h.lq9iztvontq8)

[6. Longest Common Substring](#h.8jdkqsra6tkw)

[7. Longest Common Subsequence](#h.jg0v3sd0ejpq)

[Class 24 - 强化练习 4](#h.3ysnufktut1)

[1. Reverse Singly Linked List](#h.22885ostx2x8)

[2. Reverse Linked List Pair By Pair](#h.yr8z28y76g7)

[3. Reverse Binary Tree Upside Down](#h.7ay367e89m7d)

[4. All Valid Permutations Of l (), m [], n {}](#h.5wd2df7yybop)

[5. Eight Queens](#h.54ua13aryfj6)

[6. 2 Sum](#h.79rhk5jbjqu8)

[7. 3 Sum](#h.5df0sfu7byhv)

[8. 4 Sum](#h.tzt3gwo3b1ub)

[Class 25 - 强化练习(Recursion 总结)](#h.gv1kzh87nda)

[1. Reverse String Using Recursion](#h.dyskd34zz287)

[2. Abbreviation Matching](#h.gufoik89uks8)

[3. Store Number Of Nodes In Left Subtree For Each Node](#h.3lbiqrgokq12)

[4. Node With Max Difference Number Of Nodes In Left/Right Subtree](#h.ckh5btdfl5xv)

[5. Binary Tree Maximum Path Sum II (Any Node To Any Node)](#h.jz57h0tynt4o)

[6. Binary Tree Path Sum To Target](#h.sdycrebohhs)

[7. Binary Tree Maximum Path Sum III (Any Node To Any Node On The Same Path From Root To Leaf)](#h.pofsdura9s3y)

[8. Reconstruct Complete Binary Tree From Level Order](#h.z6h6hz5khouj)

[9. Reconstruct Binary Search Tree From Pre Order](#h.o6jy2t1a18pg)

[Class 26 - 强化练习 5](#h.1ajj34txpg6p)

[1. Max Length Product Of Two Strings Without Common Characters](#h.7730rx98nz3n)

[2. Kth Smallest Number With Only 3,5,7 As Factors](#h.q3mit3jruzl5)

[3. Kth Closest Point To <0, 0, 0>](#h.rp9x5873geqz)

[4. Place To Put Chair I](#h.hjjcj7wgrtig)

[5. String Conversion](#h.fpl5k91wcksd)

[6. Largest Rectangle In Histogram](#h.56n6na8x3b60)

[7. Max Water Trapped In Histogram](#h.5tpzjeslibhb)

[8. Sky Line](#h.f94z9na8ay5z)

[Class 27 - 强化练习 6](#h.we43a8kn0yoz)

[1. Kth Smallest In Two Sorted Arrays](#h.lxs45tqukucb)

[2. Hexadecimal Representation](#h.l44ulrmmdqz1)

[3. Max Values Of Sliding Windows](#h.1gxebzq5219t)

[4. LRU Cache](#h.ivbxdgz5bpts)

[5. First Non-Repeating Character In Stream](#h.aerhmmtcix2o)

[6. Majority Number](#h.oglcxmnwr5w7)

[7. Interleave Array](#h.h9nneo91ikv5)

[Class 28 - 强化练习 7](#h.vjxhtg22om3d)

[1. Reconstruct Binary Tree With Preorder And Inorder Traversal Sequence](#h.a07rej1xassr)

[2. Reconstruct Binary Tree With Postorder And Inorder Traversal Sequence](#h.9pt4jjfwms3x)

[3. Reconstruct Binary Tree With Levelorder And Inorder Traversal Sequence](#h.694n31irvwwr)

[4. Reconstruct Binary Search Tree With Preorder Traversal Sequence](#h.l8yx07p2qkmt)

[5. Reconstruct Binary Search Tree With Postorder Traversal Sequence](#h.6jcxmqx8oqrm)

[6. Reconstruct Binary Search Tree With Levelorder Traversal Sequence](#h.bl0tzpukkf2r)

[7. Most Points On A Line](#h.6bfp20puzh1e)

[8. Largest Set Of Points With Positive Slopes](#h.s4al5wcib5t4)

[9. Generate Random Maze](#h.84gjmbiqhx4l)

[10. Disjoint White Objects](#h.qcv78tus0pqw)

### 

### 

### 

### Class 1 - Array and Sorting Algorithms

#### 1. Selection Sort

/\*\*

\* Array selection sort implementation.

\*/

public class SelectionSort {

public void selectionSort(int[] array) {

// sanity check before the main logic is applied.

// conditions to consider: null? empty? ......

if (array == null) {

return;

}

for (int index = 0; index < array.length - 1; index++) {

int minIndex = index;

// find the min element in the subarray of (index, array.length - 1)

for (int tmpIndex = index + 1; tmpIndex < array.length; tmpIndex++) {

if (array[tmpIndex] < array[minIndex]) {

minIndex = tmpIndex;

}

}

swap(array, index, minIndex);

}

}

public void swap(int[] array, int left, int right) {

int temp = array[left];

array[left] = array[right];

array[right] = temp;

}

public static void main(String[] args) {

SelectionSort solution = new SelectionSort();

// test cases to cover all the possible situations.

int[] array = new int[0];

solution.selectionSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 1, 2, 3, 4 };

solution.selectionSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 4, 3, 2, 1 };

solution.selectionSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 2, 4, 1, 5, 3 };

solution.selectionSort(array);

System.out.println(Arrays.toString(array));

}

}

#### 2. Merge Sort

/\*\*

\* Array merge sort implementation.

\*/

public class MergeSort {

public void mergeSort(int[] array) {

// sanity check at first

if (array == null) {

return;

}

// allocate helper array to help merge step

int[] helper = new int[array.length];

mergeSort(array, helper, 0, array.length - 1);

}

// make dest array's subarray (left, right) identical as source array's

public void copyArray(int[] source, int[] dest, int left, int right) {

for (int index = left; index <= right; index++) {

dest[index] = source[index];

}

}

private void mergeSort(int[] array, int[] helper, int left, int right) {

if (left >= right) {

return;

}

int mid = left + (right - left) / 2;

mergeSort(array, helper, left, mid);

mergeSort(array, helper, mid + 1, right);

merge(array, helper, left, mid, right);

}

private void merge(int[] array, int[] helper, int left, int mid, int right) {

// first copy the content to helper array

copyArray(array, helper, left, right);

int leftIndex = left;

int rightIndex = mid + 1;

while (leftIndex <= mid && rightIndex <= right) {

if (helper[leftIndex] <= helper[rightIndex]) {

array[left++] = helper[leftIndex++];

} else {

array[left++] = helper[rightIndex++];

}

}

// if we have some elements left at left side, we need to copy them

while (leftIndex <= mid) {

array[left++] = helper[leftIndex++];

}

}

public static void main(String[] args) {

MergeSort solution = new MergeSort();

int[] array = new int[0];

solution.mergeSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 1, 2, 3, 4 };

solution.mergeSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 4, 3, 2, 1 };

solution.mergeSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 2, 4, 1, 5, 3 };

solution.mergeSort(array);

System.out.println(Arrays.toString(array));

}

}

#### 3. Quick Sort

/\*\*

\* Array quick sort implementation.

\*/

public class QuickSort {

public void quickSort(int[] array) {

if (array == null) {

return;

}

quickSort(array, 0, array.length - 1);

}

public void quickSort(int[] array, int left, int right) {

if (left >= right) {

return;

}

int pivotPos = partition(array, left, right);

quickSort(array, left, pivotPos - 1);

quickSort(array, pivotPos + 1, right);

}

public void swap(int[] array, int left, int right) {

int temp = array[left];

array[left] = array[right];

array[right] = temp;

}

private int partition(int[] array, int left, int right) {

int pivotPos = pivotPos(left, right);

int pivot = array[pivotPos];

// swap the pivot element to the rightmost position first

swap(array, pivotPos, right);

int leftBound = left;

int rightBound = right - 1;

while (leftBound <= rightBound) {

if (array[leftBound] < pivot) {

leftBound++;

} else {

swap(array, leftBound, rightBound--);

}

}

// swap back the pivot element.

swap(array, leftBound, right);

return leftBound;

}

private int pivotPos(int left, int right) {

// sample implementation, pick random element as pivot each time.

return left + (int) (Math.random() \* (right - left + 1));

}

public static void main(String[] args) {

QuickSort solution = new QuickSort();

int[] array = new int[0];

solution.quickSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 1, 2, 3, 4 };

solution.quickSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 4, 3, 2, 1 };

solution.quickSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 2, 5, 3, 1, 4 };

solution.quickSort(array);

System.out.println(Arrays.toString(array));

}

}

#### 4. Rainbow Sort

/\*\*

\* Rainbow sort implementation.

\* Assumption:

\* 1).we have three colors denoted as 0, 1, 2 and all the elements in the array

\* are valid.

\*/

public class RainbowSort {

public void rainbowSort(int[] array) {

if (array == null) {

return;

}

// three bounds:

// 1. the left side of boundZero is 0.

// 2. the right side of boundTwo is 2.

// 3. the part between boundZero and boundOne is 1.

// 4. the part between boundOne and boundTwo is to be discovered.

int boundZero = 0;

int boundOne = 0;

int boundTwo = array.length - 1;

while (boundOne <= boundTwo) {

if (array[boundOne] == 1) {

boundOne++;

} else if (array[boundOne] == 0) {

swap(array, boundZero++, boundOne++);

} else {

swap(array, boundOne, boundTwo--);

}

}

}

private void swap(int[] array, int left, int right) {

int temp = array[left];

array[left] = array[right];

array[right] = temp;

}

public static void main(String[] args) {

RainbowSort solution = new RainbowSort();

int[] array = new int[0];

solution.rainbowSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 0, 0, 0, 0 };

solution.rainbowSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 0, 0, 1, 1 };

solution.rainbowSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 2, 0, 0, 0 };

solution.rainbowSort(array);

System.out.println(Arrays.toString(array));

array = new int[] { 2, 0, 1, 2, 0 };

solution.rainbowSort(array);

System.out.println(Arrays.toString(array));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 2 - Recursion and Binary Search

#### 1. Classical Binary Search

/\*\*

\* Classical binary search implementation.

\* Assumption:

\* 1). the array has been already sorted.

\*/

public class ClassicalBinarySearch {

public int binarySearch(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid;

} else if (array[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

public static void main(String[] args) {

ClassicalBinarySearch solution = new ClassicalBinarySearch();

int target = 5;

int[] array = new int[0];

System.out.println(solution.binarySearch(array, target));

array = new int[] { 1, 2, 3, 4 };

System.out.println(solution.binarySearch(array, target));

array = new int[] { 6, 9, 11 };

System.out.println(solution.binarySearch(array, target));

array = new int[] { 1, 2, 5, 8, 10 };

System.out.println(solution.binarySearch(array, target));

}

}

#### 2. Binary Search Variants

/\*\*

\* Binary search variants implementation.

\* Variant 1.1, 1.2 and 1.3

\* Assumption:

\* 1). the array has been already sorted.

\*/

public class BinarySearchVariants {

// 1.1

public int firstOccur(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left < right - 1) {

int mid = left + (right - left) / 2;

if (array[mid] >= target) {

right = mid;

} else {

left = mid;

}

}

if (array[left] == target) {

return left;

}

if (array[right] == target) {

return right;

}

return -1;

}

// 1.2

public int lastOccur(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left < right - 1) {

int mid = left + (right - left) / 2;

if (array[mid] <= target) {

left = mid;

} else {

right = mid;

}

}

if (array[right] == target) {

return right;

}

if (array[left] == target) {

return left;

}

return -1;

}

// 1.3 return is the index of the closest number.

public int closest(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left < right - 1) {

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid;

} else if (array[mid] < target) {

left = mid;

} else {

right = mid;

}

}

if (Math.abs(array[left] - target) <= Math.abs(array[right] - target)) {

return left;

} else {

return right;

}

}

// 1.3a closest k numbers to target, return k closest numbers' indices

// assumption: k <= array.length

public int[] kClosest(int[] array, int target, int K) {

assert array != null && array.length != 0 && K <= array.length;

int[] kClosest = new int[K];

if (K == 0) {

return kClosest;

}

int closest = closest(array, target);

kClosest[0] = closest;

int smaller = closest - 1;

int larger = closest + 1;

for (int i = 1; i < K; i++) {

if (larger >= array.length || smaller >= 0

&& (target - array[smaller]) <= (array[larger] - target)) {

kClosest[i] = smaller--;

} else {

kClosest[i] = larger++;

}

}

return kClosest;

}

public static void main(String[] args) {

BinarySearchVariants solution = new BinarySearchVariants();

int target = 5;

int K = 3;

int[] array = new int[0];

System.out.println("first: " + solution.firstOccur(array, target) + " last: "

+ solution.lastOccur(array, target) + " closest: " + solution.closest(array, target)

+ " k closest: " + Arrays.toString(solution.kClosest(array, target, K)));

array = new int[] { 1, 2, 3, 4 };

K = 2;

System.out.println("first: " + solution.firstOccur(array, target) + " last: "

+ solution.lastOccur(array, target) + " closest: " + solution.closest(array, target)

+ " k closest: " + Arrays.toString(solution.kClosest(array, target, K)));

array = new int[] { 6, 9, 11 };

K = 3;

System.out.println("first: " + solution.firstOccur(array, target) + " last: "

+ solution.lastOccur(array, target) + " closest: " + solution.closest(array, target)

+ " k closest: " + Arrays.toString(solution.kClosest(array, target, K)));

array = new int[] { 1, 2, 5, 5, 5, 8, 10 };

K = 4;

System.out.println("first: " + solution.firstOccur(array, target) + " last: "

+ solution.lastOccur(array, target) + " closest: " + solution.closest(array, target)

+ " k closest: " + Arrays.toString(solution.kClosest(array, target, K)));

}

}

#### 3. Unknown Size Binary Search

/\*\*

\* Binary search implementation on an dictionary with unknown size.

\* Assumption:

\* 1). The dictionary is an unknown sized array, it only provides get(int index)

\* functionality, if the index asked for is out of right bound, it will return

\* null.

\* 2). The elements in the dictionary are all Integers.

\*/

public class UnknownSizeBinarySearch {

/\*

\* Wrapper class for an unknown sized int array. The length() method is not

\* provided to outside the class.

\*/

public static class Dictionary {

private int[] array;

public Dictionary(int[] array) {

this.array = array;

}

public Integer get(int index) {

if (array == null || index >= array.length) {

return null;

}

return array[index];

}

}

public int unknownSizeBinarySearch(Dictionary dictionary, int target) {

if (dictionary == null) {

return -1;

}

int left = 0;

// initialize right index as 2 ^ 0 - 1

int right = 0;

while (dictionary.get(right) != null && dictionary.get(right) < target) {

// 1. move left to right,

// 2. double right index, since the array's indices are started from 0,

// we use right = 2 ^ n - 1

left = right;

right = 2 \* right + 1;

}

// not very necessary, but can return earlier.

if (dictionary.get(right) != null && dictionary.get(right) == target) {

return right;

}

return binarySearch(dictionary, target, left, right);

}

private int binarySearch(Dictionary dict, int target, int left, int right) {

// classical binary search

while (left <= right) {

int mid = left + (right - left) / 2;

if (dict.get(mid) == null || dict.get(mid) > target) {

right = mid - 1;

} else if (dict.get(mid) < target) {

left = mid + 1;

} else {

return mid;

}

}

return -1;

}

public static void main(String[] args) {

UnknownSizeBinarySearch solution = new UnknownSizeBinarySearch();

int target = 5;

Dictionary dictionary = new Dictionary(new int[0]);

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

dictionary = new Dictionary(new int[] { 1, 2, 3, 4, 5 });

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

dictionary = new Dictionary(new int[] { 1, 2 });

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

dictionary = new Dictionary(new int[] { 6 });

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

target = 9999;

dictionary = new Dictionary(generateSequence(10001));

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

dictionary = new Dictionary(generateSequence(9998));

System.out.println(solution.unknownSizeBinarySearch(dictionary, target));

}

/\*

\* Generate int arry of sequence from 0 to a specified number.

\*/

private static int[] generateSequence(int number) {

assert number >= 0;

int[] result = new int[number + 1];

for (int index = 0; index <= number; index++) {

result[index] = index;

}

return result;

}

}

#### 4. Shifted Sorted Array Binary Search

/\*\*

\* Binary search implementation in a shifted sorted array.

\* Assumption:

\* 1). there is no duplicated elements in the array.

\*/

public class ShiftedSortedArrayBinarySearch {

public int search(int[] array, int target) {

if (array == null) {

return -1;

}

int left = 0;

int right = array.length - 1;

// case 1. array[mid] is the target

// case 2. array's left part is monotonic increasing

// case 3. array's right part is monotonic increasing

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid;

} else if (array[left] <= array[mid]) {

if (target >= array[left] && target < array[mid]) {

right = mid - 1;

} else {

left = mid + 1;

}

} else {

if (target > array[mid] && target <= array[right]) {

left = mid + 1;

} else {

right = mid - 1;

}

}

}

return -1;

}

public static void main(String[] args) {

ShiftedSortedArrayBinarySearch solution = new ShiftedSortedArrayBinarySearch();

int target = 5;

int[] array = new int[0];

System.out.println(solution.search(array, target));

array = new int[] { 1, 2, 3, 4, 5 };

System.out.println(solution.search(array, target));

array = new int[] { 5, 6, 9, 10 };

System.out.println(solution.search(array, target));

array = new int[] { 4, 5, 1, 2, 3 };

System.out.println(solution.search(array, target));

array = new int[] { 4, 1, 2, 3 };

System.out.println(solution.search(array, target));

}

}

#### 5. Occurrence Of Target Number In Sorted Array

/\*\*

\* Count the number of occurrence of the target value in a sorted array.

\*/

public class TotalOccurrence {

public int totalOccurrence(int[] array, int target) {

if (array == null) {

return 0;

}

// the idea is to find the first occurrence and the last occurrence,

// then we know how many target values we have in the sorted array.

int left = firstOccur(array, target);

if (left == -1) {

return 0;

}

int right = lastOccur(array, target);

return right - left + 1;

}

public int firstOccur(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] >= target) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return left < array.length && array[left] == target ? left : -1;

}

public int lastOccur(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] <= target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return right >= 0 && array[right] == target ? right : -1;

}

public static void main(String[] args) {

TotalOccurrence solution = new TotalOccurrence();

int target = 5;

int[] array = new int[0];

System.out.println(solution.totalOccurrence(array, target));

array = new int[] { 1, 2, 3 };

System.out.println(solution.totalOccurrence(array, target));

array = new int[] { 1, 3, 5 };

System.out.println(solution.totalOccurrence(array, target));

array = new int[] { 1, 3, 5, 5, 5, 5, 7, 9 };

System.out.println(solution.totalOccurrence(array, target));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 3 - Stack and Linked List

#### 1. ListNode class and utilities

/\*\*

\* ListNode definition in a singled linked list.

\*/

public class ListNode {

public int value;

public ListNode next;

public ListNode(int value) {

this.value = value;

}

/\*

\* Utility method to convert an int array to singled linked list.

\*/

public static ListNode arrayToList(int[] array) {

if (array == null || array.length == 0) {

return null;

}

ListNode head = new ListNode(array[0]);

ListNode current = head;

for (int index = 1; index < array.length; index++) {

current.next = new ListNode(array[index]);

current = current.next;

}

return head;

}

/\*\*

\* Utility method to convert an int array to singled linked list

\* with circle.

\*

\* Example:

\* arrayToListWithCircle(new int[] {1, 2, 3, 4, 5}, 2) will construct

\* a list with circle starting at the node of 3.

\*

\* @param array

\* the int array

\* @param preLength

\* the number of nodes before entering the circle.

\* @return the head of the list

\*/

public static ListNode arrayToListWithCircle(int[] array, int preLength) {

if (array == null || array.length == 0) {

return null;

}

ListNode dummyHead = new ListNode(Integer.MIN\_VALUE);

ListNode current = dummyHead;

ListNode circleStart = null;

for (int index = 0; index < array.length; index++) {

current.next = new ListNode(array[index]);

current = current.next;

if (index == preLength) {

circleStart = current;

}

}

current.next = circleStart;

return dummyHead.next;

}

/\*

\* Utility method to print out a singled linked list.

\*/

public static void printList(ListNode head) {

StringBuilder builder = new StringBuilder();

while (head != null) {

builder.append(head.value).append(" -> ");

head = head.next;

}

builder.append("null");

System.out.println(builder.toString());

}

/\*

\* Utility method to print out a node in linked list.

\*/

public static void printListNode(ListNode node) {

if (node == null) {

System.out.println("null");

} else {

System.out.println("[ " + node.value + " ]");

}

}

}

#### 2. Reverse Linked List (Recursive/Iterative)

/\*\*

\* Reverse singled linked list, both iterative and recursive ways.

\*/

public class ReverseLinkedList {

public ListNode reverseIterative(ListNode head) {

if (head == null) {

return head;

}

ListNode current = head;

ListNode prev = null;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

public ListNode reverseRecursive(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode result = reverseRecursive(head.next);

head.next.next = head;

head.next = null;

return result;

}

public static void main(String[] args) {

ReverseLinkedList solution = new ReverseLinkedList();

ListNode head = null;

ListNode.printList(solution.reverseIterative(head));

head = null;

ListNode.printList(solution.reverseRecursive(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.reverseIterative(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.reverseRecursive(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.reverseIterative(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.reverseRecursive(head));

}

}

#### 3. Middle Node Of Linked List

/\*\*

\* Find the middle node of a singled linked list.

\* Assumption:

\* 1). 1 -> 2 -> 3 ->null, the middle node is 2.

\* 2). 1 -> 2 -> 3 -> 4 -> null, the middle node is 2.

\* 3). null, the middle node is null.

\*/

public class MiddleNode {

public ListNode findMiddle(ListNode head) {

if (head == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

public static void main(String[] args) {

MiddleNode solution = new MiddleNode();

ListNode head = null;

ListNode.printList(solution.findMiddle(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.findMiddle(head));

head = ListNode.arrayToList(new int[] { 1, 2 });

ListNode.printList(solution.findMiddle(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4 });

ListNode.printList(solution.findMiddle(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4, 5, 6, 7 });

ListNode.printList(solution.findMiddle(head));

}

}

#### 4. Determine If Linked List Has Cycle

/\*\*

\* Determine if a linked list has circle.

\* If it has circle, return the start node of the circle.

\* Else return null.

\*/

public class CircularList {

public ListNode hasCircle(ListNode head) {

if (head == null || head.next == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

if (fast == slow) {

break;

}

}

if (fast == null || fast.next == null) {

return null;

}

fast = head;

while (fast != slow) {

fast = fast.next;

slow = slow.next;

}

return fast;

}

public static void main(String[] args) {

CircularList solution = new CircularList();

ListNode head = null;

ListNode.printListNode(solution.hasCircle(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printListNode(solution.hasCircle(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4, 5 });

ListNode.printListNode(solution.hasCircle(head));

head = ListNode.arrayToListWithCircle(new int[] { 1 }, 0);

ListNode.printListNode(solution.hasCircle(head));

head = ListNode.arrayToListWithCircle(new int[] { 1, 2, 3, 4, 5 }, 3);

ListNode.printListNode(solution.hasCircle(head));

}

}

#### 5. Merge Two Sorted Linked Lists

/\*\*

\* Merge two sorted list into one sorted list.

\*/

public class MergeTwoSortedList {

public ListNode merge(ListNode headOne, ListNode headTwo) {

ListNode dummyHead = new ListNode(0);

ListNode current = dummyHead;

while (headOne != null && headTwo != null) {

if (headOne.value <= headTwo.value) {

current.next = headOne;

headOne = headOne.next;

} else {

current.next = headTwo;

headTwo = headTwo.next;

}

current = current.next;

}

if (headOne != null) {

current.next = headOne;

} else {

current.next = headTwo;

}

return dummyHead.next;

}

public static void main(String[] args) {

MergeTwoSortedList solution = new MergeTwoSortedList();

ListNode headOne = null;

ListNode headTwo = null;

ListNode.printList(solution.merge(headOne, headTwo));

headOne = null;

headTwo = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.merge(headOne, headTwo));

headOne = ListNode.arrayToList(new int[] { 4, 5, 6 });

headTwo = null;

ListNode.printList(solution.merge(headOne, headTwo));

headOne = ListNode.arrayToList(new int[] { 1, 3, 5, 7 });

headTwo = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.merge(headOne, headTwo));

}

}

#### 6. ReOrder Linked List

/\*\*

\* Given a Linked List N1 -> N2 -> N3 -> ... -> Nk -> null,

\* Convert it to N1 -> Nk -> N2 -> Nk-1 -> N3 -> Nk-2 -> ...

\*

\* Requirement: In place.

\*/

public class ReOrderList {

public ListNode reOrder(ListNode head) {

if (head == null) {

return head;

}

ListNode mid = findMiddle(head);

ListNode headTwo = mid.next;

ListNode headOne = head;

mid.next = null;

return merge(headOne, reverse(headTwo));

}

public ListNode findMiddle(ListNode head) {

if (head == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

}

return slow;

}

public ListNode reverse(ListNode head) {

if (head == null) {

return null;

}

ListNode current = head;

ListNode prev = null;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

public ListNode merge(ListNode headOne, ListNode headTwo) {

ListNode dummyHead = new ListNode(0);

ListNode current = dummyHead;

while (headOne != null && headTwo != null) {

current.next = headOne;

headOne = headOne.next;

current = current.next;

current.next = headTwo;

headTwo = headTwo.next;

current = current.next;

}

if (headOne != null) {

current.next = headOne;

} else {

current.next = headTwo;

}

return dummyHead.next;

}

public static void main(String[] args) {

ReOrderList solution = new ReOrderList();

ListNode head = null;

ListNode.printList(solution.reOrder(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.reOrder(head));

head = ListNode.arrayToList(new int[] { 1, 2 });

ListNode.printList(solution.reOrder(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.reOrder(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4, 5, 6 });

ListNode.printList(solution.reOrder(head));

}

}

#### 7. Partition Linked List

/\*\*

\* Given a Linked List and a target value x, partition it such that

\* all the nodes with value less than x are listed before all the

\* nodes equal to or larger than x. The relative order of the nodes

\* should be maintained in each of the two partitions.

\*/

public class PartitionList {

public ListNode partition(ListNode head, int target) {

if (head == null) {

return null;

}

ListNode dummySmall = new ListNode(0);

ListNode dummyLarge = new ListNode(0);

ListNode curSmall = dummySmall;

ListNode curLarge = dummyLarge;

while (head != null) {

if (head.value < target) {

curSmall.next = head;

curSmall = curSmall.next;

} else {

curLarge.next = head;

curLarge = curLarge.next;

}

head = head.next;

}

curSmall.next = dummyLarge.next;

curLarge.next = null;

return dummySmall.next;

}

public static void main(String[] args) {

PartitionList solution = new PartitionList();

int target = 5;

ListNode head = null;

ListNode.printList(solution.partition(head, target));

head = ListNode.arrayToList(new int[] { 4 });

ListNode.printList(solution.partition(head, target));

head = ListNode.arrayToList(new int[] { 6 });

ListNode.printList(solution.partition(head, target));

head = ListNode.arrayToList(new int[] { 1, 6, 5, 3, 2, 5, 2 });

ListNode.printList(solution.partition(head, target));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 4 - Binary Tree & Binary Search Tree

#### 1. TreeNode class and utilities

/\*\*

\* Binary tree node class and utilities.

\*/

public class TreeNode {

/\*

\* Special symbol for null node, used in pre-order reconstruction.

\*/

public static final String NULL\_NODE = "#";

/\*

\* A TreeNode contains an integer key, a reference to the

\* root of left subtree and a reference to root of the right subtree.

\*/

public int key;

public TreeNode left;

public TreeNode right;

public TreeNode(int key) {

this.key = key;

}

@Override

public String toString() {

return String.valueOf(key);

}

/\*\*

\* Construct the binary tree using pre-order traversal sequence.

\*

\* Note: the null node will be represented as a special value "#".

\*

\* If null node is ignored in the pre-order traversal sequence,

\* the constructed binary tree would not be UNIQUE.

\*

\* Example:

\* 2

\* / \

\* 1 3

\* / \ / \

\* null null null null

\*

\* The pre-order traversal sequence is:

\* {"2", "1", "#", "#", "3", "#", "#"}

\* Using the above sequence we can reconstruct the binary tree.

\*

\* @param preOrder

\* the preOrder traversal sequence

\* @return the root of the constructed binary tree

\*/

public static TreeNode reConstruct(String[] preOrder) {

if (preOrder == null || preOrder.length == 0) {

return null;

}

return reConstruct(preOrder, 0).node;

}

/\*

\* The real implementation of re-construction from preOrder traverse.

\*/

private static Result reConstruct(String[] preOrder, int left) {

if (preOrder[left].equals(NULL\_NODE)) {

return new Result(left + 1, null);

}

TreeNode root = new TreeNode(Integer.parseInt(preOrder[left]));

Result leftResult = reConstruct(preOrder, left + 1);

root.left = leftResult.node;

Result rightResult = reConstruct(preOrder, leftResult.index);

root.right = rightResult.node;

rightResult.node = root;

return rightResult;

}

/\*

\* Helper class for storing the intermediate information used in

\* reConstruct(String[] preOrder, int left) method.

\*/

private static class Result {

int index;

TreeNode node;

Result(int index, TreeNode node) {

this.index = index;

this.node = node;

}

}

/\*

\* pre-order traversal to print the given binary tree.

\* null node will be represented as "#".

\*/

public static String preOrder(TreeNode root) {

if (root == null) {

return NULL\_NODE;

}

StringBuilder preOrder = new StringBuilder();

preOrder(root, preOrder);

preOrder.deleteCharAt(preOrder.length() - 1);

return preOrder.toString();

}

private static void preOrder(TreeNode root, StringBuilder preOrder) {

if (root == null) {

preOrder.append(NULL\_NODE).append(" ");

return;

}

preOrder.append(root.key).append(" ");

preOrder(root.left, preOrder);

preOrder(root.right, preOrder);

}

}

#### 2. Binary Tree Pre-order, In-order, Post-order Iterative Traversals

/\*\*

\* Binary Tree preorder, inorder, postorder traversal iterative solutions.

\*/

public class TraversalIterative {

public List<Integer> preOrder(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

if (root == null) {

return result;

}

Deque<TreeNode> stack = new LinkedList<TreeNode>();

stack.offerFirst(root);

while (!stack.isEmpty()) {

TreeNode current = stack.pollFirst();

result.add(current.key);

if (current.right != null) {

stack.offerFirst(current.right);

}

if (current.left != null) {

stack.offerFirst(current.left);

}

}

return result;

}

public List<Integer> inOrder(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

if (root == null) {

return result;

}

Deque<TreeNode> stack = new LinkedList<TreeNode>();

TreeNode current = root;

while (current != null || !stack.isEmpty()) {

if (current != null) {

stack.offerFirst(current);

current = current.left;

} else {

current = stack.pollFirst();

result.add(current.key);

current = current.right;

}

}

return result;

}

public List<Integer> postOrderI(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

if (root == null) {

return result;

}

Deque<TreeNode> stack = new LinkedList<TreeNode>();

stack.offerFirst(root);

TreeNode prev = null;

while (!stack.isEmpty()) {

TreeNode current = stack.peekFirst();

if (prev == null || current == prev.left || current == prev.right) {

if (current.left != null) {

stack.offerFirst(current.left);

} else if (current.right != null) {

stack.offerFirst(current.right);

} else {

result.add(current.key);

stack.pollFirst();

}

} else if (current.left == prev && current.right != null) {

stack.offerFirst(current.right);

} else {

result.add(current.key);

stack.pollFirst();

}

prev = current;

}

return result;

}

public List<Integer> postOrderII(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

if (root == null) {

return result;

}

Deque<TreeNode> stack1 = new LinkedList<TreeNode>();

Deque<TreeNode> stack2 = new LinkedList<TreeNode>();

stack1.offerFirst(root);

while (!stack1.isEmpty()) {

TreeNode current = stack1.pollFirst();

stack2.offerFirst(current);

if (current.left != null) {

stack1.offerFirst(current.left);

}

if (current.right != null) {

stack1.offerFirst(current.right);

}

}

while (!stack2.isEmpty()) {

result.add(stack2.pollFirst().key);

}

return result;

}

public static void main(String[] args) {

TraversalIterative solution = new TraversalIterative();

TreeNode root = TreeNode.reConstruct(new String[] { "1", "#", "#" });

System.out.println(solution.preOrder(root).toString());

System.out.println(solution.inOrder(root).toString());

System.out.println(solution.postOrderI(root).toString());

System.out.println(solution.postOrderII(root).toString());

/\*

\* 3

\* / \

\* 2 5

\* / / \

\* 1 4 6

\*/

root = TreeNode.reConstruct(new String[] { "3", "2", "1", "#", "#", "#", "5", "4", "#", "#",

"6", "#", "#" });

System.out.println(solution.preOrder(root).toString());

System.out.println(solution.inOrder(root).toString());

System.out.println(solution.postOrderI(root).toString());

System.out.println(solution.postOrderII(root).toString());

}

}

#### 3. Judge Whether A Binary Tree Is Binary Search Tree

/\*\*

\* Judge if a binary tree is a binary search tree.

\* Assumption:

\* 1). a binary search tree does not accept duplicate values.

\*/

public class JudgeBinarySearchTree {

public boolean isBST(TreeNode root) {

return isBST(root, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

}

public boolean isBST(TreeNode root, int lower, int upper) {

if (root == null) {

return true;

}

if (root.key <= lower || root.key >= upper) {

return false;

}

return isBST(root.left, lower, root.key) && isBST(root.right, root.key, upper);

}

public static void main(String[] args) {

JudgeBinarySearchTree solution = new JudgeBinarySearchTree();

TreeNode root = null;

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "1", "#", "#" });

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "2", "1", "#", "#", "3", "#", "#" });

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "2", "2", "#", "#", "3", "#", "#" });

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "1", "3", "#", "#", "2", "#", "#" });

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "3", "5", "#", "#", "4", "#", "6", "#", "#" });

System.out.println(solution.isBST(root));

root = TreeNode.reConstruct(new String[] { "5", "3", "#", "#", "8", "7", "#", "#", "#" });

System.out.println(solution.isBST(root));

}

}

#### 4. Judge Whether Two Binary Trees Are Identical

/\*\*

\* Judge if two binary trees are identical.

\*/

public class JudgeIdentical {

public boolean isIdentical(TreeNode rootOne, TreeNode rootTwo) {

if (rootOne == null && rootTwo == null) {

return true;

} else if (rootOne == null || rootTwo == null) {

return false;

} else if (rootOne.key != rootTwo.key) {

return false;

} else {

return isIdentical(rootOne.left, rootTwo.left) && isIdentical(rootOne.right, rootTwo.right);

}

}

public static void main(String[] args) {

JudgeIdentical solution = new JudgeIdentical();

TreeNode rootOne = null;

TreeNode rootTwo = null;

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = null;

rootTwo = TreeNode.reConstruct(new String[] { "1", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "1", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "1", "2", "#", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "1", "2", "#", "#", "3", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "1", "2", "#", "#", "3", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "1", "2", "#", "#", "3", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "1", "2", "#", "#", "4", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

}

}

#### 5. Judge Whether A Binary Tree Is Symmetric

/\*\*

\* Judge if a binary tree is a mirror of itself.

\*/

public class JudgeSymmetric {

public boolean isSymmetric(TreeNode root) {

if (root == null) {

return true;

}

return isSymmetric(root.left, root.right);

}

public boolean isSymmetric(TreeNode rootOne, TreeNode rootTwo) {

if (rootOne == null && rootTwo == null) {

return true;

} else if (rootOne == null || rootTwo == null) {

return false;

} else if (rootOne.key != rootTwo.key) {

return false;

} else {

return isSymmetric(rootOne.left, rootTwo.right) && isSymmetric(rootOne.right, rootTwo.left);

}

}

public static void main(String[] args) {

JudgeSymmetric solution = new JudgeSymmetric();

TreeNode root = null;

System.out.println(solution.isSymmetric(root));

root = TreeNode.reConstruct(new String[] { "1", "#", "#" });

System.out.println(solution.isSymmetric(root));

root = TreeNode.reConstruct(new String[] { "2", "1", "#", "#", "#" });

System.out.println(solution.isSymmetric(root));

root = TreeNode.reConstruct(new String[] { "2", "1", "#", "#", "1", "#", "#" });

System.out.println(solution.isSymmetric(root));

root = TreeNode.reConstruct(new String[] { "2", "1", "3", "#", "#", "4", "#", "#", "1", "4",

"#", "#", "3", "#", "#" });

System.out.println(solution.isSymmetric(root));

root = TreeNode.reConstruct(new String[] { "2", "1", "3", "#", "#", "4", "#", "#", "1", "3",

"#", "#", "4", "#", "#" });

System.out.println(solution.isSymmetric(root));

}

}

#### 6. Judge Whether Two Binary Trees Are Tweaked Identical

/\*\*

\* Let’s assume if we tweak the left subtree with right subtree of an arbitrary

\* node in a binary tree, then the “structure” of the tree are not changed.

\* Determine whether two binary trees’ structures are identical.

\*/

public class JudgeTweakIdentical {

public boolean isIdentical(TreeNode rootOne, TreeNode rootTwo) {

if (rootOne == null && rootTwo == null) {

return true;

} else if (rootOne == null || rootTwo == null) {

return false;

} else if (rootOne.key != rootTwo.key) {

return false;

} else {

return isIdentical(rootOne.left, rootTwo.left) && isIdentical(rootOne.right, rootTwo.right)

|| isIdentical(rootOne.left, rootTwo.right) && isIdentical(rootOne.right, rootTwo.left);

}

}

public static void main(String[] args) {

JudgeTweakIdentical solution = new JudgeTweakIdentical();

TreeNode rootOne = null;

TreeNode rootTwo = null;

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "1", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "1", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "1", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "2", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "5", "4", "#", "#", "8", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "5", "8", "#", "#", "4", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

rootOne = TreeNode.reConstruct(new String[] { "5", "4", "#", "#", "#" });

rootTwo = TreeNode.reConstruct(new String[] { "5", "8", "#", "#", "4", "#", "#" });

System.out.println(solution.isIdentical(rootOne, rootTwo));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 5 - Heap & Graph + Search Algorithms

#### 1. GraphNode class and sample graphs

/\*\*

\* GraphNode class represents a node in an undirected graph.

\* Sample graphs for tests are also provided.

\*/

public class GraphNode {

// Each node in an undirected graph maintains a list of neighbors.

public int id;

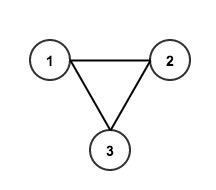
public List<GraphNode> neighbors;

public GraphNode(int id) {

this.id = id;

neighbors = new ArrayList<GraphNode>();

}



public static List<GraphNode> sampleGraphOne() {

GraphNode nodeOne = new GraphNode(1);

GraphNode nodeTwo = new GraphNode(2);

GraphNode nodeThree = new GraphNode(3);

nodeOne.neighbors.add(nodeTwo);

nodeOne.neighbors.add(nodeThree);

nodeTwo.neighbors.add(nodeOne);

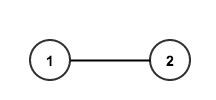
nodeTwo.neighbors.add(nodeThree);

nodeThree.neighbors.add(nodeOne);

nodeThree.neighbors.add(nodeTwo);

return Arrays.asList(new GraphNode[] { nodeOne, nodeTwo, nodeThree });

}



public static List<GraphNode> sampleGraphTwo() {

GraphNode nodeOne = new GraphNode(1);

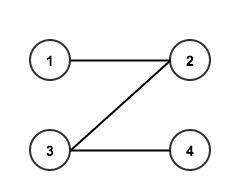
GraphNode nodeTwo = new GraphNode(2);

nodeOne.neighbors.add(nodeTwo);

nodeTwo.neighbors.add(nodeOne);

return Arrays.asList(new GraphNode[] { nodeOne, nodeTwo });

}



public static List<GraphNode> sampleGraphThree() {

GraphNode nodeOne = new GraphNode(1);

GraphNode nodeTwo = new GraphNode(2);

GraphNode nodeThree = new GraphNode(3);

GraphNode nodeFour = new GraphNode(4);

nodeOne.neighbors.add(nodeTwo);

nodeTwo.neighbors.add(nodeOne);

nodeTwo.neighbors.add(nodeThree);

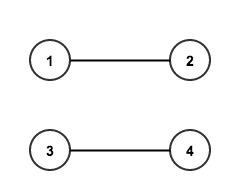
nodeThree.neighbors.add(nodeTwo);

nodeThree.neighbors.add(nodeFour);

nodeFour.neighbors.add(nodeThree);

return Arrays.asList(new GraphNode[] { nodeOne, nodeTwo, nodeThree, nodeFour });

}



public static List<GraphNode> sampleGraphFour() {

GraphNode nodeOne = new GraphNode(1);

GraphNode nodeTwo = new GraphNode(2);

GraphNode nodeThree = new GraphNode(3);

GraphNode nodeFour = new GraphNode(4);

nodeOne.neighbors.add(nodeTwo);

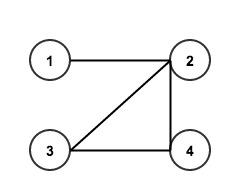
nodeTwo.neighbors.add(nodeOne);

nodeThree.neighbors.add(nodeFour);

nodeFour.neighbors.add(nodeThree);

return Arrays.asList(new GraphNode[] { nodeOne, nodeTwo, nodeThree, nodeFour });

}



public static List<GraphNode> sampleGraphFive() {

GraphNode nodeOne = new GraphNode(1);

GraphNode nodeTwo = new GraphNode(2);

GraphNode nodeThree = new GraphNode(3);

GraphNode nodeFour = new GraphNode(4);

nodeOne.neighbors.add(nodeTwo);

nodeTwo.neighbors.add(nodeOne);

nodeTwo.neighbors.add(nodeThree);

nodeTwo.neighbors.add(nodeFour);

nodeThree.neighbors.add(nodeFour);

nodeThree.neighbors.add(nodeTwo);

nodeFour.neighbors.add(nodeThree);

nodeFour.neighbors.add(nodeTwo);

return Arrays.asList(new GraphNode[] { nodeOne, nodeTwo, nodeThree, nodeFour });

}

}

#### 2. Determine Whether An Undirected Graph Is Bipartite

/\*\*

\* Check if an undirected graph is bipartite.

\* The graph will be represented by a List of Node.

\* Each Node maintains the list of neighbors of itself.

\*/

public class Bipartite {

public boolean isBipartite(List<GraphNode> graph) {

assert graph != null;

if (graph.isEmpty()) {

return true;

}

// the hashmap has two purpose:

// 1). only contains the visited nodes

// 2). for each visited nodes, mark the group number(either 0 or 1).

HashMap<GraphNode, Integer> groups = new HashMap<GraphNode, Integer>();

// do BFS for each of the nodes,

// if conflict is found for any of the nodes, the graph is not bipartite.

for (GraphNode node : graph) {

if (!searchAndMark(node, groups)) {

return false;

}

}

return true;

}

private boolean searchAndMark(GraphNode node, HashMap<GraphNode, Integer> groups) {

// if the node is already visited, just return true.

if (groups.containsKey(node)) {

return true;

}

// BFS starting from the current node.

Queue<GraphNode> queue = new LinkedList<GraphNode>();

queue.offer(node);

groups.put(node, 0);

while (!queue.isEmpty()) {

GraphNode current = queue.poll();

int group = groups.get(current);

int neighborGroup = group == 0 ? 1 : 0;

for (GraphNode neighbor : current.neighbors) {

if (!groups.containsKey(neighbor)) {

// if we enter this block, means the neighbor has not been visited yet,

// then we append this neighbor to the queue, and mark the group for it.

queue.offer(neighbor);

groups.put(neighbor, neighborGroup);

} else if (groups.get(neighbor).intValue() != neighborGroup) {

// if we enter this block, we find a conflict.

return false;

}

}

}

return true;

}

public static void main(String[] args) {

Bipartite solution = new Bipartite();

System.out.println(solution.isBipartite(GraphNode.sampleGraphOne()));

System.out.println(solution.isBipartite(GraphNode.sampleGraphTwo()));

System.out.println(solution.isBipartite(GraphNode.sampleGraphThree()));

System.out.println(solution.isBipartite(GraphNode.sampleGraphFour()));

System.out.println(solution.isBipartite(GraphNode.sampleGraphFive()));

}

}

#### 3. Smallest K Elements In An Unsorted Array

/\*\*

\* Find the smallest k elements in an unsorted array.

\*/

public class SmallestK {

public int[] smallestK(int[] array, int k) {

assert array != null && k >= 0 && k <= array.length;

if (k == 0) {

return new int[0];

}

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k,

// maxHeap's comparator

new Comparator<Integer>() {

@Override

public int compare(Integer i1, Integer i2) {

if (i1 < i2) {

return 1;

} else if (i1 > i2) {

return -1;

} else {

return 0;

}

}

});

for (int index = 0; index < array.length; index++) {

if (index < k) {

// for the first k elements, just offer them to the maxHeap.

maxHeap.offer(array[index]);

} else if (array[index] < maxHeap.peek()) {

// for the other elements, compare its value with the peek

// of the maxHeap to determine if it can be inserted.

maxHeap.poll();

maxHeap.offer(array[index]);

}

}

return toSortedArray(maxHeap);

}

private int[] toSortedArray(PriorityQueue<Integer> maxHeap) {

assert maxHeap != null;

int[] array = new int[maxHeap.size()];

for (int index = array.length - 1; index >= 0; index--) {

array[index] = maxHeap.poll();

}

return array;

}

public static void main(String[] args) {

SmallestK solution = new SmallestK();

int k = 3;

int[] array = new int[] { 1, 2, 3, 4 };

System.out.println(Arrays.toString(solution.smallestK(array, k)));

array = new int[] { 3, 1, 2, 4, 5 };

System.out.println(Arrays.toString(solution.smallestK(array, k)));

array = new int[] { 7, 9, 10, 1, 3, 2 };

System.out.println(Arrays.toString(solution.smallestK(array, k)));

}

}

#### 4. K-th smallest Element In Row Sorted-Column Sorted Matrix

/\*\*

\* Find the kth smallest number in a matrix.

\* The elements in each row of the matrix are sorted.

\* The elements in each column of the matrix are also sorted.

\*/

public class KthSmallestInMatrix {

private static class Entry {

int x;

int y;

int value;

public Entry(int x, int y, int value) {

this.x = x;

this.y = y;

this.value = value;

}

// this method needs to be overridden to be used by HashSet.

// we need to guarantee two things:

// 1). for any equal Entry objects, their hashCode should be same.

// 2). provide as less conflicts as possible for not equal objects.

@Override

public int hashCode() {

return x \* 31 + y;

}

// this method needs to be overridden to be used by HashSet.

// if two Entry objects have the same x Index and y Index,

// we think them equal.

@Override

public boolean equals(Object obj) {

if (obj == null || !(obj instanceof Entry)) {

return false;

}

if (this == obj) {

return true;

}

Entry another = (Entry) obj;

return this.x == another.x && this.y == another.y;

}

}

public int kth(final int[][] matrix, int k) {

sanityCheck(matrix, k);

int len = matrix.length;

PriorityQueue<Entry> minHeap = new PriorityQueue<Entry>(k,

// minHeap comparator for Entry class.

// use a minHeap because we need to expand

// the smallest one every time.

new Comparator<Entry>() {

public int compare(Entry e1, Entry e2) {

if (matrix[e1.x][e1.y] < matrix[e2.x][e2.y]) {

return -1;

} else if (matrix[e1.x][e1.y] > matrix[e2.x][e2.y]) {

return 1;

} else {

return 0;

}

}

});

// use a HashSet to mark the visited Entries,

// if the Entry is in the set, means it is in the heap.

HashSet<Entry> visited = new HashSet<Entry>();

int result = Integer.MIN\_VALUE;

Entry start = new Entry(0, 0, matrix[0][0]);

minHeap.offer(start);

visited.add(start);

while (k > 0) {

Entry cur = minHeap.poll();

result = cur.value;

// there are two neighbors need to be examined.

// if the neighbor has been visited, means it is already in the heap,

// we do not need to do anything;

// if the neighbor has not been visited,

// we need to insert it into the heap.

if (cur.x + 1 < len) {

Entry down = new Entry(cur.x + 1, cur.y, matrix[cur.x + 1][cur.y]);

if (!visited.contains(down)) {

minHeap.offer(down);

visited.add(down);

}

}

if (cur.y + 1 < len) {

Entry right = new Entry(cur.x, cur.y + 1, matrix[cur.x][cur.y + 1]);

if (!visited.contains(right)) {

minHeap.offer(right);

visited.add(right);

}

}

k--;

}

return result;

}

// usually for this kind of problem, we do not need to have a complete

// sanity check, we can confirm with interviewer for assumptions that

// the input is already clean.

private void sanityCheck(int[][] matrix, int k) {

assert matrix != null && matrix.length > 0;

int len = matrix.length;

assert k >= 1 && k <= len \* len;

for (int index = 0; index < len; index++) {

assert matrix[index] != null && matrix[index].length == len;

}

}

public static void main(String[] args) {

KthSmallestInMatrix solution = new KthSmallestInMatrix();

int[][] matrix = { { 1, 2, 7, 8 }, { 3, 5, 9, 13 }, { 4, 6, 10, 14 }, { 6, 8, 12, 15 } };

for (int k = 1; k <= 10; k++) {

System.out.println(solution.kth(matrix, k));

}

}

}

#### 5. Print Values Of Nodes Layer By Layer In Binary Tree

/\*\*

\* Print values of the binary tree layer by layer.

\*/

public class BinaryTreeBFS {

public void BFS(TreeNode root) {

if (root == null) {

System.out.println(root);

return;

}

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

for (int index = 0; index < size; index++) {

TreeNode curNode = queue.poll();

if (curNode.left != null) {

queue.offer(curNode.left);

}

if (curNode.right != null) {

queue.offer(curNode.right);

}

System.out.print(curNode.key);

}

System.out.println();

}

}

public static void main(String[] args) {

BinaryTreeBFS solution = new BinaryTreeBFS();

TreeNode root = null;

solution.BFS(root);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

root = TreeNode.reConstruct(new String[] { "2", "1", "#", "#", "3", "#", "#" });

solution.BFS(root);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

root = TreeNode.reConstruct(new String[] { "4", "2", "1", "#", "#", "3", "#", "#", "7", "#",

"#" });

solution.BFS(root);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 6 - DFS & Hashtable

#### 1. All SubSets Of A Set Without Duplicates

/\*\*

\* Print all the subsets of a set.

\* Assumption:

\* 1). the set is represented by a string.

\* 2). there is no duplicate elements in the set.

\*/

public class SubSetI {

public void subSet(String set) {

assert set != null;

StringBuilder builder = new StringBuilder();

subSet(set, 0, builder);

}

private void subSet(String set, int index, StringBuilder builder) {

if (index == set.length()) {

// only when we have examined all the elements in the set,

// we can terminate DFS and print the solution.

System.out.println(builder.toString());

} else {

// for each of the element in the set, either we append it

// to the solution or we do not append it.

subSet(set, index + 1, builder);

subSet(set, index + 1, builder.append(set.charAt(index)));

// when falling back to the previous state, we need to cleanup

// the change made to the solution at current state.

builder.deleteCharAt(builder.length() - 1);

}

}

public void anotherSubSet(String set) {

assert set != null;

StringBuilder builder = new StringBuilder();

anotherSubSet(set, 0, builder);

}

private void anotherSubSet(String set, int index, StringBuilder builder) {

System.out.println(builder.toString());

// another way of DFS

// at each state, what we can choose is which is the next

// element to append, before we choose the next element(if any),

// we need to print the current solution first.

for (int i = index; i < set.length(); i++) {

builder.append(set.charAt(i));

anotherSubSet(set, i + 1, builder);

builder.deleteCharAt(builder.length() - 1);

}

}

public static void main(String[] args) {

SubSetI solution = new SubSetI();

String set = "";

solution.subSet(set);

solution.anotherSubSet(set);

set = "abc";

solution.subSet(set);

solution.anotherSubSet(set);

}

}

#### 2. All Valid Permutations Of Parentheses

/\*\*

\* Print all the valid permutation of k pairs of parentheses.

\* For Example:

\* k = 3, the solution should be:

\* ((()))

\* (()())

\* (())()

\* ()(())

\* ()()()

\*/

public class ValidParentheses {

public void validParentheses(int k) {

char[] solution = new char[2 \* k];

validParentheses(k, 0, 0, solution);

}

private void validParentheses(int k, int left, int right, char[] solution) {

if (left + right == 2 \* k) {

System.out.println(new String(solution));

return;

}

// left = how many ( we already have

// right = how many ) we already have

// only when we have some ( left, we can append (

if (left < k) {

solution[left + right] = '(';

validParentheses(k, left + 1, right, solution);

}

// only when we have less ) then (, we can append )

if (right < left) {

solution[left + right] = ')';

validParentheses(k, left, right + 1, solution);

}

}

public static void main(String[] args) {

ValidParentheses solution = new ValidParentheses();

int k = 0;

solution.validParentheses(k);

k = 3;

solution.validParentheses(k);

}

}

#### 3. Various Combinations Of Coins

/\*\*

\* Given a target value of cents and a list of possible coins,

\* print all the combinations of coins, the sum value is the target.

\*/

public class VariousCombination {

// Assumptions:

// 1). the value of coins are sorted by descending order

// 2). there are no duplicate values in the value of coins

// 3). there is no 0 value in the coins

public void combinations(int target, int[] coins) {

assert target >= 0 && coins != null && coins.length > 0;

int[] solution = new int[coins.length];

combinations(target, coins, solution, 0);

}

// the value in solution array means how many coins we have

// for the corresponding coin in the coins array.

// For Example,

// coins = {25, 10, 5, 2, 1}

// solution = { 1, 2, 3, 4, 5}

// means we have 1 x 25 cents coins, 2 x 10 cents coins, 3 x 5 cents coins,

// 4 x 2 cents coins, 5 x 1 cent coins.

private void combinations(int target, int[] coins, int[] solution, int index) {

if (index == coins.length - 1) {

// early termination, the remainder of target divided by

// the smallest coin must be 0 for a valid solution.

if (target % coins[index] == 0) {

solution[index] = target / coins[index];

printSolution(coins, solution);

}

return;

}

// how many coins we can have for the current target.

int maxNum = target / coins[index];

for (int num = 0; num <= maxNum; num++) {

solution[index] = num;

combinations(target - num \* coins[index], coins, solution, index + 1);

}

}

private void printSolution(int[] coins, int[] solution) {

assert coins.length == solution.length;

StringBuilder builder = new StringBuilder();

for (int i = 0; i < coins.length; i++) {

builder.append("[").append(coins[i]).append(" cent coins: ").append(solution[i]).append("] ");

}

System.out.println(builder.toString());

}

public static void main(String[] args) {

VariousCombination solution = new VariousCombination();

int[] coins = new int[] { 25, 10, 5, 2, 1 };

int target = 0;

solution.combinations(target, coins);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

target = 11;

solution.combinations(target, coins);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

target = 99;

solution.combinations(target, coins);

System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

}

}

#### 4. Top K Frequent Words

/\*\*

\* Find the top k frequent words from a list of words.

\*/

public class TopKFrequent {

private static class WordFreq implements Comparable<WordFreq> {

String word;

long frequency;

public WordFreq(String word, long frequency) {

this.word = word;

this.frequency = frequency;

}

@Override

public String toString() {

return word + ":" + frequency;

}

@Override

public int compareTo(WordFreq another) {

if (this.frequency == another.frequency) {

return 0;

} else if (this.frequency < another.frequency) {

return -1;

} else {

return 1;

}

}

}

public List<WordFreq> topK(int k, List<String> words) {

assert k >= 1 && words != null;

// construct the map between the words and their frequency.

Map<String, Long> freqMap = frequencyMap(words);

// from the frequency map construct the heap contains top K.

PriorityQueue<WordFreq> freqHeap = frequencyHeap(k, freqMap);

return topList(freqHeap);

}

private List<WordFreq> topList(PriorityQueue<WordFreq> freqHeap) {

List<WordFreq> result = new ArrayList<WordFreq>();

while (!freqHeap.isEmpty()) {

result.add(freqHeap.poll());

}

return result;

}

// construct the min heap containing the top k.

private PriorityQueue<WordFreq> frequencyHeap(int k, Map<String, Long> freqMap) {

ArrayList<WordFreq> heapifyCandidate = new ArrayList<WordFreq>(k);

PriorityQueue<WordFreq> minHeap = null;

int count = 0;

for (Entry<String, Long> entry : freqMap.entrySet()) {

if (count < k - 1) {

// for the first k elements, add them to the list first

heapifyCandidate.add(new WordFreq(entry.getKey(), entry.getValue()));

} else if (count == k - 1) {

// when there are k elements in the list, use the list to construct the

// min heap.

heapifyCandidate.add(new WordFreq(entry.getKey(), entry.getValue()));

minHeap = new PriorityQueue<WordFreq>(heapifyCandidate);

} else if (entry.getValue() > minHeap.peek().frequency) {

// for the other elements, we need to compare it with the peek to

// determine

// if they should be inserted into the heap.

minHeap.poll();

minHeap.offer(new WordFreq(entry.getKey(), entry.getValue()));

}

count++;

}

// if we do not even have k elements in the frequency map.

if (freqMap.size() <= k) {

minHeap = new PriorityQueue<WordFreq>(heapifyCandidate);

}

return minHeap;

}

private Map<String, Long> frequencyMap(List<String> words) {

Map<String, Long> freqMap = new HashMap<String, Long>();

for (String word : words) {

if (!freqMap.containsKey(word)) {

// if we have not seen the word before, count the frequency as 1.

freqMap.put(word, 1L);

} else {

// if we have seen the word before, increment the frequency by 1.

freqMap.put(word, freqMap.get(word) + 1L);

}

}

return freqMap;

}

public static void main(String[] args) {

TopKFrequent solution = new TopKFrequent();

int k = 3;

List<String> words = Arrays.asList();

System.out.println(solution.topK(k, words));

words = Arrays.asList("a", "b", "a");

System.out.println(solution.topK(k, words));

words = Arrays.asList("a", "b", "c", "d", "c", "b", "a", "a", "b", "a");

System.out.println(solution.topK(k, words));

}

}

#### 5. Missing Number

/\*\*

\* Given an array of size of n - 1 containing the number

\* from 1 to n(no duplicates), there is one number missing,

\* find the number in O(n).

\*/

public class MissingNumber {

public int findMissingHashMap(int n, int[] array) {

assert array != null && n >= 1 && n == array.length + 1;

HashSet<Integer> set = new HashSet<Integer>();

for (int number : array) {

set.add(number);

}

for (int number = 1; number < n; number++) {

if (!set.contains(number)) {

return number;

}

}

return n;

}

public int findMissing(int n, int[] array) {

assert array != null && n >= 1 && n == array.length + 1;

int sum = 0;

for (int i = 0; i < array.length; i++) {

sum += array[i];

}

return n \* (n + 1) / 2 - sum;

}

public static void main(String[] args) {

MissingNumber solution = new MissingNumber();

int[] array = new int[0];

int n = 1;

System.out.println(solution.findMissingHashMap(n, array));

System.out.println(solution.findMissing(n, array));

array = new int[] { 1, 2 };

n = 3;

System.out.println(solution.findMissingHashMap(n, array));

System.out.println(solution.findMissing(n, array));

array = new int[] { 3, 2 };

n = 3;

System.out.println(solution.findMissingHashMap(n, array));

System.out.println(solution.findMissing(n, array));

}

}

#### 6. Common Numbers Between Two Sorted Arrays

/\*\*

\* Find the common numbers in two sorted array.

\* Assumption:

\* 1). there is no duplicate number in each of the arrays.

\*/

public class CommonNumbers {

public List<Integer> commonNumbersHashSet(int[] array1, int[] array2) {

assert array1 != null && array2 != null;

HashSet<Integer> set = new HashSet<Integer>();

for (int number : array1) {

set.add(number);

}

List<Integer> result = new ArrayList<Integer>();

for (int number : array2) {

if (set.contains(number)) {

result.add(number);

}

}

return result;

}

public List<Integer> commonNumbers(int[] array1, int[] array2) {

assert array1 != null && array2 != null;

int i1 = 0;

int i2 = 0;

List<Integer> result = new ArrayList<Integer>();

while (i1 < array1.length && i2 < array2.length) {

if (array1[i1] == array2[i2]) {

result.add(array1[i1]);

i1++;

i2++;

} else if (array1[i1] < array2[i2]) {

i1++;

} else {

i2++;

}

}

return result;

}

public static void main(String[] args) {

CommonNumbers solution = new CommonNumbers();

int[] array1 = new int[0];

int[] array2 = new int[] { 1, 2 };

System.out.println(solution.commonNumbersHashSet(array1, array2));

System.out.println(solution.commonNumbers(array1, array2));

array1 = new int[] { 2, 3, 4 };

array2 = new int[] { 1, 2, 4 };

System.out.println(solution.commonNumbersHashSet(array1, array2));

System.out.println(solution.commonNumbers(array1, array2));

array1 = new int[] { 2, 3, 4 };

array2 = new int[] { 1, 5 };

System.out.println(solution.commonNumbersHashSet(array1, array2));

System.out.println(solution.commonNumbers(array1, array2));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 7 - Midterm 1

#### 1. All Permutations Of A String With/WithOut Duplicates

/\*\*

\* Find all permutations of a string with/without duplicate letters.

\*/

public class Permutations {

public List<String> perWithOutDup(String str) {

assert str != null;

char[] current = str.toCharArray();

List<String> result = new ArrayList<String>();

perWithOutDup(result, current, 0);

return result;

}

private void perWithOutDup(List<String> result, char[] current, int index) {

if (index == current.length) {

result.add(new String(current));

} else {

for (int i = index; i < current.length; i++) {

swap(current, i, index);

perWithOutDup(result, current, index + 1);

swap(current, i, index);

}

}

}

public List<String> perWithDup(String str) {

assert str != null;

char[] array = str.toCharArray();

List<String> result = new ArrayList<String>();

perWithDup(result, array, 0);

return result;

}

private void perWithDup(List<String> result, char[] array, int index) {

if (index == array.length) {

result.add(new String(array));

return;

}

HashSet<Character> used = new HashSet<Character>();

for (int i = index; i < array.length; i++) {

if (!used.contains(array[i])) {

used.add(array[i]);

swap(array, index, i);

perWithDup(result, array, index + 1);

swap(array, index, i);

}

}

}

private void swap(char[] array, int left, int right) {

char temp = array[left];

array[left] = array[right];

array[right] = temp;

}

public static void main(String[] args) {

Permutations solution = new Permutations();

String str = "";

System.out.println(solution.perWithOutDup(str));

System.out.println(solution.perWithDup(str));

str = "abc";

System.out.println(solution.perWithOutDup(str));

System.out.println(solution.perWithDup(str));

str = "abba";

System.out.println(solution.perWithDup(str));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 8 - Bit representation of a number and bit operation

#### 1. Determine If An Integer Is Power Of 2

/\*\*

\* Determine if an integer is power of 2.

\*/

public class PowerOfTwo {

public boolean isPowerOfTwoI(int number) {

if (number <= 0) {

return false;

}

while ((number & 1) == 0) {

number >>>= 1;

}

return number == 1;

}

public boolean isPowerOfTwoII(int number) {

if (number <= 0) {

return false;

}

int count = 0;

while (number > 0) {

count += number & 1;

number >>>= 1;

}

return count == 1;

}

public boolean isPowerOfTwoIII(int number) {

if (number <= 0) {

return false;

}

return (number & (number - 1)) == 0;

}

public static void main(String[] args) {

PowerOfTwo solution = new PowerOfTwo();

int number = -1;

System.out.println(solution.isPowerOfTwoI(number));

System.out.println(solution.isPowerOfTwoII(number));

System.out.println(solution.isPowerOfTwoIII(number));

number = 0;

System.out.println(solution.isPowerOfTwoI(number));

System.out.println(solution.isPowerOfTwoII(number));

System.out.println(solution.isPowerOfTwoIII(number));

number = 1;

System.out.println(solution.isPowerOfTwoI(number));

System.out.println(solution.isPowerOfTwoII(number));

System.out.println(solution.isPowerOfTwoIII(number));

number = 4;

System.out.println(solution.isPowerOfTwoI(number));

System.out.println(solution.isPowerOfTwoII(number));

System.out.println(solution.isPowerOfTwoIII(number));

number = 6;

System.out.println(solution.isPowerOfTwoI(number));

System.out.println(solution.isPowerOfTwoII(number));

System.out.println(solution.isPowerOfTwoIII(number));

}

}

#### 2. Determine The Number Of Different Bits Between 2 Integers

/\*\*

\* Determine the number of different bits between two integers.

\*/

public class DiffBits {

public int diffBits(int num1, int num2) {

num1 ^= num2;

int count = 0;

while (num1 != 0) {

if ((num1 & 1) == 1) {

count++;

}

num1 >>>= 1;

}

return count;

}

public static void main(String[] args) {

DiffBits solution = new DiffBits();

int num1 = 1;

int num2 = 2;

System.out.println(solution.diffBits(num1, num2));

num1 = 0;

num2 = 127;

System.out.println(solution.diffBits(num1, num2));

num1 = Integer.MAX\_VALUE;

num2 = 127;

System.out.println(solution.diffBits(num1, num2));

}

}

#### 3. Determine Whether A ASCII Word Contains All Unique Characters

/\*\*

\* Determine if the letters in a word are all unique.

\* Assumption:

\* We are using ASCII encoding and the number of valid letters

\* is 256, encoded from 0 to 255.

\*/

public class AllUniqueLetters {

public boolean allUniqueLetters(String word) {

assert word != null;

// for a size k bitmap, the number of integers needed is (k - 1) / 32 + 1

int[] vec = new int[(256 - 1) / 32 + 1];

for (int i = 0; i < word.length(); i++) {

char temp = word.charAt(i);

if ((vec[temp / 32] >>> (temp % 32) & 1) != 0) {

return false;

}

vec[temp / 32] |= 1 << (temp % 32);

}

return true;

}

public static void main(String[] args) {

AllUniqueLetters solution = new AllUniqueLetters();

String word = "";

System.out.println(solution.allUniqueLetters(word));

word = "abcde";

System.out.println(solution.allUniqueLetters(word));

word = "abcaa";

System.out.println(solution.allUniqueLetters(word));

}

}

#### 4. Get The Hexadecimal Representation String Of A Non-negative Integer

/\*\*

\* Convert a non-negative integer to its hexadecimal representation.

\*/

public class Hex {

public String hex(int number) {

String prefix = "0x";

if (number == 0) {

return prefix + 0;

}

StringBuilder builder = new StringBuilder();

while (number > 0) {

int remainder = number % 16;

if (remainder <= 9) {

builder.append((char) ('0' + remainder));

} else {

builder.append((char) ('A' + (remainder - 10)));

}

number >>>= 4;

}

return prefix + builder.reverse().toString();

}

public static void main(String[] args) {

Hex solution = new Hex();

int number = 0;

System.out.println(solution.hex(number));

number = 251;

System.out.println(solution.hex(number));

number = Integer.MAX\_VALUE;

System.out.println(solution.hex(number));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 11 - Dynamic Programming I

#### 1. Fibonacci Number

/\*\*

\* Get the nth Fibonacci Number.

\*/

public class Fibonacci {

// recursive solution.

public int fibonacciI(int nth) {

if (nth <= 0) {

return 0;

}

if (nth == 1 || nth == 2) {

return 1;

}

return fibonacciI(nth - 2) + fibonacciI(nth - 1);

}

// dynamic programming solution with O(n) space.

public int fibonacciII(int nth) {

if (nth <= 0) {

return 0;

}

int[] fibonacci = new int[nth + 1];

fibonacci[0] = 0;

fibonacci[1] = 1;

for (int i = 2; i <= nth; i++) {

fibonacci[i] = fibonacci[i - 2] + fibonacci[i - 1];

}

return fibonacci[nth];

}

// dynamic programming solution with O(1) space.

public int fibonacciIII(int nth) {

if (nth <= 0) {

return 0;

}

int prev = 0;

int current = 1;

for (int i = 1; i < nth; i++) {

int next = prev + current;

prev = current;

current = next;

}

return current;

}

public static void main(String[] args) {

Fibonacci solution = new Fibonacci();

int nth = -1;

System.out.println(solution.fibonacciI(nth));

System.out.println(solution.fibonacciII(nth));

System.out.println(solution.fibonacciIII(nth));

nth = 1;

System.out.println(solution.fibonacciI(nth));

System.out.println(solution.fibonacciII(nth));

System.out.println(solution.fibonacciIII(nth));

nth = 6;

System.out.println(solution.fibonacciI(nth));

System.out.println(solution.fibonacciII(nth));

System.out.println(solution.fibonacciIII(nth));

}

}

#### 2. Longest Ascending Subarray

/\*\*

\* Given an unsorted array, find the length of the longest subarray in which

\* the numbers are in ascending order.

\*/

public class LongestAscendingSubarray {

// O(n ^ 2) solution, for each index of the original array, find the

// longest ascending subarray starting with that index.

public int longestI(int[] array) {

assert array != null;

int longest = 0;

for (int start = 0; start < array.length; start++) {

int curLongest = 1;

for (int i = start + 1; i < array.length; i++) {

if (array[i] > array[i - 1]) {

curLongest++;

} else {

break;

}

}

longest = curLongest > longest ? curLongest : longest;

}

return longest;

}

// O(n) solution, for each index of the original array, find the longest

// ascending subarray ending with that index.

public int longestII(int[] array) {

assert array != null;

if (array.length == 0) {

return 0;

}

// the global longest subarray length

int longest = 1;

// the current longest subarray length

int curLongest = 1;

for (int i = 1; i < array.length; i++) {

// if array[i] > array[i - 1], we can let curLongest++

// else the current ascending subarray is ended and

// we need to start with a new one.

if (array[i] > array[i - 1]) {

curLongest++;

longest = curLongest > longest ? curLongest : longest;

} else {

curLongest = 1;

}

}

return longest;

}

public static void main(String[] args) {

LongestAscendingSubarray solution = new LongestAscendingSubarray();

int[] array = new int[0];

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

array = new int[] { 1 };

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

array = new int[] { 3, 3, 3 };

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

array = new int[] { 3, 2, 1 };

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

array = new int[] { 1, 2, 3, 4 };

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

array = new int[] { 7, 2, 3, 1, 5, 8, 9 };

System.out.println(solution.longestI(array));

System.out.println(solution.longestII(array));

}

}

#### 3. Max Product Of Cutting Rope

/\*\*

\* Given a rope with integer-length n,

\* how to cut the rope into m integer-length parts with length p[0], p[1],...,p[m-1],

\* in order to get the maximal product of p[0]\*p[1]\* ... \*p[m-1]?

\* m is determined by you and must be greater than 0 (at least one cut must be made).

\*/

public class CuttingRope {

public int maxCut(int n) {

assert n > 1;

int[] maxCut = new int[n + 1];

maxCut[1] = 1;

for (int i = 2; i <= n; i++) {

for (int j = 1; j <= i / 2; j++) {

// since we need at least one cut, the smaller part of the cut

// is in range of [1, i / 2], for each of the values in this range,

// we need to know what is the maximal product.

// Two options, either we do not cut the i - j, or we cut the i - j,

// So we have to pick the maximal value of maxCut[i - j] and i - j.

maxCut[i] = Math.max(maxCut[i], j \* Math.max(maxCut[i - j], i - j));

}

}

return maxCut[n];

}

public static void main(String[] args) {

CuttingRope solution = new CuttingRope();

int n = 2;

System.out.println(solution.maxCut(n));

n = 3;

System.out.println(solution.maxCut(n));

n = 4;

System.out.println(solution.maxCut(n));

n = 11;

System.out.println(solution.maxCut(n));

}

}

#### 4. Jump Game

/\*\*

\* Given an array of non-negative integers,

\* you are initially positioned at the first index of the array.

\* Each element in the array represents your maximum jump length at that position.

\* Determine if you are able to reach the last index.

\*/

public class JumpGame {

public boolean canJumpI(int[] array) {

assert array != null && array.length != 0;

boolean[] canJump = new boolean[array.length];

canJump[0] = true;

for (int i = 1; i < array.length; i++) {

for (int j = 0; j < i; j++) {

if (canJump[j] && array[j] + j >= i) {

canJump[i] = true;

break;

}

}

}

return canJump[array.length - 1];

}

public boolean canJumpII(int[] array) {

assert array != null && array.length != 0;

if (array.length == 1) {

return true;

}

boolean[] canJump = new boolean[array.length];

for (int i = array.length - 2; i >= 0; i--) {

if (i + array[i] >= array.length - 1) {

canJump[i] = true;

} else {

for (int j = array[i]; j >= 1; j--) {

if (canJump[j + i]) {

canJump[i] = true;

break;

}

}

}

}

return canJump[0];

}

public static void main(String[] args) {

JumpGame solution = new JumpGame();

int[] array = new int[] { 2, 3, 1, 1, 4 };

System.out.println(solution.canJumpI(array));

System.out.println(solution.canJumpII(array));

array = new int[] { 3, 2, 1, 0, 4 };

System.out.println(solution.canJumpI(array));

System.out.println(solution.canJumpII(array));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 12 - String

#### 1. Remove Adjacent And Duplicated Characters(Leave Only One)

/\*\*

\* Remove duplicated and adjacent letters,

\* (leave only one letter in each duplicated section) in a string.

\*/

public class RemoveDuplicate {

public String deDup(String input) {

assert input != null;

if (input.length() <= 1) {

return input;

}

char[] array = input.toCharArray();

int end = 0;

for (int i = 1; i < array.length; i++) {

// if any duplicate characters, we only take the

// first one, when array[i] == array[end], we just skip.

if (array[i] != array[end]) {

array[++end] = array[i];

}

}

return new String(array, 0, end + 1);

}

public static void main(String[] args) {

RemoveDuplicate solution = new RemoveDuplicate();

String input = "";

System.out.println(solution.deDup(input));

input = "abc";

System.out.println(solution.deDup(input));

input = "aaabbc";

System.out.println(solution.deDup(input));

input = "aaaa";

System.out.println(solution.deDup(input));

}

}

#### 2. Replace Space Character With “20%”

/\*\*

\* Write a method to replace all spaces in a string with '%20'.

\*/

public class ReplaceSpace {

public String replaceSpace(String input) {

assert input != null;

if (input.length() == 0) {

return input;

}

int spaceCount = 0;

for (int i = 0; i < input.length(); i++) {

if (input.charAt(i) == ' ') {

spaceCount++;

}

}

int newLength = input.length() + 2 \* spaceCount;

char[] result = new char[newLength];

int current = newLength - 1;

for (int i = input.length() - 1; i >= 0; i--) {

if (input.charAt(i) == ' ') {

result[current--] = '0';

result[current--] = '2';

result[current--] = '%';

} else {

result[current--] = input.charAt(i);

}

}

return new String(result);

}

public static void main(String[] args) {

ReplaceSpace solution = new ReplaceSpace();

String input = "";

System.out.println(solution.replaceSpace(input));

input = " ";

System.out.println(solution.replaceSpace(input));

input = "yahoo.com/q?flo wer market";

System.out.println(solution.replaceSpace(input));

}

}

#### 3. Reverse By Words

/\*\*

\* Reverse the words in a sentence.

\*

\* Example:

\* "I love Yahoo" --> "Yahoo love I"

\*

\* Assumption:

\* 1). The words are separated by one space character.

\* 2). There are no leading or trailing spaces.

\*/

public class ReverseWords {

public String reverseWords(String input) {

assert input != null;

char[] array = input.toCharArray();

// reverse the whole char array first

reverse(array, 0, array.length - 1);

int start = 0;

// reverse each of the words

for (int i = 0; i < array.length; i++) {

// the start index of a word

if (array[i] != ' ' && i == 0 || array[i - 1] == ' ') {

start = i;

}

// the end index of a word

if (array[i] != ' ' && i == array.length - 1 || array[i + 1] == ' ') {

reverse(array, start, i);

}

}

return new String(array);

}

private void reverse(char[] array, int left, int right) {

while (left < right) {

char temp = array[left];

array[left] = array[right];

array[right] = temp;

left++;

right--;

}

}

public static void main(String[] args) {

ReverseWords solution = new ReverseWords();

String input = "";

System.out.println(solution.reverseWords(input));

input = "I love Yahoo";

System.out.println(solution.reverseWords(input));

input = "yahoo";

System.out.println(solution.reverseWords(input));

}

}

#### 4. Determine Whether A String Is Substring Of Another

/\*\*

\* how to determine whether a string is a substring of another string.

\*/

public class Strstr {

// return the first match index of the large String,

// if not found, return -1

public int strStr(String large, String small) {

assert large != null && small != null;

assert !large.isEmpty() && !small.isEmpty();

int largeLen = large.length();

int smallLen = small.length();

if (largeLen < smallLen) {

return -1;

}

char[] largeArray = large.toCharArray();

char[] smallArray = small.toCharArray();

for (int i = 0; i <= largeLen - smallLen; i++) {

if (match(largeArray, i, smallArray)) {

return i;

}

}

return -1;

}

// check if the subarray starting from index start of the largeArray

// can match the smallArray.

private boolean match(char[] largeArray, int start, char[] smallArray) {

for (int i = 0; i < smallArray.length; i++) {

if (largeArray[start + i] != smallArray[i]) {

return false;

}

}

return true;

}

public int robinKarp(String large, String small) {

assert large != null && small != null;

assert !large.isEmpty() && !small.isEmpty();

int largeLen = large.length();

int smallLen = small.length();

if (largeLen < smallLen) {

return -1;

}

char[] largeArray = large.toCharArray();

char[] smallArray = small.toCharArray();

long seed = 1;

long targetHash = small.charAt(0);

for (int i = 1; i < smallLen; i++) {

targetHash = targetHash \* 31 + small.charAt(i);

seed \*= 31;

}

long hash = 0;

for (int i = 0; i < smallLen; i++) {

hash = hash \* 31 + large.charAt(i);

}

if (hash == targetHash && match(largeArray, 0, smallArray)) {

return 0;

}

for (int i = smallLen; i < largeLen; i++) {

hash -= seed \* large.charAt(i - smallLen);

hash = hash \* 31 + large.charAt(i);

if (hash == targetHash && match(largeArray, i - smallLen + 1, smallArray)) {

return i - smallLen + 1;

}

}

return -1;

}

public static void main(String[] args) {

Strstr solution = new Strstr();

String large = "abcd";

String small = "cd";

System.out.println(solution.strStr(large, small));

System.out.println(solution.robinKarp(large, small));

large = "abcdefghijklmnopqrstuvwxyz";

small = "cdefghijklmnopqrstuvwxy";

System.out.println(solution.strStr(large, small));

System.out.println(solution.robinKarp(large, small));

}

}

#### 5. Repeatedly Deduplicate Adjacent Repeated Characters

/\*\*

\* Repeatedly de-duplicate adjacent repeated letters.

\* Example:

\* abbbaz --> aaz --> z

\*/

public class RepeatedDedup {

public String repeatedDedup(String input) {

assert input != null;

Deque<Character> stack = new LinkedList<Character>();

for (int i = 0; i < input.length(); i++) {

if (stack.isEmpty() || input.charAt(i) != stack.peekFirst()) {

stack.offerFirst(input.charAt(i));

} else {

while (i + 1 < input.length() && input.charAt(i + 1) == input.charAt(i)) {

i++;

}

stack.pollFirst();

}

}

StringBuilder builder = new StringBuilder();

while (!stack.isEmpty()) {

builder.append(stack.pollLast());

}

return builder.toString();

}

public String repeatedDedupII(String input) {

assert input != null;

char[] array = input.toCharArray();

int end = -1;

for (int i = 0; i < array.length; i++) {

if (end == -1 || array[i] != array[end]) {

array[++end] = array[i];

} else {

while (i + 1 < array.length && array[i + 1] == array[i]) {

i++;

}

end--;

}

}

return new String(array, 0, end + 1);

}

public static void main(String[] args) {

RepeatedDedup solution = new RepeatedDedup();

String input = "";

System.out.println(solution.repeatedDedup(input));

System.out.println(solution.repeatedDedupII(input));

input = "abc";

System.out.println(solution.repeatedDedup(input));

System.out.println(solution.repeatedDedupII(input));

input = "abbccbaaazx";

System.out.println(solution.repeatedDedup(input));

System.out.println(solution.repeatedDedupII(input));

input = "abbacddada";

System.out.println(solution.repeatedDedup(input));

System.out.println(solution.repeatedDedupII(input));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 14 - Dynamic Programming II

#### 1. Minimum Number Of Jumps

/\*\*

\* Given the same setup as the Jump problem,

\* return the minimum number of jumps needed to reach the end.

\* If the end of array can not be reached, return the length of the array.

\*/

public class MinJump {

public int minJumpI(int[] array) {

assert array != null && array.length != 0;

int length = array.length;

// minJump record the min number of jumps from 0 to each of the indices.

int[] minJump = new int[length];

// we do not need to jump for index 0.

minJump[0] = 0;

for (int i = 1; i < length; i++) {

minJump[i] = length;

for (int j = i - 1; j >= 0; j--) {

if (j + array[j] >= i) {

// minJump[i] = min(minJump[j] + 1) for all the j < i

// and can jump from j to i.

minJump[i] = Math.min(minJump[i], minJump[j] + 1);

}

}

}

return minJump[length - 1];

}

public int minJumpII(int[] array) {

assert array != null && array.length != 0;

// from left index, we need current number of jumps

// the left - 1 index is the largest one we can reach

// using current number of jumps - 1 jumps

int left = 0;

// the right index is the largest one we can reach

// using current number of jumps

int right = 0;

// current number of jumps

int jump = 0;

// if left > right, it means at some point we can never proceed.

while (left <= right) {

// if using current number of jumps can reach the end of array

// we can return here.

if (right >= array.length - 1) {

return jump;

}

int nextRight = right;

// get the largest index we can reach using

// current number of jumps + 1 jumps,

// it should be the largest index we can reach with

// jumping one step further

for (int i = left; i <= right; i++) {

nextRight = Math.max(nextRight, i + array[i]);

}

// update left/right for the next jump

left = right + 1;

right = nextRight;

jump++;

}

return array.length;

}

public static void main(String[] args) {

MinJump solution = new MinJump();

int[] array = new int[] { 2, 3, 1, 1, 4 };

System.out.println(solution.minJumpI(array));

System.out.println(solution.minJumpII(array));

array = new int[] { 3, 2, 1, 0, 4 };

System.out.println(solution.minJumpI(array));

System.out.println(solution.minJumpII(array));

}

}

#### 2. Dictionary Word

/\*\*

\* Given a word, determine if it can be composed by

\* concatenating words from a given dictionary.

\* Assumption:

\* There is no empty string or null in the dictionary.

\*/

public class DictionaryWord {

public boolean canBreak(String input, Set<String> dictionary) {

assert input != null && input.length() != 0;

int length = input.length();

boolean[] canBreak = new boolean[length];

for (int i = 0; i < length; i++) {

canBreak[i] = dictionary.contains(input.substring(0, i + 1));

if (canBreak[i]) {

continue;

}

for (int j = 0; j < i; j++) {

if (canBreak[j] && dictionary.contains(input.substring(j + 1, i + 1))) {

canBreak[i] = true;

break;

}

}

}

return canBreak[length - 1];

}

public static void main(String[] args) {

DictionaryWord solution = new DictionaryWord();

Set<String> dictionary = new HashSet<String>();

dictionary.add("bob");

dictionary.add("ro");

dictionary.add("cat");

String input = "bcoact";

System.out.println(solution.canBreak(input, dictionary));

input = "bobcatrocat";

System.out.println(solution.canBreak(input, dictionary));

}

}

#### 3. Edit Distance

/\*\*

\* Given two strings of alphanumeric characters,

\* determine the minimum number of Replace, Delete, and Insert operations

\* needed to transform one string into the other.

\*/

public class EditDistance {

public int editDistance(String str1, String str2) {

assert str1 != null && str2 != null;

int len1 = str1.length();

int len2 = str2.length();

int[][] distance = new int[len1 + 1][len2 + 1];

for (int i = 0; i <= len1; i++) {

distance[i][0] = i;

}

for (int j = 0; j <= len2; j++) {

distance[0][j] = j;

}

for (int i = 1; i <= len1; i++) {

for (int j = 1; j <= len2; j++) {

if (str1.charAt(i - 1) == str2.charAt(j - 1)) {

distance[i][j] = distance[i - 1][j - 1];

} else {

distance[i][j] = Math.min(Math.min(distance[i - 1][j], distance[i][j - 1]),

distance[i - 1][j - 1]) + 1;

}

}

}

return distance[len1][len2];

}

public static void main(String[] args) {

EditDistance solution = new EditDistance();

String str1 = "";

String str2 = "";

System.out.println(solution.editDistance(str1, str2));

str1 = "";

str2 = "abc";

System.out.println(solution.editDistance(str1, str2));

str1 = "bdc";

str2 = "abc";

System.out.println(solution.editDistance(str1, str2));

str1 = "acde";

str2 = "bbb";

System.out.println(solution.editDistance(str1, str2));

}

}

#### 4. Largest Square Of 1’s In Binary Matrix

/\*\*

\* Determine the edge length of the largest square of 1’s

\* in a given binary matrix.

\*/

public class LargestOneSquare {

// Assumption: the matrix is n \* n, and n > 0.

public int largest(int[][] matrix) {

assert matrix != null;

int len = matrix.length;

// to record the largest square with the bottom right corner's

// coordinate as i, j for all i, j in [0, len - 1].

int[][] largest = new int[len][len];

// to get the largest square in one pass, we need to record and

// update the global largest square, it is initialized to 0 at

// first place.

int result = 0;

for (int row = 0; row < len; row++) {

for (int col = 0; col < len; col++) {

if (matrix[row][col] == 0) {

largest[row][col] = 0;

} else if (row == 0 || col == 0) {

// for all the border cell with value 1, the largest square is 1.

largest[row][col] = 1;

} else {

largest[row][col] = Math.min(largest[row - 1][col - 1],

Math.min(largest[row - 1][col], largest[row][col - 1])) + 1;

}

result = Math.max(result, largest[row][col]);

}

}

return result;

}

public static void main(String[] args) {

LargestOneSquare solution = new LargestOneSquare();

int[][] matrix = new int[][] { { 1, 1, 1, 1 }, { 1, 1, 1, 1 }, { 1, 1, 1, 0 }, { 1, 1, 0, 0 }, };

System.out.println(solution.largest(matrix));

}

}

#### 5. Merge Stones

/\*\*

\* There are n piles of stones in a row，

\* The piles have a1, a2, ..., an stones respectively.

\* You can merge two adjacent piles of stones,

\* paying a cost which is equal to the number of stones in the resulting pile.

\* Find the minimum cost of merging all the stones to a single pile.

\*/

public class MergeStones {

public int minCost(int[] stones) {

assert stones != null;

if (stones.length == 0) {

return 0;

}

int length = stones.length;

// minCost to record the min cost of merging the stones at subarray [i, j]

// subSum to record the sum of subarray [i, j]

int[][] minCost = new int[length][length];

int[][] subSum = new int[length][length];

for (int end = 0; end < length; end++) {

for (int start = end; start >= 0; start--) {

if (start == end) {

// if start == end, we do not need to merge, the cost is 0.

subSum[start][end] = stones[start];

minCost[start][end] = 0;

} else {

// else, we need to find the min cost of the next cut at any indices

// between [start, end - 1],

// minCost[start][end] = min(minCost[start][mid] + minCost[mid +

// 1][end] + subSsum[start][end]),

// for any mid in [start, end - 1]

subSum[start][end] = subSum[start][end - 1] + stones[end];

minCost[start][end] = Integer.MAX\_VALUE;

for (int mid = end - 1; mid >= start; mid--) {

minCost[start][end] = Math.min(minCost[start][end], minCost[start][mid]

+ minCost[mid + 1][end] + subSum[start][end]);

}

}

}

}

return minCost[0][length - 1];

}

public static void main(String[] args) {

MergeStones solution = new MergeStones();

int[] stones = new int[0];

System.out.println(solution.minCost(stones));

stones = new int[] { 1 };

System.out.println(solution.minCost(stones));

stones = new int[] { 4, 3, 3, 4 };

System.out.println(solution.minCost(stones));

stones = new int[] { 13, 7, 8 };

System.out.println(solution.minCost(stones));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 16 - Dynamic Programming III

#### 1. Largest Sum Of A Subarray

/\*\*

\* Given an unsorted array, return the sum of the subarray

\* having the largest sum.

\*/

public class LargestSumSubArray {

public int largest(int[] array) {

assert array != null && array.length != 0;

int largest = array[0];

int curLargest = largest;

for (int i = 1; i < array.length; i++) {

curLargest = Math.max(curLargest + array[i], array[i]);

largest = Math.max(largest, curLargest);

}

return largest;

}

public static void main(String[] args) {

LargestSumSubArray solution = new LargestSumSubArray();

int[] array = new int[] { 1 };

System.out.println(solution.largest(array));

array = new int[] { 1, 2, 3 };

System.out.println(solution.largest(array));

array = new int[] { -3, -2, -1 };

System.out.println(solution.largest(array));

array = new int[] { 2, -1, 4, 1, -2, -1 };

System.out.println(solution.largest(array));

}

}

#### 2. Longest Consecutive 1’s

/\*\*

\* Find the longest consecutive "1"s in an unsorted integer array.

\*/

public class LongestConsecutiveOnes {

public int longest(int[] array) {

assert array != null;

int longest = 0;

int curLongest = 0;

for (int i = 0; i < array.length; i++) {

if (array[i] == 1) {

curLongest++;

longest = Math.max(longest, curLongest);

} else {

curLongest = 0; // reset curLongest to 0 if array[i] == 0

}

}

return longest;

}

public static void main(String[] args) {

LongestConsecutiveOnes solution = new LongestConsecutiveOnes();

int[] array = new int[0];

System.out.println(solution.longest(array));

array = new int[] { 1, 1, 1, 1 };

System.out.println(solution.longest(array));

array = new int[] { 0, 0, 0, 0 };

System.out.println(solution.longest(array));

array = new int[] { 0, 1, 1, 1, 0, 0, 1, 0 };

System.out.println(solution.longest(array));

}

}

#### 3. Largest Cross Of 1’s

/\*\*

\* Given a Matrix that contains only 1s and 0s,

\* find the largest cross which contains only 1s,

\* return the length of the arm.

\*/

public class LargestCross {

/\*\*

\* Assumption:

\* 1. matrix is not empty, row > 0 & col > 0

\* 2. all the matrix's rows has same length

\*/

public int largest(int[][] matrix) {

assert matrix != null;

int rows = matrix.length;

int cols = matrix[0].length;

return merge(longestUpLeft(matrix, rows, cols), longestDownRight(matrix, rows, cols));

}

// instead of get the longest from up/left direction separately,

// we can do it in one traverse.

private int[][] longestUpLeft(int[][] matrix, int rows, int cols) {

int[][] longestUp = new int[rows][cols];

int[][] longestLeft = new int[rows][cols];

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

if (matrix[i][j] == 1) {

longestUp[i][j] = getNumber(i - 1, j, longestUp) + 1;

longestLeft[i][j] = getNumber(i, j - 1, longestLeft) + 1;

}

}

}

merge(longestUp, longestLeft);

return longestUp;

}

// instead of getting the longest from down/right direction separately,

// we can do it in one traverse.

private int[][] longestDownRight(int[][] matrix, int rows, int cols) {

int[][] longestDown = new int[rows][cols];

int[][] longestRight = new int[rows][cols];

for (int i = rows - 1; i >= 0; i--) {

for (int j = cols - 1; j >= 0; j--) {

if (matrix[i][j] == 1) {

longestDown[i][j] = getNumber(i + 1, j, longestDown) + 1;

longestRight[i][j] = getNumber(i, j + 1, longestRight) + 1;

}

}

}

merge(longestDown, longestRight);

return longestDown;

}

private int getNumber(int row, int col, int[][] longest) {

if (row < 0 || row >= longest.length) {

return 0;

} else if (col < 0 || col >= longest[0].length) {

return 0;

}

return longest[row][col];

}

// update matrix1's elements to the min value of the two values

// in the same position of matrix1 and matrix2

// and return the max value of matrix1 after merging.

private int merge(int[][] matrix1, int[][] matrix2) {

int max = 0;

for (int row = 0; row < matrix1.length; row++) {

for (int col = 0; col < matrix1[0].length; col++) {

matrix1[row][col] = Math.min(matrix1[row][col], matrix2[row][col]);

max = Math.max(max, matrix1[row][col]);

}

}

return max;

}

public static void main(String[] args) {

LargestCross solution = new LargestCross();

int[][] matrix = new int[][] {

{ 1, 0, 1, 0, 1 },

{ 0, 1, 1, 1, 1 },

{ 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1 },

{ 1, 0, 1, 1, 1 } };

System.out.println(solution.largest(matrix));

}

}

#### 4. Biggest Rectangle In Histogram

/\*\*

\* Get the area of the largest rectangle in histogram.

\* Assumption:

\* 1). the input array is not null.

\*/

public class LargestRectangleHistogram {

public int largest(int[] array) {

assert array != null;

int result = 0;

Deque<Integer> stack = new LinkedList<Integer>();

for (int i = 0; i <= array.length; i++) {

int cur = i == array.length ? 0 : array[i];

while (!stack.isEmpty() && array[stack.peekFirst()] >= cur) {

int height = array[stack.pollFirst()];

int left = stack.isEmpty() ? 0 : stack.peekFirst() + 1;

result = Math.max(result, height \* (i - left));

}

stack.offerFirst(i);

}

return result;

}

public static void main(String[] args) {

LargestRectangleHistogram solution = new LargestRectangleHistogram();

int[] array = new int[0];

System.out.println(solution.largest(array));

array = new int[] { 2 };

System.out.println(solution.largest(array));

array = new int[] { 2, 1, 3, 4, 1 };

System.out.println(solution.largest(array));

array = new int[] { 5, 4, 2, 3, 4 };

System.out.println(solution.largest(array));

}

}

#### 5. Largest Rectangle Of 1’s

/\*\*

\* Get the area of the largest rectangle of 1's in a binary matrix.

\* A binary matrix contains only 0 and 1.

\* Assumptions:

\* 1). the matrix is M \* N, M > 0 and N > 0

\*/

public class LargestRectangleOfOnes {

public int largest(int[][] matrix) {

int rows = matrix.length;

int cols = matrix[0].length;

int[] cum = new int[cols];

int result = 0;

for (int i = 0; i < rows; i++) {

cummulate(cum, matrix[i]);

result = Math.max(result, maxRectangle(cum));

}

return result;

}

private void cummulate(int[] cum, int[] current) {

for (int i = 0; i < cum.length; i++) {

cum[i] = current[i] == 0 ? 0 : cum[i] + 1;

}

}

private int maxRectangle(int[] array) {

int result = 0;

Deque<Integer> stack = new LinkedList<Integer>();

for (int i = 0; i <= array.length; i++) {

int cur = i == array.length ? 0 : array[i];

while (!stack.isEmpty() && array[stack.peekFirst()] >= cur) {

int height = array[stack.pollFirst()];

int left = stack.isEmpty() ? 0 : stack.peekFirst() + 1;

result = Math.max(result, height \* (i - left));

}

stack.offerFirst(i);

}

return result;

}

public static void main(String[] args) {

LargestRectangleOfOnes solution = new LargestRectangleOfOnes();

int[][] matrix = new int[][] { { 1, 1, 0, 1 }, { 1, 1, 1, 0 }, { 1, 1, 0, 1 }, { 1, 1, 1, 0 } };

System.out.println(solution.largest(matrix));

}

}

#### 6. Largest Sum Of Submatrix

/\*\*

\* Given a Matrix of integers (positive & negative numbers & 0s),

\* find the submatrix with the largest sum, return the sum.

\*/

public class LargestSubMatrix {

/\*

\* Assumption:

\* 1. matrix is not empty, row > 0 & col > 0

\* 2. all the matrix's rows has same length

\*/

public int largest(int[][] matrix) {

assert matrix != null;

int largest = Integer.MIN\_VALUE;

for (int i = 0; i < matrix.length; i++) {

int[] array = new int[matrix[0].length];

for (int j = i; j < matrix.length; j++) {

addArray(array, matrix[j]);

int curLargest = largest(array);

largest = Math.max(largest, curLargest);

}

}

return largest;

}

private void addArray(int[] array, int[] addition) {

for (int i = 0; i < array.length; i++) {

array[i] += addition[i];

}

}

private int largest(int[] array) {

int largest = Integer.MIN\_VALUE;

int curLargest = 0;

for (int i = 0; i < array.length; i++) {

curLargest = Math.max(curLargest + array[i], array[i]);

largest = Math.max(largest, curLargest);

}

return largest;

}

public static void main(String[] args) {

LargestSubMatrix solution = new LargestSubMatrix();

int[][] matrix = new int[][] {

{ 1, 2, -1, -4, -20 },

{ -8, -3, 4, 2, 1 },

{ 3, 8, 10, 1, 3 },

{ -4, -1, 1, 7, -6 } };

System.out.println(solution.largest(matrix));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 18 - Probability, Sampling And Randomization

#### 1. Shuffle

/\*\*

\* Shuffle of an array.

\*/

public class Shuffle {

public void shuffle(int[] array) {

assert array != null;

for (int i = 0; i < array.length; i++) {

swap(array, i, i + (int) (Math.random() \* (array.length - i)));

}

}

private void swap(int[] array, int index1, int index2) {

int temp = array[index1];

array[index1] = array[index2];

array[index2] = temp;

}

public static void main(String[] args) {

Shuffle solution = new Shuffle();

int[] array = new int[] { 1 };

for (int i = 0; i < 10; i++) {

solution.shuffle(array);

System.out.println(Arrays.toString(array));

}

array = new int[] { 1, 2, 3, 4, 5 };

for (int i = 0; i < 100; i++) {

solution.shuffle(array);

System.out.println(Arrays.toString(array));

}

}

}

#### 2. Reservoir Sampling

/\*\*

\* Do sampling for an unlimited data flow and when reading the nth

\* element we are required to return one random number among all the

\* numbers read so far, such that the probability of each of the n elements is 1/n.

\*/

public class ReservoirSampling {

public static final int NOT\_START = Integer.MIN\_VALUE;

private int counter;

private int random;

public ReservoirSampling() {

counter = 0;

random = NOT\_START;

}

public void read(int value) {

counter++;

if ((int) (Math.random() \* counter) == 0) {

random = value;

}

}

public int random() {

return random;

}

public static void main(String[] args) {

ReservoirSampling solution = new ReservoirSampling();

for (int i = 0; i < 100; i++) {

solution.read(i);

System.out.println(solution.random());

}

}

}

#### 3. Random7 Using Random5

/\*\*

\* Use random7() to generate random5().

\*/

public class RandomFive {

public int random7() {

while (true) {

int current = random5() \* 5 + random5();

if (current < 21) {

return current % 7;

}

}

}

public int random1000() {

while (true) {

int current = 0;

for (int i = 0; i < 5; i++) {

current = current \* 5 + random5();

}

System.out.println(current);

if (current < 3000) {

return current % 1000;

}

}

}

public int random5() {

return (int) (Math.random() \* 5);

}

public static void main(String[] args) {

RandomFive solution = new RandomFive();

System.out.println(solution.random7());

System.out.println(solution.random1000());

}

}

#### 4. Median Tracker

/\*\*

\* Track median of an unlimited data flow.

\*/

public class MedianTracker {

PriorityQueue<Integer> minHeap;

PriorityQueue<Integer> maxHeap;

public MedianTracker() {

minHeap = new PriorityQueue<Integer>(16);

maxHeap = new PriorityQueue<Integer>(16, Collections.reverseOrder());

}

public void read(int value) {

// if maxHeap is empty or the value can be put into maxHeap, put the value

// into maxHeap.

// else put the value into minHeap.

if (maxHeap.isEmpty() || value <= maxHeap.peek()) {

maxHeap.offer(value);

} else {

minHeap.offer(value);

}

// we maintain the invariant that either

// the size of maxHeap == the size of minHeap,

// or the size of maxHeap == the size of minHeap + 1.

if (minHeap.size() > maxHeap.size()) {

maxHeap.offer(minHeap.poll());

} else if (maxHeap.size() > minHeap.size() + 1) {

minHeap.offer(maxHeap.poll());

}

}

public double median() {

int size = size();

if (size == 0) {

throw new NoSuchElementException("No such element!");

} else if (size % 2 != 0) {

// we know the median is in maxHeap.

return maxHeap.peek();

} else {

return (maxHeap.peek() + minHeap.peek()) / 2.0;

}

}

public int size() {

return maxHeap.size() + minHeap.size();

}

public static void main(String[] args) {

MedianTracker solution = new MedianTracker();

for (int i = 0; i < 10; i++) {

solution.read(i);

System.out.println(solution.median());

}

for (int i = 9; i >= 0; i--) {

solution.read(i);

System.out.println(solution.median());

}

}

}

#### 5. 95 Percentile Of Urls’ Lengths

/\*\*

\* Given 200 urls, find the 95-th percentile of all urls' length.

\*/

public class Percentile {

// To simplify the problem, we only take the lengths instead of the

// the real urls.

// Assumption: the length of any urls is <= 4010.

public int percentile95(int[] lengths) {

int[] count = new int[4011];

for (int length : lengths) {

count[length]++;

}

int sum = 0;

int result = 4011;

while (sum <= 200 \* 0.05) {

sum += count[--result];

}

return result;

}

public static void main(String[] args) {

Percentile solution = new Percentile();

int[] lengths = new int[200];

for (int i = 0; i < 200; i++) {

lengths[i] = 3000 + i / 10;

}

System.out.println(solution.percentile95(lengths));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 19 - 强化练习 1

#### 1. Remove Duplicates From Sorted Array

/\*\*

\* Remove duplicate elements from sorted array.

\* 1). for the elements with same value, retain only one

\* {1, 1, 1, 2, 2, 3} -> {1, 2, 3}

\* 2). for the elements with same value, retain at most two

\* {1, 1, 1, 2, 2, 3} -> {1, 1, 2, 2, 3}

\* 3). for the elements with same value, do not retain any

\* {1, 1, 1, 2, 2, 3} -> {3}

\*/

public class RemoveDuplicates {

// retain only one element among the group with same value

// {1, 1, 1, 2, 2, 3} -> {1, 2, 3}

// return the new length

public int removeI(int[] array) {

assert array != null;

if (array.length == 0) {

return 0;

}

int end = 1;

for (int i = 1; i < array.length; i++) {

if (array[i] != array[end - 1]) {

array[end++] = array[i];

}

}

return end;

}

// retain at most two elements among the group with same value

// {1, 1, 1, 2, 2, 3} -> {1, 1, 2, 2, 3}

// return the new length

public int removeII(int[] array) {

assert array != null;

if (array.length < 2) {

return array.length;

}

int end = 2;

for (int i = 2; i < array.length; i++) {

if (array[i] != array[end - 2]) {

array[end++] = array[i];

}

}

return end;

}

// remove all elements among the group with same value

// {1, 1, 1, 2, 2, 3} -> {3}

// return the new length

public int removeIII(int[] array) {

assert array != null;

if (array.length < 2) {

return array.length;

}

int end = 0;

boolean flag = false;

for (int i = 1; i < array.length; i++) {

if (array[i] == array[end]) {

flag = true;

} else if (flag) {

// replace array[end] with current candidate value of array[i],

// whether it should be retained will be determined later.

array[end] = array[i];

flag = false;

} else {

// if flag is not turned on, means array[end] has no duplicate

// values and it should be retained.

array[++end] = array[i];

}

}

// if flag is turned on, means the element of array[end]

// should be deleted.

return flag ? end : end + 1;

}

public static void main(String[] args) {

RemoveDuplicates solution = new RemoveDuplicates();

int[] array1 = new int[0];

int[] array2 = new int[0];

int[] array3 = new int[0];

int length1 = solution.removeI(array1);

System.out.println(Arrays.toString(Arrays.copyOf(array1, length1)));

int length2 = solution.removeII(array2);

System.out.println(Arrays.toString(Arrays.copyOf(array2, length2)));

int length3 = solution.removeIII(array3);

System.out.println(Arrays.toString(Arrays.copyOf(array3, length3)));

array1 = new int[] { 1, 1, 1, 2, 2, 3 };

array2 = new int[] { 1, 1, 1, 2, 2, 3 };

array3 = new int[] { 1, 1, 1, 2, 2, 3 };

length1 = solution.removeI(array1);

System.out.println(Arrays.toString(Arrays.copyOf(array1, length1)));

length2 = solution.removeII(array2);

System.out.println(Arrays.toString(Arrays.copyOf(array2, length2)));

length3 = solution.removeIII(array3);

System.out.println(Arrays.toString(Arrays.copyOf(array3, length3)));

array1 = new int[] { 0, 1, 1, 2, 3, 3, 3 };

array2 = new int[] { 0, 1, 1, 2, 3, 3, 3 };

array3 = new int[] { 0, 1, 1, 2, 3, 3, 3 };

length1 = solution.removeI(array1);

System.out.println(Arrays.toString(Arrays.copyOf(array1, length1)));

length2 = solution.removeII(array2);

System.out.println(Arrays.toString(Arrays.copyOf(array2, length2)));

length3 = solution.removeIII(array3);

System.out.println(Arrays.toString(Arrays.copyOf(array3, length3)));

}

}

#### 2. Least Comparisons

/\*\*

\* Least number of comparisons for getting

\* 1). the largest and smallest number.

\* 2). the largest and the second largest number.

\*/

public class LeastNumberComparisons {

// the Pair class is a wrapper of two integers.

static class Pair {

int first;

int second;

Pair(int first, int second) {

this.first = first;

this.second = second;

}

}

public Pair largestAndSmallest(int[] array) {

if (array == null || array.length == 0) {

return null;

}

ArrayList<Integer> larger = new ArrayList<Integer>();

ArrayList<Integer> smaller = new ArrayList<Integer>();

for (int i = 0; i < array.length; i += 2) {

if (i == array.length - 1) {

// if we have odd number of elements in the array,

// we need to add the last element to both larger

// smaller groups.

smaller.add(array[i]);

larger.add(array[i]);

} else if (array[i] <= array[i + 1]) {

smaller.add(array[i]);

larger.add(array[i + 1]);

} else {

smaller.add(array[i + 1]);

larger.add(array[i]);

}

}

// use Pair to store the largest number and smallest number,

return new Pair(findLargest(larger), findSmallest(smaller));

}

public Pair largestAndSecond(int[] array) {

if (array == null || array.length < 2) {

return null;

}

// key: the index in original array.

// value: the list of values smaller than array[key]

HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();

// binary reduction, use the list of all <index, array[index]> pairs conducted

// by the original array as the initial list.

ArrayList<Pair> list = new ArrayList<Pair>();

for (int i = 0; i < array.length; i++) {

list.add(new Pair(i, array[i]));

}

while (list.size() > 1) {

ArrayList<Pair> nextRound = new ArrayList<Pair>();

for (int i = 0; i < list.size(); i += 2) {

if (i + 1 < list.size()) {

Pair p1 = list.get(i);

Pair p2 = list.get(i + 1);

compare(p1, p2, nextRound, map);

} else {

nextRound.add(list.get(i));

}

}

list = nextRound;

}

Pair largest = list.get(0);

return new Pair(largest.second, findLargest(map.get(largest.first)));

}

private void compare(Pair p1, Pair p2, ArrayList<Pair> nextRound,

HashMap<Integer, ArrayList<Integer>> map) {

if (p1.second <= p2.second) {

nextRound.add(p2);

if (!map.containsKey(p2.first)) {

map.put(p2.first, new ArrayList<Integer>());

}

map.get(p2.first).add(p1.second);

} else {

nextRound.add(p1);

if (!map.containsKey(p1.first)) {

map.put(p1.first, new ArrayList<Integer>());

}

map.get(p1.first).add(p2.second);

}

}

private int findLargest(ArrayList<Integer> list) {

int max = list.get(0);

for (int i = 1; i < list.size(); i++) {

max = Math.max(max, list.get(i));

}

return max;

}

private int findSmallest(ArrayList<Integer> list) {

int min = list.get(0);

for (int i = 1; i < list.size(); i++) {

min = Math.min(min, list.get(i));

}

return min;

}

public static void main(String[] args) {

LeastNumberComparisons solution = new LeastNumberComparisons();

int[] array = new int[] { 1, 2, 3 };

Pair largestAndSmallest = solution.largestAndSmallest(array);

Pair largestAndSecond = solution.largestAndSecond(array);

System.out.println("largest: " + largestAndSmallest.first + ", smallest: "

+ largestAndSmallest.second);

System.out.println("largest: " + largestAndSecond.first + ", second largest: "

+ largestAndSecond.second);

array = new int[] { 5, 1, 2, 4, 3 };

largestAndSmallest = solution.largestAndSmallest(array);

largestAndSecond = solution.largestAndSecond(array);

System.out.println("largest: " + largestAndSmallest.first + ", smallest: "

+ largestAndSmallest.second);

System.out.println("largest: " + largestAndSecond.first + ", second largest: "

+ largestAndSecond.second);

array = new int[] { 1, 1, 2, 2, 3, 4, 5, 6 };

largestAndSmallest = solution.largestAndSmallest(array);

largestAndSecond = solution.largestAndSecond(array);

System.out.println("largest: " + largestAndSmallest.first + ", smallest: "

+ largestAndSmallest.second);

System.out.println("largest: " + largestAndSecond.first + ", second largest: "

+ largestAndSecond.second);

}

}

#### 3. Generate 2D Array In Spiral Order

/\*\*

\* Spiral order generate matrix.

\*/

public class SpiralGenerate {

// spiral order(clock-wise) generate matrix with specified rows and columns.

// the number is starting from 1.

public int[][] spiralGenerate(int rows, int cols) {

assert rows > 0 && cols > 0;

int[][] matrix = new int[rows][cols];

int top = 0;

int bottom = rows - 1;

int left = 0;

int right = cols - 1;

int current = 1;

while (top < bottom && left < right) {

for (int i = left; i <= right; i++) {

matrix[top][i] = current++;

}

for (int i = top + 1; i <= bottom - 1; i++) {

matrix[i][right] = current++;

}

for (int i = right; i >= left; i--) {

matrix[bottom][i] = current++;

}

for (int i = bottom - 1; i >= top + 1; i--) {

matrix[i][left] = current++;

}

top++;

bottom--;

left++;

right--;

}

if (top > bottom || left > right) {

return matrix;

}

if (top == bottom) {

for (int i = left; i <= right; i++) {

matrix[top][i] = current++;

}

} else {

for (int i = top; i <= bottom; i++) {

matrix[i][left] = current++;

}

}

return matrix;

}

public static void main(String[] args) {

SpiralGenerate solution = new SpiralGenerate();

print(solution.spiralGenerate(1, 1));

print(solution.spiralGenerate(1, 4));

print(solution.spiralGenerate(2, 5));

print(solution.spiralGenerate(4, 2));

print(solution.spiralGenerate(4, 4));

print(solution.spiralGenerate(3, 3));

}

public static void print(int[][] matrix) {

for (int i = 0; i < matrix.length; i++) {

System.out.println(Arrays.toString(matrix[i]));

}

}

}

#### 4. Rotate 2D Array Clockwise By 90 Degrees

/\*\*

\* Rotate N \* N matrix by 90 degree clock-wisely.

\*/

public class RotateMatrix {

public int[][] rotate(int[][] matrix) {

assert matrix != null;

assert matrix.length != 0 && matrix[0].length != 0;

int N = matrix.length;

for (int offset = 0; offset < N / 2; offset++) {

for (int i = offset; i < N - 1 - offset; i++) {

int temp = matrix[offset][i];

matrix[offset][i] = matrix[N - 1 - i][offset];

matrix[N - 1 - i][offset] = matrix[N - 1 - offset][N - 1 - i];

matrix[N - 1 - offset][N - 1 - i] = matrix[i][N - 1 - offset];

matrix[i][N - 1 - offset] = temp;

}

}

return matrix;

}

public static void main(String[] args) {

RotateMatrix solution = new RotateMatrix();

int[][] matrix = new int[][] { { 1 } };

print(solution.rotate(matrix));

matrix = new int[][] { { 1, 2 }, { 3, 4 } };

print(solution.rotate(matrix));

matrix = new int[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

print(solution.rotate(matrix));

matrix = new int[][] { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }, { 13, 14, 15, 16 } };

print(solution.rotate(matrix));

}

public static void print(int[][] matrix) {

for (int i = 0; i < matrix.length; i++) {

System.out.println(Arrays.toString(matrix[i]));

}

}

}

#### 5. Binary Tree Level Order Traversal In Zig-Zag Way

/\*\*

\* Print binary tree in zigzag order.

\*/

public class ZigZagBinaryTree {

public void zigZagPrint(TreeNode root) {

assert root != null;

Deque<TreeNode> deque = new LinkedList<TreeNode>();

deque.offerLast(root);

boolean leftToRight = true;

while (!deque.isEmpty()) {

int size = deque.size();

if (leftToRight) {

for (int i = 0; i < size; i++) {

TreeNode current = deque.pollFirst();

System.out.print(current.key);

if (current.left != null) {

deque.offerLast(current.left);

}

if (current.right != null) {

deque.offerLast(current.right);

}

}

} else {

for (int i = 0; i < size; i++) {

TreeNode current = deque.pollLast();

System.out.print(current.key);

if (current.right != null) {

deque.offerFirst(current.right);

}

if (current.left != null) {

deque.offerFirst(current.left);

}

}

}

leftToRight = !leftToRight;

System.out.println();

}

}

public static void main(String[] args) {

ZigZagBinaryTree solution = new ZigZagBinaryTree();

/\*

\* 1

\* 2 3

\* 4 5 6 7

\*/

TreeNode root = TreeNode.reConstruct(new String[] { "1", "2", "4", "#", "#", "5", "#", "#",

"3", "6", "#", "#", "7", "#", "#" });

solution.zigZagPrint(root);

}

}

#### 6. Lowest Common Ancestor

/\*\*

\* Lowest Common Ancestor of nodes.

\* Assumption: the provided 2 nodes are in the tree.

\* 1). without parent reference.

\* 2). with parent reference.

\* 3). lowest common ancestor of k nodes.

\*/

public class LowestCommonAncestor {

static class PTreeNode {

int value;

PTreeNode left;

PTreeNode right;

PTreeNode parent;

}

// 1. find lowest common ancestor of two nodes(without parent reference in the node)

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode node1, TreeNode node2) {

if (root == null) {

return null;

}

if (root == node1 || root == node2) {

return root;

}

TreeNode left = lowestCommonAncestor(root.left, node1, node2);

TreeNode right = lowestCommonAncestor(root.right, node1, node2);

if (left != null && right != null) {

return root;

}

return left == null ? right : left;

}

// 2. find lowes common ancestor of two nodes(with parent reference in the node)

public PTreeNode lowestCommonAncestor(PTreeNode node1, PTreeNode node2) {

assert node1 != null && node2 != null;

int len1 = length(node1);

int len2 = length(node2);

return mergeNode(node1, node2, len1 - len2);

}

private PTreeNode mergeNode(PTreeNode node1, PTreeNode node2, int lenDiff) {

if (lenDiff < 0) {

return mergeNode(node2, node1, -lenDiff);

}

while (lenDiff > 0) {

node1 = node1.parent;

lenDiff--;

}

while (node1 != node2) {

node1 = node1.parent;

node2 = node2.parent;

}

return node1;

}

private int length(PTreeNode node) {

int length = 0;

while (node != null) {

node = node.parent;

length++;

}

return length;

}

// 3. lowest common ancestor for k nodes, assuming all the k nodes are in the tree.

public TreeNode lowestCommonAncestor(TreeNode root, List<TreeNode> nodes) {

assert nodes.size() >= 2;

if (root == null) {

return null;

}

for (TreeNode node : nodes) {

if (root == node) {

return root;

}

}

TreeNode left = lowestCommonAncestor(root.left, nodes);

TreeNode right = lowestCommonAncestor(root.right, nodes);

if (left != null && right != null) {

return root;

}

return left == null ? right : left;

}

public static void main(String[] args) {

LowestCommonAncestor solution = new LowestCommonAncestor();

/\*

\* 1

\* 2 3

\* 4 5

\* 6

\*/

TreeNode root = TreeNode.reConstruct(new String[] { "1", "2", "4", "6", "#", "#", "#", "5",

"#", "#", "3", "#", "#" });

System.out.println(solution.lowestCommonAncestor(root, root.left.left.left, root.left.right));

System.out.println(solution.lowestCommonAncestor(root, root.left.left.left, root.right));

System.out.println(solution.lowestCommonAncestor(root, root.left.left.left, root.left));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 20 - Midterm 2

#### 1. Max Path Sum From One Leaf To Another

/\*\*

\* Given a binary tree in which each node element contains a number.

\* Find the maximum possible sum from one leaf node to another.

\* If there does not exist such a path, return Integer.MIN\_VALUE.

\*/

public class MaxPathSum {

public int maxSum(TreeNode root) {

assert root != null;

int[] max = new int[] { Integer.MIN\_VALUE };

maxSumHelper(root, max);

return max[0];

}

private int maxSumHelper(TreeNode root, int[] max) {

if (root == null) {

return 0;

}

int leftRes = maxSumHelper(root.left, max);

int rightRes = maxSumHelper(root.right, max);

int tmpSum = leftRes + rightRes + root.key;

if (root.left != null && root.right != null && tmpSum > max[0]) {

max[0] = tmpSum;

}

if (root.left == null) {

return root.key + rightRes;

} else if (root.right == null) {

return root.key + leftRes;

}

return Math.max(leftRes, rightRes) + root.key;

}

public static void main(String[] args) {

MaxPathSum solution = new MaxPathSum();

TreeNode root = TreeNode.reConstruct(new String[] { "-8", "4", "#", "#", "2", "1", "#", "#",

"1", "#", "#" });

System.out.println(solution.maxSum(root));

root = TreeNode.reConstruct(new String[] { "10", "#", "1", "#", "6", "#", "#" });

System.out.println(solution.maxSum(root));

root = TreeNode.reConstruct(new String[] { "2", "-11", "#", "#", "6", "#", "-2", "#", "#" });

System.out.println(solution.maxSum(root));

root = TreeNode.reConstruct(new String[] { "-15", "5", "-8", "#", "#", "1", "#", "#", "6", "3",

"#", "#", "9", "#", "0", "4", "#", "#", "-1", "10", "#", "#", "#" });

System.out.println(solution.maxSum(root));

}

}

#### 2. Move All 0’s To The Right End

/\*\*

\* Move all 0's to the right end of the original array.

\* 1. No need to maintain the relative order of the other elements.

\* 2. Need to maintain the relative order of the other elements.

\*/

public class MoveZero {

// no need to maintain the relative order of other elements.

// using partition strategy.

public void moveI(int[] array) {

if (array == null || array.length <= 1) {

return;

}

int left = 0;

int right = array.length - 1;

while (left <= right) {

if (array[left] == 0) {

swap(array, left, right--);

} else {

left++;

}

}

}

// need to maintain the relative order of other elements.

// move all non-zero elements to the left first, then fill all zero for the right end.

public void moveII(int[] array) {

if (array == null || array.length <= 1) {

return;

}

int nonZero = 0;

for (int i = 0; i < array.length; i++) {

if (array[i] != 0) {

array[nonZero++] = array[i];

}

}

for (int i = nonZero; i < array.length; i++) {

array[i] = 0;

}

}

private void swap(int[] array, int left, int right) {

int tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

public static void main(String[] args) {

MoveZero solution = new MoveZero();

int[] array = new int[] { 1, 2, 0, 3 };

solution.moveI(array);

System.out.println(Arrays.toString(array));

array = new int[] { 1, 2, 0, 3 };

solution.moveII(array);

System.out.println(Arrays.toString(array));

array = new int[] { 0, 0, 1, 0, 2, 0 };

solution.moveI(array);

System.out.println(Arrays.toString(array));

array = new int[] { 0, 0, 1, 0, 2, 0 };

solution.moveII(array);

System.out.println(Arrays.toString(array));

}

}

#### 3. Min Cuts Of Palindrome Partition

/\*\*

\* Given a string, a partitioning of the string is a palindrome partitioning

\* if every substring of the partition is a palindrome.

\* For example, “aba |b | bbabb |a| b| aba” is a palindrome partitioning of

\* “ababbbabbababa”.

\* Determine the fewest cuts needed for palindrome partitioning of a given string.

\* For example, minimum 3 cuts are needed for “ababbbabbababa”.

\* The three cuts are “a | babbbab | b | ababa”.

\* If a string is palindrome, then minimum 0 cuts are needed.

\*/

public class PalindromePartition {

public int minCuts(String input) {

assert input != null;

if (input.isEmpty()) {

return 0;

}

char[] array = input.toCharArray();

// determine if substring(i, j) is palindrome

boolean[][] isPa = new boolean[array.length + 1][array.length + 1];

// determine the min cuts for substrings starting from index 0

int[] minCut = new int[array.length + 1];

for (int i = 0; i < minCut.length; i++) {

minCut[i] = i;

}

for (int end = 1; end <= array.length; end++) {

for (int start = end; start >= 1; start--) {

if (start == end || start + 1 == end) {

isPa[start][end] = array[start - 1] == array[end - 1];

} else if (array[start - 1] == array[end - 1]) {

isPa[start][end] = isPa[start + 1][end - 1];

}

if (isPa[start][end]) {

minCut[end] = Math.min(minCut[end], 1 + minCut[start - 1]);

}

}

}

return minCut[array.length] - 1;

}

public static void main(String[] args) {

PalindromePartition solution = new PalindromePartition();

String input = "";

System.out.println(solution.minCuts(input));

input = "a";

System.out.println(solution.minCuts(input));

input = "ab";

System.out.println(solution.minCuts(input));

input = "ababbbabbababa";

System.out.println(solution.minCuts(input));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 22 - 强化练习 2

#### 1. Deep Copy Skip List

/\*\*

\* Make a deep copy of a skip list.

\* Besides the "next" reference, the node in a skip list also

\* contains a reference to another node in the skip list.

\*/

public class CopySkipList {

static class SkipListNode {

int value;

SkipListNode next;

SkipListNode forward;

SkipListNode(int value) {

this.value = value;

}

// only for print purpose.

@Override

public String toString() {

return value + "";

}

}

public SkipListNode copy(SkipListNode head) {

if (head == null) {

return null;

}

HashMap<SkipListNode, SkipListNode> map = new HashMap<SkipListNode, SkipListNode>();

SkipListNode newHead = new SkipListNode(head.value);

map.put(head, newHead);

SkipListNode newCur = newHead;

while (head != null) {

if (head.next != null) {

if (!map.containsKey(head.next)) {

map.put(head.next, new SkipListNode(head.next.value));

}

newCur.next = map.get(head.next);

}

if (head.forward != null) {

if (!map.containsKey(head.forward)) {

map.put(head.forward, new SkipListNode(head.forward.value));

}

newCur.forward = map.get(head.forward);

}

head = head.next;

newCur = newCur.next;

}

return newHead;

}

public static void main(String[] args) {

CopySkipList solution = new CopySkipList();

SkipListNode zero = new SkipListNode(0);

SkipListNode one = new SkipListNode(1);

SkipListNode two = new SkipListNode(2);

SkipListNode three = new SkipListNode(3);

zero.next = one;

one.next = two;

two.next = three;

zero.forward = two;

one.forward = three;

three.forward = one;

SkipListNode copy = solution.copy(zero);

print(copy);

}

public static void print(SkipListNode head) {

while (head != null) {

System.out.println("value: " + head + ", forward node: " + head.forward);

head = head.next;

}

}

}

#### 2. Deep Copy Graph With Possible Cycle

/\*\*

\* Make a deep copy of a graph, the graph could have cycles.

\*/

public class DeepCopyUGraph {

public List<GraphNode> copy(List<GraphNode> graph) {

if (graph == null) {

return null;

}

HashMap<GraphNode, GraphNode> map = new HashMap<GraphNode, GraphNode>();

for (GraphNode node : graph) {

if (!map.containsKey(node)) {

map.put(node, new GraphNode(node.key));

DFS(node, map);

}

}

return new ArrayList<GraphNode>(map.values());

}

private void DFS(GraphNode seed, HashMap<GraphNode, GraphNode> map) {

GraphNode copy = map.get(seed);

for (GraphNode nei : seed.neighbors) {

if (!map.containsKey(nei)) {

map.put(nei, new GraphNode(nei.key));

DFS(nei, map);

}

copy.neighbors.add(map.get(nei));

}

}

}

#### 3. Merge K Sorted Arrays

public class MergeKSortedArray {

public int[] merge(int[][] arrayOfArrays) {

// arrayOfArrays is not null.

assert arrayOfArrays != null;

PriorityQueue<Entry> minHeap = new PriorityQueue<Entry>(11, new MyComparator());

int length = 0;

for (int i = 0; i < arrayOfArrays.length; i++) {

int[] array = arrayOfArrays[i];

length += array.length;

if (array != null && array.length != 0) {

minHeap.offer(new Entry(i, 0, array[0]));

}

}

int[] result = new int[length];

int cur = 0;

while (!minHeap.isEmpty()) {

Entry tmp = minHeap.poll();

result[cur++] = tmp.value;

if (tmp.y + 1 < arrayOfArrays[tmp.x].length) {

tmp.y++;

tmp.value = arrayOfArrays[tmp.x][tmp.y];

minHeap.offer(tmp);

}

}

return result;

}

static class MyComparator implements Comparator<Entry> {

@Override

public int compare(Entry e1, Entry e2) {

if (e1.value < e2.value) {

return -1;

} else if (e1.value > e2.value) {

return 1;

} else {

return 0;

}

}

}

static class Entry {

int x;

int y;

int value;

Entry(int x, int y, int value) {

this.x = x;

this.y = y;

this.value = value;

}

}

}

#### 4. Merge K Sorted Lists

public class MergeKSortedList {

public ListNode merge(List<ListNode> listOfLists) {

// listOfLists is not null.

assert listOfLists != null;

PriorityQueue<ListNode> minHeap = new PriorityQueue<ListNode>(11, new MyComparator());

ListNode dummy = new ListNode(0);

ListNode cur = dummy;

for (ListNode node : listOfLists) {

if (node != null) {

minHeap.offer(node);

}

}

while (!minHeap.isEmpty()) {

cur.next = minHeap.poll();

if (cur.next.next != null) {

minHeap.offer(cur.next.next);

}

cur = cur.next;

}

return dummy.next;

}

static class MyComparator implements Comparator<ListNode> {

@Override

public int compare(ListNode o1, ListNode o2) {

if (o1.value == o2.value) {

return 0;

}

return o1.value < o2.value ? -1 : 1;

}

}

}

#### 5. Binary Search Tree Closest Node To Target

public class ClosestNumberBST {

public TreeNode closest(TreeNode root, int target) {

if (root == null) {

return null;

}

TreeNode result = root;

while (root != null) {

if (root.key == target) {

return root;

} else {

if (Math.abs(root.key - target) < Math.abs(result.key - target)) {

result = root;

}

if (root.key < target) {

root = root.right;

} else {

root = root.left;

}

}

}

return result;

}

}

#### 6. Binary Search Tree Largest Node Smaller Than Target

public class LargestNumberSmallerBST {

public TreeNode largestSmaller(TreeNode root, int target) {

if (root == null) {

return null;

}

TreeNode result = null;

while (root != null) {

if (root.key >= target) {

root = root.left;

} else {

result = root;

root = root.right;

}

}

return result;

}

}

#### 7. Binary Search Tree Delete Target

public class DeleteBST {

public TreeNode delete(TreeNode root, int key) {

if (root == null) {

return null;

} else if (key < root.key) {

root.left = delete(root.left, key);

return root;

} else if (key > root.key) {

root.right = delete(root.right, key);

return root;

} else {

if (root.left == null) {

return root.right;

} else if (root.right == null) {

return root.right;

} else if (root.right.left == null) {

root.right.left = root.left;

return root.right;

} else {

TreeNode newRoot = deleteSmallest(root.right);

newRoot.left = root.left;

newRoot.right = root.right;

return newRoot;

}

}

}

private TreeNode deleteSmallest(TreeNode root) {

while (root.left.left != null) {

root = root.left;

}

TreeNode smallest = root.left;

root.left = root.left.right;

return smallest;

}

}

#### 8. Binary Search Tree Insert Target

public class InsertBST {

public TreeNode insert(TreeNode root, int key) {

if (root == null) {

return new TreeNode(key);

} else if (root.key > key) {

root.left = insert(root.left, key);

} else if (root.key < key) {

root.right = insert(root.right, key);

}

return root;

}

}

#### 9. Cutting Wood

public class CuttingWoodI {

public int minCost(int[] cuts, int length) {

// cuts is not null, length > 0, all cuts are valid numbers.

int[] helper = new int[cuts.length + 2];

helper[0] = 0;

for (int i = 0; i < cuts.length; i++) {

helper[i + 1] = cuts[i];

}

helper[helper.length - 1] = length;

int[][] minCost = new int[helper.length][helper.length];

for (int i = 1; i < helper.length; i++) {

for (int j = i - 1; j >= 0; j--) {

if (j + 1 == i) {

minCost[j][i] = 0;

} else {

minCost[j][i] = Integer.MAX\_VALUE;

for (int k = j + 1; k <= i - 1; k++) {

minCost[j][i] = Math.min(minCost[j][i], minCost[j][k] + minCost[k][i]);

}

minCost[j][i] += helper[i] - helper[j];

}

}

}

return minCost[0][helper.length - 1];

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 23 - 强化练习 3

#### 1. Common Numbers Of Two Sorted Arrays

public class CommonNumbersII {

public List<Integer> common(int[] a, int[] b) {

// a, b is not null.

List<Integer> common = new ArrayList<Integer>();

int i = 0;

int j = 0;

while (i < a.length && j < b.length) {

if (a[i] == b[j]) {

common.add(a[i]);

i++;

j++;

} else if (a[i] < b[j]) {

i++;

} else {

j++;

}

}

return common;

}

}

#### 2. Common Elements In Three Sorted Array

public class CommonElementsII {

public List<Integer> common(int[] a, int[] b, int[] c) {

// a, b, c is not null

List<Integer> common = new ArrayList<Integer>();

int ai = 0;

int bi = 0;

int ci = 0;

while (ai < a.length && bi < b.length && ci < c.length) {

if (a[ai] == b[bi] && b[bi] == c[ci]) {

common.add(a[ai]);

ai++;

bi++;

ci++;

} else if (a[ai] <= b[bi] && a[ai] <= c[ci]) {

ai++;

} else if (b[bi] <= a[ai] && b[bi] <= c[ci]) {

bi++;

} else {

ci++;

}

}

return common;

}

}

#### 3. String Replace

/\*\*

\* Replace all substrings s1 in a string s with s2

\* (with possible minimum memory allocation, in-place if possible).

\*/

public class StrReplace {

// Solution with using StringBuilder and substring().

public String replace(String input, String s, String t) {

// input, s, t is not null, s is not empty

assert input != null && s != null && t != null;

assert !s.isEmpty();

StringBuilder sb = new StringBuilder();

int index = input.indexOf(s);

while (index != -1) {

sb.append(input.substring(0, index)).append(t);

input = input.substring(index + s.length());

index = input.indexOf(s);

}

sb.append(input);

return sb.toString();

}

// Solution with char array.

public String replaceII(String input, String s, String t) {

// input, s, t is not null, s is not empty

assert input != null && s != null && t != null;

assert !s.isEmpty();

if (s.length() >= t.length()) {

return replaceShorter(input, s, t);

} else {

return replaceLonger(input, s, t);

}

}

private String replaceShorter(String input, String s, String t) {

char[] array = input.toCharArray();

int end = 0;

for (int i = 0; i < input.length();) {

if (i <= input.length() - s.length() && equalSubArray(input, i, s)) {

copyFromLeft(array, end, t);

i += s.length();

end += t.length();

} else {

array[end++] = input.charAt(i++);

}

}

return new String(array, 0, end);

}

private String replaceLonger(String input, String s, String t) {

ArrayList<Integer> matches = new ArrayList<Integer>();

for (int i = 0; i <= input.length() - s.length(); i++) {

if (equalSubArray(input, i, s)) {

matches.add(i + s.length() - 1);

}

}

int newLength = input.length() + matches.size() \* (t.length() - s.length());

char[] result = new char[newLength];

int lastIndex = matches.size() - 1;

int end = newLength - 1;

for (int i = input.length() - 1; i >= 0;) {

if (lastIndex >= 0 && i == matches.get(lastIndex)) {

copyFromRight(result, end, t);

lastIndex--;

i -= s.length();

end -= t.length();

} else {

result[end--] = input.charAt(i--);

}

}

return new String(result);

}

public boolean equalSubArray(String input, int index, String s) {

for (int i = 0; i < s.length(); i++) {

if (input.charAt(index + i) != s.charAt(i)) {

return false;

}

}

return true;

}

public void copyFromLeft(char[] array, int index, String t) {

for (int i = 0; i < t.length(); i++) {

array[index++] = t.charAt(i);

}

}

public void copyFromRight(char[] array, int index, String t) {

for (int i = t.length() - 1; i >= 0; i--) {

array[index--] = t.charAt(i);

}

}

public static void main(String[] args) {

StrReplace solution = new StrReplace();

String input = "abcaabcabcabca";

String s = "abc";

String t = "xy";

System.out.println(solution.replace(input, s, t));

System.out.println(solution.replaceII(input, s, t));

t = "xyz";

System.out.println(solution.replace(input, s, t));

System.out.println(solution.replaceII(input, s, t));

t = "opqr";

System.out.println(solution.replace(input, s, t));

System.out.println(solution.replaceII(input, s, t));

}

}

#### 4. Decompress String II

/\*\*

\* Given a string such as "a3b1c4d0" --> "aaabcccc".

\* Assumption:

\* 1). number of consecutive characters is in the range of [0, 9],

\* 2). valid characters in the string is 'a'-'z'.

\*/

public class DecompressII {

// Assumption: the input is not null, and it is valid String.

// using StringBuilder.

public String decompress(String input) {

// input is not null

char[] array = input.toCharArray();

StringBuilder sb = new StringBuilder();

for (int i = 0; i < array.length; i++) {

char ch = array[i++];

int count = array[i] - '0';

for (int c = 0; c < count; c++) {

sb.append(ch);

}

}

return sb.toString();

}

// Assumption: the input is not null, and it is valid String.

// Using char array.

public String decompressII(String input) {

assert input != null;

if (input.isEmpty()) {

return input;

}

char[] array = input.toCharArray();

return decodeLong(array, decodeShort(array));

}

private int decodeShort(char[] input) {

int end = 0;

for (int i = 0; i < input.length; i += 2) {

int count = getDigit(input[i + 1]);

if (count <= 2) {

for (int j = 0; j < count; j++) {

input[end++] = input[i];

}

} else {

input[end++] = input[i];

input[end++] = input[i + 1];

}

}

return end;

}

private String decodeLong(char[] input, int length) {

int newLength = length;

for (int i = 0; i < length; i++) {

if (isDigit(input[i])) {

newLength += getDigit(input[i]) - 2;

}

}

char[] result = new char[newLength];

int end = newLength - 1;

for (int i = length - 1; i >= 0; i--) {

if (isDigit(input[i])) {

int count = getDigit(input[i--]);

for (int j = 0; j < count; j++) {

result[end--] = input[i];

}

} else {

result[end--] = input[i];

}

}

return new String(result);

}

private int getDigit(char digit) {

return digit - '0';

}

private boolean isDigit(char digit) {

return digit - '0' >= 0 && digit - '0' <= 9;

}

public static void main(String[] args) {

DecompressII solution = new DecompressII();

String input = "a0b3c1d4e2";

System.out.println(solution.decompress(input));

System.out.println(solution.decompressII(input));

}

}

#### 5. Check If Binary Tree Is Balanced

public class CheckBalanced {

public boolean isBalanced(TreeNode root) {

if (root == null) {

return true;

}

return height(root) != -1;

}

private int height(TreeNode root) {

if (root == null) {

return 0;

}

int leftHeight = height(root.left);

if (leftHeight == -1) {

return -1;

}

int rightHeight = height(root.right);

if (rightHeight == -1) {

return -1;

}

if (Math.abs(leftHeight - rightHeight) > 1) {

return -1;

}

return Math.max(leftHeight, rightHeight) + 1;

}

}

#### 6. Longest Common Substring

public class LCS {

// Assumption: s1, s2 is not null.

public String LCSubstring(String s1, String s2) {

assert s1 != null && s2 != null;

int s1End = 0;

int longest = 0;

int[][] length = new int[s1.length() + 1][s2.length() + 1];

for (int i = 1; i <= s1.length(); i++) {

for (int j = 1; j <= s2.length(); j++) {

if (s1.charAt(i - 1) == s2.charAt(j - 1)) {

length[i][j] = length[i - 1][j - 1] + 1;

if (length[i][j] > longest) {

s1End = i;

longest = length[i][j];

}

}

}

}

return s1.substring(s1End - longest, s1End);

}

}

#### 7. Longest Common Subsequence

public class LongestCommonSubSequence {

public int longest(String s, String t) {

// s, t is not null

int[][] longest = new int[s.length() + 1][t.length() + 1];

for (int i = 1; i <= s.length(); i++) {

for (int j = 1; j <= t.length(); j++) {

if (s.charAt(i - 1) == t.charAt(j - 1)) {

longest[i][j] = longest[i - 1][j - 1] + 1;

} else {

longest[i][j] = Math.max(longest[i - 1][j], longest[i][j - 1]);

}

}

}

return longest[s.length()][t.length()];

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 24 - 强化练习 4

#### 1. Reverse Singly Linked List

/\*\*

\* Reverse a singled linked list.

\*/

public class ReverseLinkedList {

public ListNode reverseRecursive(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode newHead = reverseRecursive(head.next);

head.next.next = head;

head.next = null;

return newHead;

}

public ListNode reverseIterative(ListNode head) {

ListNode prev = null;

ListNode cur = head;

while (cur != null) {

ListNode next = cur.next;

cur.next = prev;

prev = cur;

cur = next;

}

return prev;

}

public static void main(String[] args) {

ReverseLinkedList solution = new ReverseLinkedList();

ListNode head = null;

ListNode.printList(solution.reverseIterative(head));

head = null;

ListNode.printList(solution.reverseRecursive(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.reverseIterative(head));

head = ListNode.arrayToList(new int[] { 1 });

ListNode.printList(solution.reverseRecursive(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4 });

ListNode.printList(solution.reverseIterative(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4 });

ListNode.printList(solution.reverseRecursive(head));

}

}

#### 2. Reverse Linked List Pair By Pair

/\*\*

\* Reverse a singled linked list's nodes pair by pair.

\* Example:

\* 1 -> 2 -> 3 -> 4 -> 5 -> NULL will be transferred to

\* 2 -> 1 -> 4 -> 3 -> 5 -> NULL

\*/

public class ReverseListByPair {

public ListNode reverseByPair(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode temp = reverseByPair(head.next.next);

ListNode newHead = head.next;

newHead.next = head;

head.next = temp;

return newHead;

}

public ListNode reverseByPairIterative(ListNode head) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

while (head != null && head.next != null) {

ListNode next = head.next.next;

prev.next = head.next;

prev.next.next = head;

head.next = next;

prev = head;

head = next;

}

return dummy.next;

}

public static void main(String[] args) {

ReverseListByPair solution = new ReverseListByPair();

ListNode head = null;

System.out.println(solution.reverseByPair(head));

head = null;

System.out.println(solution.reverseByPairIterative(head));

head = new ListNode(1);

ListNode.printList(solution.reverseByPair(head));

head = new ListNode(1);

ListNode.printList(solution.reverseByPairIterative(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.reverseByPair(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3 });

ListNode.printList(solution.reverseByPairIterative(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4, 5, 6 });

ListNode.printList(solution.reverseByPair(head));

head = ListNode.arrayToList(new int[] { 1, 2, 3, 4, 5, 6 });

ListNode.printList(solution.reverseByPairIterative(head));

}

}

#### 3. Reverse Binary Tree Upside Down

/\*\*

\* Given a binary tree where all the right nodes are leaf nodes,

\* flip it upside down and turn it into a tree with left leaf nodes.

\* For example, turn these:

\* 1 4

\* / \ / \

\* 2 3 --> 2 5

\* / \ / \

\* 4 5 1 3

\*/

public class ReverseBinaryTree {

public TreeNode reverse(TreeNode root) {

if (root == null || root.left == null) {

return root;

}

TreeNode newRoot = reverse(root.left);

root.left.left = root;

root.left.right = root.right;

root.left = null;

root.right = null;

return newRoot;

}

public TreeNode reverseIterative(TreeNode root) {

TreeNode prev = null;

TreeNode prevRight = null;

while (root != null) {

TreeNode next = root.left;

TreeNode curRight = root.right;

root.left = prev;

root.right = prevRight;

prev = root;

prevRight = curRight;

root = next;

}

return prev;

}

public static void main(String[] args) {

ReverseBinaryTree solution = new ReverseBinaryTree();

TreeNode root = null;

System.out.println(solution.reverse(root));

System.out.println(solution.reverseIterative(root));

root = TreeNode.reConstruct(new String[] { "1", "2", "4", "#", "#", "5", "#", "#", "3", "#",

"#" });

TreeNode.preOrder(solution.reverse(root));

System.out.println();

root = TreeNode.reConstruct(new String[] { "1", "2", "4", "#", "#", "5", "#", "#", "3", "#",

"#" });

TreeNode.preOrder(solution.reverseIterative(root));

System.out.println();

}

}

#### 4. All Valid Permutations Of l (), m [], n {}

/\*\*

\* Print out all valid permutations of l (), m [], n {}.

\*/

public class ValidParentheses {

public static final char[] PARENS = { '(', ')', '[', ']', '{', '}' };

public void validParentheses(int l, int m, int n) {

assert l >= 0 && m >= 0 && n >= 0;

StringBuilder temp = new StringBuilder();

Deque<Character> stack = new LinkedList<Character>();

validHelper(new int[] { l, l, m, m, n, n }, temp, 2 \* l + 2 \* m + 2 \* n, stack);

}

private void validHelper(int[] remain, StringBuilder temp, int targetLength,

Deque<Character> stack) {

if (temp.length() == targetLength) {

System.out.println(temp);

return;

}

for (int i = 0; i < 6; i++) {

if (remain[i] > 0) {

if (i % 2 == 0) {

temp.append(PARENS[i]);

remain[i]--;

stack.offerFirst(PARENS[i]);

validHelper(remain, temp, targetLength, stack);

stack.pollFirst();

remain[i]++;

temp.deleteCharAt(temp.length() - 1);

} else if (!stack.isEmpty() && stack.peekFirst() == PARENS[i - 1]) {

temp.append(PARENS[i]);

remain[i]--;

stack.pollFirst();

validHelper(remain, temp, targetLength, stack);

stack.offerFirst(PARENS[i - 1]);

remain[i]++;

temp.deleteCharAt(temp.length() - 1);

}

}

}

}

public static void main(String[] args) {

ValidParentheses solution = new ValidParentheses();

int l = 1;

int m = 2;

int n = 3;

solution.validParentheses(l, m, n);

}

}

#### 5. Eight Queens

/\*\*

\* Print all valid configurations for 8 queens on a chess board

\* such that each of the queens can not hit another in one move.

\*/

public class EightQueen {

public void eightQueen() {

int[] config = new int[8];

NQueen(config, 0);

}

public void NQueen(int[] config, int index) {

assert config != null && config.length > 0;

if (index == config.length) {

System.out.println(Arrays.toString(config));

return;

}

for (int i = 0; i < config.length; i++) {

config[index] = i;

if (valid(config, index)) {

NQueen(config, index + 1);

}

}

}

private boolean valid(int[] config, int index) {

for (int i = 0; i < index; i++) {

if (config[i] == config[index] || config[i] - config[index] == i - index

|| config[i] - config[index] == index - i) {

return false;

}

}

return true;

}

public static void main(String[] args) {

EightQueen solution = new EightQueen();

solution.eightQueen();

}

}

#### 6. 2 Sum

/\*\*

\* Determine if there exists two numbers in an integer array,

\* the sum of the two numbers is a given target value.

\*/

public class TwoSum {

public boolean twoSum(int[] array, int target) {

assert array != null && array.length >= 2;

Arrays.sort(array);

int left = 0;

int right = array.length - 1;

while (left < right) {

int sum = array[left] + array[right];

if (sum == target) {

return true;

} else if (sum < target) {

left++;

} else {

right--;

}

}

return false;

}

public boolean twoSumI(int[] array, int target) {

assert array != null && array.length >= 2;

HashSet<Integer> set = new HashSet<Integer>();

for (int number : array) {

if (set.contains(target - number)) {

return true;

}

set.add(number);

}

return false;

}

public static void main(String[] args) {

TwoSum solution = new TwoSum();

int[] array = { 1, 2, 4 };

int target = 4;

System.out.println(solution.twoSum(array, target));

System.out.println(solution.twoSumI(array, target));

array = new int[] { 1, 2, 2, 4 };

System.out.println(solution.twoSum(array, target));

System.out.println(solution.twoSumI(array, target));

}

}

#### 7. 3 Sum

/\*\*

\* Determine if there exists three numbers in an integer array,

\* the sum of the three numbers is a given target value.

\*/

public class ThreeSum {

public boolean threeSum(int[] array, int target) {

assert array != null && array.length >= 3;

Arrays.sort(array);

for (int i = 0; i < array.length - 2; i++) {

int left = i + 1;

int right = array.length - 1;

while (left < right) {

int sum = array[left] + array[right];

if (sum == target - array[i]) {

return true;

} else if (sum < target - array[i]) {

left++;

} else {

right--;

}

}

}

return false;

}

public static void main(String[] args) {

ThreeSum solution = new ThreeSum();

int[] array = { 1, 2, 3 };

int target = 7;

System.out.println(solution.threeSum(array, target));

array = new int[] { 1, 2, 3, 3, 4 };

System.out.println(solution.threeSum(array, target));

}

}

#### 8. 4 Sum

/\*\*

\* Determine if there exists four numbers in an integer array,

\* the sum of the four numbers is a given target value.

\*/

public class FourSum {

// Method 1

public boolean fourSum(int[] array, int target) {

assert array != null && array.length >= 4;

Arrays.sort(array);

for (int i = 0; i < array.length - 3; i++) {

for (int j = i + 1; j < array.length - 2; j++) {

int left = j + 1;

int right = array.length - 1;

int curTarget = target - array[i] - array[j];

while (left < right) {

int sum = array[left] + array[right];

if (sum == curTarget) {

return true;

} else if (sum < curTarget) {

left++;

} else {

right--;

}

}

}

}

return false;

}

static class Element implements Comparable<Element> {

int left;

int right;

int sum;

Element(int left, int right, int sum) {

this.left = left;

this.right = right;

this.sum = sum;

}

@Override

public int compareTo(Element another) {

if (this.sum != another.sum) {

return this.sum < another.sum ? -1 : 1;

} else if (this.right != another.right) {

return this.right < another.right ? -1 : 1;

} else if (this.left != another.left) {

return this.left < another.left ? -1 : 1;

}

return 0;

}

}

// Method 2: O(n^2 \* logn)

public boolean fourSumI(int[] array, int target) {

assert array != null && array.length >= 4;

Element[] pairSum = getPairSum(array);

Arrays.sort(pairSum);

int left = 0;

int right = pairSum.length - 1;

// pairSum are sorted by sum, then right index, then left index.

while (left < right) {

// only return true if two pair sums' sum is target and the larger pair sum's

// left index > smaller pair sum's large index.

if (pairSum[left].sum + pairSum[right].sum == target

&& pairSum[left].right < pairSum[right].left) {

return true;

} else if (pairSum[left].sum + pairSum[right].sum < target) {

left++;

} else {

// when two pair sums' sum > target, right--

// when two pair sums' sum == target but larger pair sum's left index

// <= smaller pair sum's large index, we need to do right--,

// because the only thing we can guarantee is that

// right now the smaller pair sum's right index is the smallest one among

// all pairSums with the same sum.

right--;

}

}

return false;

}

private Element[] getPairSum(int[] array) {

Element[] pairSum = new Element[array.length \* (array.length - 1) / 2];

int curIndex = 0;

for (int i = 1; i < array.length; i++) {

for (int j = 0; j < i; j++) {

pairSum[curIndex++] = new Element(j, i, array[i] + array[j]);

}

}

return pairSum;

}

static class Pair {

int left;

int right;

Pair(int left, int right) {

this.left = left;

this.right = right;

}

}

// Method 3: HashMap O(n ^ 2)

public boolean fourSumII(int[] array, int target) {

assert array != null && array.length >= 4;

HashMap<Integer, Pair> map = new HashMap<Integer, Pair>();

// the order of traversing i, j is not arbitrary, we should guarantee

// we can always look at the pair with the smallest right index.

for (int i = 1; i < array.length; i++) {

for (int j = 0; j < i; j++) {

int pairSum = array[j] + array[i];

if (map.containsKey(target - pairSum) && map.get(target - pairSum).right < j) {

return true;

}

// we only need to store the pair with smallest right index.

if (!map.containsKey(pairSum)) {

map.put(pairSum, new Pair(j, i));

}

}

}

return false;

}

public static void main(String[] args) {

FourSum solution = new FourSum();

int[] array = { 1, 2, 3, 4 };

int target = 10;

System.out.println(solution.fourSum(array, target));

System.out.println(solution.fourSumI(array, target));

System.out.println(solution.fourSumII(array, target));

target = 12;

System.out.println(solution.fourSum(array, target));

System.out.println(solution.fourSumI(array, target));

System.out.println(solution.fourSumII(array, target));

target = 12;

array = new int[] { 3, 1, 3, 3, 8, 3 };

System.out.println(solution.fourSum(array, target));

array = new int[] { 3, 1, 3, 3, 8, 3 };

System.out.println(solution.fourSumI(array, target));

array = new int[] { 3, 1, 3, 3, 8, 3 };

System.out.println(solution.fourSumII(array, target));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 25 - 强化练习(Recursion 总结)

#### 1. Reverse String Using Recursion

public class ReverseString {

// Method 1: reverse the String recursively

public String reverseRecursive(String input) {

if (input == null) {

return null;

}

char[] array = input.toCharArray();

reverseHelper(array, 0);

return new String(array);

}

private void reverseHelper(char[] array, int index) {

if (index != array.length) {

// reverse the substring from index + 1 first

reverseHelper(array, index + 1);

// move the char at index to the correct position

char tmp = array[index];

for (int i = index; i < array.length - 1; i++) {

array[i] = array[i + 1];

}

array[array.length - 1] = tmp;

}

}

// Method 2: reverse the String iteratively

public String reverseIterative(String input) {

if (input == null) {

return null;

}

char[] array = input.toCharArray();

for (int left = 0, right = array.length - 1; left < right; left++, right--) {

swap(array, left, right);

}

return new String(array);

}

private void swap(char[] array, int left, int right) {

char tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

public static void main(String[] args) {

ReverseString sol = new ReverseString();

String input = "";

System.out.println(sol.reverseIterative(input));

System.out.println(sol.reverseRecursive(input));

input = "abc";

System.out.println(sol.reverseIterative(input));

System.out.println(sol.reverseRecursive(input));

input = "abcdefgh";

System.out.println(sol.reverseIterative(input));

System.out.println(sol.reverseRecursive(input));

}

}

#### 2. Abbreviation Matching

public class AbbrevMatch {

// Assumptions: word and abbrev are not null

public boolean match(String word, String abbrev) {

assert word != null && abbrev != null;

return match(word, 0, abbrev, 0);

}

private boolean match(String word, int wordIndex, String abbrev, int abbIndex) {

if (wordIndex == word.length() && abbIndex == abbrev.length()) {

return true;

}

if (wordIndex >= word.length() || abbIndex >= abbrev.length()) {

return false;

}

if (!isDigit(abbrev.charAt(abbIndex))) {

return word.charAt(wordIndex) == abbrev.charAt(abbIndex)

&& match(word, wordIndex + 1, abbrev, abbIndex + 1);

}

int count = 0;

while (abbIndex < abbrev.length() && isDigit(abbrev.charAt(abbIndex))) {

count = count \* 10 + getDigit(abbrev.charAt(abbIndex));

abbIndex++;

}

return match(word, wordIndex + count, abbrev, abbIndex);

}

private boolean isDigit(char ch) {

return ch - '0' >= 0 && ch - '0' <= 9;

}

private int getDigit(char ch) {

return ch - '0';

}

public static void main(String[] args) {

AbbrevMatch sol = new AbbrevMatch();

String word = "book";

String abbrev = "4";

System.out.println(sol.match(word, abbrev));

word = "book";

abbrev = "b1o1";

System.out.println(sol.match(word, abbrev));

word = "book";

abbrev = "2o2";

System.out.println(sol.match(word, abbrev));

word = "sophisticated";

abbrev = "2ph4cate1";

System.out.println(sol.match(word, abbrev));

word = "sophisticated";

abbrev = "s11d";

System.out.println(sol.match(word, abbrev));

}

}

#### 3. Store Number Of Nodes In Left Subtree For Each Node

public class LeftSubTreeNumber {

// We will use the root.key to store the number of nodes in left subtree.

public int generate(TreeNode root) {

if (root == null) {

return 0;

}

root.key = generate(root.left);

return root.key + generate(root.right) + 1;

}

}

#### 4. Node With Max Difference Number Of Nodes In Left/Right Subtree

public class MaxDiffDescendents {

private TreeNode result;

private int max;

public TreeNode max(TreeNode root) {

result = null;

max = 0;

maxHelper(root);

return result;

}

public int maxHelper(TreeNode root) {

if (root == null) {

return 0;

}

int left = maxHelper(root.left);

int right = maxHelper(root.right);

int diff = Math.abs(left - right);

if (result == null || diff > max) {

max = diff;

result = root;

}

return left + right + 1;

}

}

#### 5. Binary Tree Maximum Path Sum II (Any Node To Any Node)

public class MaxPathSumII {

// Assumptions: root is not null;

public int maxPath(TreeNode root) {

assert root != null;

int[] max = new int[] { root.key };

helper(root, max);

return max[0];

}

private int helper(TreeNode root, int[] max) {

if (root == null) {

return 0;

}

int leftSingle = helper(root.left, max);

int rightSingle = helper(root.right, max);

int maxSingle = Math.max(0, Math.max(leftSingle, rightSingle)) + root.key;

int maxCur = Math.max(maxSingle, root.key + leftSingle + rightSingle);

max[0] = Math.max(max[0], maxCur);

return maxSingle;

}

}

#### 6. Binary Tree Path Sum To Target

public class PathSumToTarget {

public boolean exist(TreeNode root, int sum) {

if (root == null) {

return false;

}

List<TreeNode> path = new ArrayList<TreeNode>();

return helper(root, path, sum);

}

private boolean helper(TreeNode root, List<TreeNode> path, int sum) {

path.add(root);

int tmp = 0;

for (int i = path.size() - 1; i >= 0; i--) {

tmp += path.get(i).key;

if (tmp == sum) {

path.remove(path.size() - 1);

return true;

}

}

if (root.left != null && helper(root.left, path, sum)) {

path.remove(path.size() - 1);

return true;

}

if (root.right != null && helper(root.right, path, sum)) {

path.remove(path.size() - 1);

return true;

}

path.remove(path.size() - 1);

return false;

}

}

#### 7. Binary Tree Maximum Path Sum III (Any Node To Any Node On The Same Path From Root To Leaf)

public class MaxPathSumIII {

private int max;

public int maxPathSum(TreeNode root) {

// root is not null.

assert root != null;

max = root.key;

helper(root);

return max;

}

private int helper(TreeNode root) {

if (root == null) {

return 0;

}

int left = helper(root.left);

int right = helper(root.right);

int curMax = Math.max(root.key, Math.max(root.key + left, root.key + right));

max = Math.max(max, curMax);

return curMax;

}

}

#### 8. Reconstruct Complete Binary Tree From Level Order

public class ReconstructCompleteTree {

// Assumptions: levelOrder is not null.

public TreeNode construct(int[] levelOrder) {

assert levelOrder != null;

return construct(levelOrder, 0);

}

private TreeNode construct(int[] levelOrder, int index) {

if (index >= levelOrder.length) {

return null;

}

TreeNode root = new TreeNode(levelOrder[index]);

root.left = construct(levelOrder, 2 \* index + 1);

root.right = construct(levelOrder, 2 \* index + 2);

return root;

}

}

#### 9. Reconstruct Binary Search Tree From Pre Order

public TreeNode reconstruct(int[] pre) {

// preorder is not null, no duplicates.

return helper(pre, 0, pre.length - 1);

}

private TreeNode helper(int[] pre, int left, int right) {

if (left > right) {

return null;

}

if (left == right) {

return new TreeNode(pre[left]);

}

TreeNode root = new TreeNode(pre[left]);

int mid = findMid(pre, left + 1, right, pre[left]);

root.left = helper(pre, left + 1, mid);

root.right = helper(pre, mid + 1, right);

return root;

}

private int findMid(int[] pre, int left, int right, int target) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (pre[mid] > target) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return right;

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 26 - 强化练习 5

#### 1. Max Length Product Of Two Strings Without Common Characters

public class LargestLengthProduct {

public int largestProduct(List<String> dict) {

// dict is not null and length >= 2, there is no null String in the dict.

// only used characters is 'a' - 'z'.

HashMap<String, Integer> bitMasks = getBitMasks(dict);

Collections.sort(dict, new Comparator<String>() {

@Override

public int compare(String s0, String s1) {

if (s0.length() == s1.length()) {

return 0;

} else if (s0.length() < s1.length()) {

return 1;

} else {

return -1;

}

}

});

int largest = 0;

for (int i = 1; i < dict.size(); i++) {

for (int j = 0; j < i; j++) {

int prod = dict.get(i).length() \* dict.get(j).length();

if (prod <= largest) {

break;

}

int iMask = bitMasks.get(dict.get(i));

int jMask = bitMasks.get(dict.get(j));

if ((iMask & jMask) == 0) {

largest = prod;

}

}

}

return largest;

}

private HashMap<String, Integer> getBitMasks(List<String> dict) {

HashMap<String, Integer> map = new HashMap<String, Integer>();

for (String str : dict) {

int bitMask = 0;

for (int i = 0; i < str.length(); i++) {

bitMask |= 1 << (str.charAt(i) - 'a');

}

map.put(str, bitMask);

}

return map;

}

}

#### 2. Kth Smallest Number With Only 3,5,7 As Factors

/\*\*

\* How to find the k-th smallest value of f,

\* in the f(x,y,z) = 3^x \* 5^y \* 7^z (x > 0, y>0, z>0).

\*/

public class KthSmallestProduct {

// Method 1: BFS

public int kth(int K) {

assert K > 0;

PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(K);

HashSet<Integer> visited = new HashSet<Integer>();

minHeap.offer(3 \* 5 \* 7);

visited.add(3 \* 5 \* 7);

while (K > 1) {

int current = minHeap.poll();

if (visited.add(3 \* current)) {

minHeap.offer(3 \* current);

}

if (visited.add(5 \* current)) {

minHeap.offer(5 \* current);

}

if (visited.add(7 \* current)) {

minHeap.offer(7 \* current);

}

K--;

}

return minHeap.peek();

}

// Method 2: use 3 deques.

public int kthI(int K) {

assert K > 0;

int seed = 3 \* 5 \* 7;

Deque<Integer> three = new LinkedList<Integer>();

Deque<Integer> five = new LinkedList<Integer>();

Deque<Integer> seven = new LinkedList<Integer>();

three.add(seed \* 3);

five.add(seed \* 5);

seven.add(seed \* 7);

int result = seed;

while (K > 1) {

if (three.peekFirst() < five.peekFirst() && three.peekFirst() < seven.peekFirst()) {

result = three.pollFirst();

three.offerLast(result \* 3);

five.offerLast(result \* 5);

seven.offerLast(result \* 7);

} else if (five.peekFirst() < three.peekFirst() && five.peekFirst() < seven.peekFirst()) {

result = five.pollFirst();

five.offerLast(result \* 5);

seven.offerLast(result \* 7);

} else {

result = seven.pollFirst();

seven.offerLast(result \* 7);

}

K--;

}

return result;

}

public static void main(String[] args) {

KthSmallestProduct solution = new KthSmallestProduct();

int K = 1;

System.out.println(solution.kth(K));

System.out.println(solution.kthI(K));

K = 5;

System.out.println(solution.kth(K));

System.out.println(solution.kthI(K));

K = 10;

System.out.println(solution.kth(K));

System.out.println(solution.kthI(K));

}

}

#### 3. Kth Closest Point To <0, 0, 0>

\*\*

\* Given three arrays with numbers in ascending order.

\* Pull one number from each array to form a coordinate <x,y,z> in a 3D space.

\* How to find the coordinates of the points that is k-th closest to <0,0,0> ?

\*/

public class KthClosest {

static class Triple {

int v1;

int v2;

int v3;

Triple(int v1, int v2, int v3) {

this.v1 = v1;

this.v2 = v2;

this.v3 = v3;

}

@Override

public boolean equals(Object obj) {

if (obj == this) {

return true;

}

if (!(obj instanceof Triple)) {

return false;

}

Triple another = (Triple) obj;

return this.v1 == another.v1 && this.v2 == another.v2 && this.v3 == another.v3;

}

@Override

public int hashCode() {

return v1 \* 31 \* 31 + v2 \* 31 + v3;

}

@Override

public String toString() {

return v1 + " " + v2 + " " + v3;

}

}

public Triple kthClosest(final int[] a1, final int[] a2, final int[] a3, int K) {

assert a1 != null && a2 != null && a3 != null;

assert a1.length > 0 && a2.length > 0 && a3.length > 0;

assert K >= 1 && K <= a1.length \* a2.length \* a3.length;

PriorityQueue<Triple> minHeap = new PriorityQueue<Triple>(K, new Comparator<Triple>() {

@Override

public int compare(Triple t1, Triple t2) {

int distance1 = distance(a1[t1.v1], a2[t1.v2], a3[t1.v3]);

int distance2 = distance(a1[t2.v1], a2[t2.v2], a3[t2.v3]);

if (distance1 == distance2) {

return 0;

}

return distance1 < distance2 ? -1 : 1;

}

});

HashSet<Triple> visited = new HashSet<Triple>();

Triple seed = new Triple(0, 0, 0);

minHeap.offer(seed);

visited.add(seed);

while (K > 0) {

Triple current = minHeap.poll();

List<Triple> neighbors = getNeighbors(current, a1, a2, a3);

for (Triple neighbor : neighbors) {

if (visited.add(neighbor)) {

minHeap.offer(neighbor);

}

}

K--;

}

Triple result = minHeap.poll();

result.v1 = a1[result.v1];

result.v2 = a2[result.v2];

result.v3 = a3[result.v3];

return result;

}

private List<Triple> getNeighbors(Triple current, int[] a1, int[] a2, int[] a3) {

List<Triple> neighbors = new ArrayList<Triple>();

if (current.v1 + 1 < a1.length) {

neighbors.add(new Triple(current.v1 + 1, current.v2, current.v3));

}

if (current.v2 + 1 < a2.length) {

neighbors.add(new Triple(current.v1, current.v2 + 1, current.v3));

}

if (current.v3 + 1 < a3.length) {

neighbors.add(new Triple(current.v1, current.v2, current.v3 + 1));

}

return neighbors;

}

private int distance(int x, int y, int z) {

return x \* x + y \* y + z \* z;

}

public static void main(String[] args) {

KthClosest solution = new KthClosest();

int[] a1 = { 1, 2, 3 };

int[] a2 = { 1, 2 };

int[] a3 = { 1, 2, 3, 4 };

int K = 1;

System.out.println(solution.kthClosest(a1, a2, a3, K));

K = 5;

System.out.println(solution.kthClosest(a1, a2, a3, K));

K = 10;

System.out.println(solution.kthClosest(a1, a2, a3, K));

}

}

#### 4. Place To Put Chair I

/\*\*

\* Given a gym with k equipments, and some obstacles.

\* Let’s say we bought a chair and wanted to put this chair into the gym

\* such that the sum of the shortest path cost from the chair to the k

\* equipments is minimal.

\*

\* Assumption:

\* 1). The cost from one cell to any of its neighbors(up/down/left/right) is 1.

\* 2). 'E' denotes an equipment, 'O' denotes an obstacle.

\* 3). The chair can not be put on equipment or obstacle.

\*/

public class ShortestPathCostSum {

static class Pair {

int x;

int y;

Pair(int x, int y) {

this.x = x;

this.y = y;

}

@Override

public String toString() {

return x + " " + y;

}

}

// Assumption:

// gym is N \* N.

// 1).The cost from one cell to its neighbors(up/down/left/right) is 1.

// 2). 'E' denotes an equipment, 'O' denotes an obstacle.

// 3). The chair can not be put on equipment or obstacle.

public Pair shortestPathCostSum(char[][] gym) {

assert gym != null;

int len = gym.length;

int[][] costSum = new int[len][len];

Pair result = null;

for (int i = 0; i < len; i++) {

for (int j = 0; j < len; j++) {

if (gym[i][j] == 'E') {

if (!addCost(gym, costSum, i, j)) {

return null;

}

}

}

}

for (int i = 0; i < len; i++) {

for (int j = 0; j < len; j++) {

if (gym[i][j] != 'O' && gym[i][j] != 'E') {

if (result == null) {

result = new Pair(i, j);

} else if (costSum[i][j] < costSum[result.x][result.y]) {

result.x = i;

result.y = j;

}

}

}

}

return result;

}

private boolean addCost(char[][] gym, int[][] costSum, int i, int j) {

int len = gym.length;

boolean[][] visited = new boolean[len][len];

Queue<Pair> queue = new LinkedList<Pair>();

queue.offer(new Pair(i, j));

visited[i][j] = true;

int cost = 0;

while (!queue.isEmpty()) {

int size = queue.size();

for (int ith = 0; ith < size; ith++) {

Pair cur = queue.poll();

costSum[cur.x][cur.y] += cost;

List<Pair> neighbors = getNeighbors(cur, gym, len);

for (Pair neighbor : neighbors) {

if (!visited[neighbor.x][neighbor.y]) {

queue.add(neighbor);

visited[neighbor.x][neighbor.y] = true;

}

}

}

cost++;

}

for (int x = 0; x < len; x++) {

for (int y = 0; y < len; y++) {

if (gym[x][y] == 'E' && !visited[x][y]) {

return false;

}

}

}

return true;

}

private List<Pair> getNeighbors(Pair cur, char[][] gym, int len) {

List<Pair> neighbors = new ArrayList<Pair>();

if (cur.x + 1 < len && gym[cur.x + 1][cur.y] != 'O') {

neighbors.add(new Pair(cur.x + 1, cur.y));

}

if (cur.x - 1 >= 0 && gym[cur.x - 1][cur.y] != 'O') {

neighbors.add(new Pair(cur.x - 1, cur.y));

}

if (cur.y + 1 < len && gym[cur.x][cur.y + 1] != 'O') {

neighbors.add(new Pair(cur.x, cur.y + 1));

}

if (cur.y - 1 >= 0 && gym[cur.x][cur.y - 1] != 'O') {

neighbors.add(new Pair(cur.x, cur.y - 1));

}

return neighbors;

}

public static void main(String[] args) {

ShortestPathCostSum solution = new ShortestPathCostSum();

char[][] gym = { { 'O', 'E', ' ', ' ' }, { ' ', ' ', ' ', 'E' }, { 'E', 'O', 'O', ' ' },

{ ' ', 'E', ' ', 'E' } };

System.out.println(solution.shortestPathCostSum(gym));

}

}

#### 5. String Conversion

public class StringConversion {

public String convert(String input) {

// input is not null

assert input != null;

char[] array = input.toCharArray();

if (array.length % 2 == 0) {

reorder(array, 0, array.length - 1);

} else {

reorder(array, 0, array.length - 2);

}

return new String(array);

}

private void reorder(char[] array, int left, int right) {

int length = right - left + 1;

if (length <= 2) {

return;

}

int mid = left + (right - left) / 2;

int lmid = mid - (length + 2) / 4 + 1;

int rmid = mid + length / 4;

reverse(array, lmid, mid);

reverse(array, mid + 1, rmid);

reverse(array, lmid, rmid);

if (length % 4 == 0) {

reorder(array, left, mid);

reorder(array, mid + 1, right);

} else {

reorder(array, left, mid - 1);

reorder(array, mid, right);

}

}

private void reverse(char[] array, int left, int right) {

while (left < right) {

char tmp = array[left];

array[left] = array[right];

array[right] = tmp;

left++;

right--;

}

}

}

#### 6. Largest Rectangle In Histogram

/\*\*

\* Find the area of the maximum rectangle in the histogram.

\* Example:

\* {2, 1, 3, 4, 5, 2, 6} --> 2 \* 5 = 10

\*/

public class MaxRectangle {

public int maxRectangle(int[] array) {

assert array != null && array.length != 0;

int maxArea = 0;

Deque<Integer> stack = new LinkedList<Integer>();

for (int i = 0; i <= array.length; i++) {

int cur = i == array.length ? 0 : array[i];

while (!stack.isEmpty() && array[stack.peekFirst()] >= cur) {

int height = array[stack.pollFirst()];

int left = stack.isEmpty() ? 0 : stack.peek() + 1;

maxArea = Math.max(maxArea, height \* (i - left));

}

stack.offerFirst(i);

}

return maxArea;

}

public static void main(String[] args) {

MaxRectangle solution = new MaxRectangle();

int[] array = { 1 };

System.out.println(solution.maxRectangle(array));

array = new int[] { 2, 1, 3, 4, 5, 2, 6 };

System.out.println(solution.maxRectangle(array));

array = new int[] { 2, 1, 4, 4, 4, 2, 6 };

System.out.println(solution.maxRectangle(array));

}

}

#### 7. Max Water Trapped In Histogram

/\*\*

\* Given an array of non-negative integers representing an histogram

\* where the width of each element is 1, compute how much water

\* it is able to trap after raining.

\* Example:

\* {4, 1, 3, 4, 5, 2, 6} --> 3 + 1 + 3 = 7

\*/

public class TrapWater {

public int trap(int[] array) {

assert array != null && array.length > 0;

int left = 0;

int right = array.length - 1;

int leftMax = array[0];

int rightMax = array[array.length - 1];

int result = 0;

while (left < right) {

if (array[left] <= array[right]) {

result += Math.max(0, leftMax - array[left]);

leftMax = Math.max(leftMax, array[left]);

left++;

} else {

result += Math.max(0, rightMax - array[right]);

rightMax = Math.max(rightMax, array[right]);

right--;

}

}

return result;

}

public static void main(String[] args) {

TrapWater solution = new TrapWater();

int[] array = { 1 };

System.out.println(solution.trap(array));

array = new int[] { 1, 2, 3 };

System.out.println(solution.trap(array));

array = new int[] { 4, 1, 3, 4, 5, 2, 6 };

System.out.println(solution.trap(array));

}

}

#### 8. Sky Line

\*\*

\* Given n houses on the ground with each house represented by a rectangle.

\* The i-th rectangle is represented as [start\_i, end\_i, height\_i],

\* where 0 <= i < n. The rectangles may overlap with each other.

\* How can we calculate the total area that these rectangles cover.

\* Example: input = {<1,3,1>, <2,4,2>}, output = 5.

\*/

public class SkyLine {

static class Building {

int start;

int end;

int height;

public Building(int start, int end, int height) {

this.start = start;

this.end = end;

this.height = height;

}

}

static class StartComparator implements Comparator<Building> {

@Override

public int compare(Building b1, Building b2) {

if (b1.start == b2.start) {

return 0;

}

return b1.start < b2.start ? -1 : 1;

}

}

static class HeightComparator implements Comparator<Building> {

@Override

public int compare(Building b1, Building b2) {

if (b1.height == b2.height) {

return 0;

}

return b1.height > b2.height ? -1 : 1;

}

}

public int totalArea(Building[] buildings) {

// Assumptions: buildings is not null.

assert buildings != null;

if (buildings.length == 0) {

return 0;

}

// swipe line algorithm, first need to sort the buildings by start position.

Arrays.sort(buildings, new StartComparator());

// maintain a max heap based on building's height.

PriorityQueue<Building> maxHeap = new PriorityQueue<Building>(buildings.length,

new HeightComparator());

int area = 0;

// left means the start position of current contour.

int left = Integer.MIN\_VALUE;

Building fakeLast = new Building(Integer.MAX\_VALUE, Integer.MAX\_VALUE, 0);

// swipe the building one by one.

for (int i = 0; i <= buildings.length; i++) {

Building cur = i < buildings.length ? buildings[i] : fakeLast;

// poll all buildings having end position <= cur.start

while (!maxHeap.isEmpty() && maxHeap.peek().end <= cur.start) {

Building top = maxHeap.poll();

// update left if needed and calculate the area

area += Math.max(0, top.end - left) \* top.height;

left = Math.max(top.end, left);

}

// if max heap is empty, meaning we can start a new contour.

if (maxHeap.isEmpty()) {

left = cur.start;

} else if (maxHeap.peek().height < cur.height) {

// if cur.height is larger than the peek building's height,

// means the cur contour with the peek building's height can be ended

// and a new contour should start.

area += (cur.start - left) \* maxHeap.peek().height;

left = cur.start;

}

maxHeap.offer(cur);

}

return area;

}

public static void main(String[] args) {

SkyLine sol = new SkyLine();

Building[] buildings = new Building[] { new Building(1, 2, 3) };

System.out.println(sol.totalArea(buildings));

buildings = new Building[] { new Building(1, 3, 1), new Building(2, 4, 2) };

System.out.println(sol.totalArea(buildings));

buildings = new Building[] { new Building(7, 9, 1), new Building(6, 7, 1),

new Building(0, 3, 2), new Building(6, 8, 2), new Building(1, 4, 1), new Building(5, 9, 1),

new Building(7, 8, 3), new Building(2, 5, 3) };

System.out.println(sol.totalArea(buildings));

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 27 - 强化练习 6

#### 1. Kth Smallest In Two Sorted Arrays

public class KthSmallestTwoSortedArray {

public int kth(int[] a, int[] b, int k) {

// a, b is not null, k <= a.length + b.length, k >=1, a, b not all empty

return kth(a, 0, b, 0, k);

}

private int kth(int[] a, int aleft, int[] b, int bleft, int k) {

if (aleft >= a.length) {

return b[bleft + k - 1];

}

if (bleft >= b.length) {

return a[aleft + k - 1];

}

if (k == 1) {

return Math.min(a[aleft], b[bleft]);

}

int amid = aleft + k / 2 - 1;

int bmid = bleft + k / 2 - 1;

int aval = amid >= a.length ? Integer.MAX\_VALUE : a[amid];

int bval = bmid >= b.length ? Integer.MAX\_VALUE : b[bmid];

if (aval <= bval) {

return kth(a, amid + 1, b, bleft, k - k/2);

} else {

return kth(a, aleft, b, bmid + 1, k - k/2);

}

}

}

#### 2. Hexadecimal Representation

public class HexRepresentation {

// Assumptions: number >= 0

public String hex(int number) {

assert number >= 0;

String prefix = "0x";

if (number == 0) {

return prefix + "0";

}

StringBuilder sb = new StringBuilder();

while (number > 0) {

int rem = number % 16;

if (rem < 10) {

sb.append((char) ('0' + rem));

} else {

sb.append((char) (rem - 10 + 'A'));

}

number >>>= 4;

}

return prefix + sb.reverse().toString();

}

}

#### 3. Max Values Of Sliding Windows

public class MaxValuesSlidingWindows {

public List<Integer> maxWindows(int[] array, int k) {

// array is not null not empty, k >= 1 and <= a.length

List<Integer> max = new ArrayList<Integer>();

Deque<Integer> deque = new LinkedList<Integer>();

for (int i = 0; i < array.length; i++) {

while (!deque.isEmpty() && array[deque.peekLast()] <= array[i]) {

deque.pollLast();

}

while (!deque.isEmpty() && deque.peekFirst() <= i - k) {

deque.pollFirst();

}

deque.offerLast(i);

if (i >= k - 1) {

max.add(array[deque.peekFirst()]);

}

}

return max;

}

}

#### 4. LRU Cache

public class LRUCache<K, V> {

static class Node<K, V> {

Node<K, V> next;

Node<K, V> prev;

K key;

V value;

Node(K key, V value) {

this.key = key;

this.value = value;

}

void update(K key, V value) {

this.key = key;

this.value = value;

}

}

private int size;

private int limit;

private Node<K, V> head;

private Node<K, V> tail;

private Map<K, Node<K, V>> map;

public LRUCache(int limit) {

this.limit = limit;

this.size = 0;

this.map = new HashMap<K, Node<K, V>>();

}

public void set(K key, V value) {

Node<K, V> node = null;

if (map.containsKey(key)) {

node = map.get(key);

node.value = value;

remove(node);

} else if (size < limit) {

node = new Node<K, V>(key, value);

} else {

node = tail;

remove(node);

node.update(key, value);

}

append(node);

}

public V get(K key) {

if (!map.containsKey(key)) {

return null;

}

Node<K, V> node = map.get(key);

remove(node);

append(node);

return node.value;

}

private Node<K, V> remove(Node<K, V> node) {

map.remove(node.key);

size--;

if (node.prev != null) {

node.prev.next = node.next;

}

if (node.next != null) {

node.next.prev = node.prev;

}

if (node == head) {

head = head.next;

}

if (node == tail) {

tail = tail.prev;

}

node.next = node.prev = null;

return node;

}

private Node<K, V> append(Node<K, V> node) {

map.put(node.key, node);

size++;

if (head == null) {

head = tail = node;

} else {

node.next = head;

head.prev = node;

head = node;

}

return node;

}

}

#### 5. First Non-Repeating Character In Stream

public class FirstNonRepeatingChar {

static class Node {

Character ch;

Node prev;

Node next;

Node(Character ch) {

this.ch = ch;

}

}

private HashMap<Character, Node> NonRepeated;

private HashSet<Character> Repeated;

private Node head;

private Node tail;

public FirstNonRepeatingChar() {

NonRepeated = new HashMap<Character, Node>();

Repeated = new HashSet<Character>();

head = tail = new Node(null);

}

public void read(char ch) {

if (Repeated.contains(ch)) {

return;

}

if (NonRepeated.containsKey(ch)) {

remove(NonRepeated.remove(ch));

Repeated.add(ch);

} else {

Node node = new Node(ch);

append(node);

NonRepeated.put(ch, node);

}

}

public Character firstNonRepeating() {

if (head == tail) {

return null;

}

return head.next.ch;

}

private void remove(Node node) {

if (node.prev != null) {

node.prev.next = node.next;

}

if (node.next != null) {

node.next.prev = node.prev;

}

if (node == tail) {

tail = node.prev;

}

node.next = node.prev = null;

}

private void append(Node node) {

tail.next = node;

node.prev = tail;

tail = node;

}

public static void main(String[] args) {

FirstNonRepeatingChar sol = new FirstNonRepeatingChar();

for (char ch : new char[] { 'a', 'b', 'c', 'a', 'c', 'b', 'd', 'e' }) {

sol.read(ch);

System.out.println(sol.firstNonRepeating());

}

}

}

#### 6. Majority Number

public class MajorityNumberI {

public int majority(int[] array) {

// array is not null and is not empty. majority number guarantees to exist.

int cur = array[0];

int count = 1;

for (int i = 1; i < array.length; i++) {

if (count == 0) {

count++;

cur = array[i];

} else if (cur == array[i]) {

count++;

} else {

count--;

}

}

return cur;

}

}

#### 7. Interleave Array

public class InterleaveArrays {

public boolean canMerge(char[] a, char[] b, char[] c) {

// a, b, c is not null,

int alen = a.length;

int blen = b.length;

int clen = c.length;

if (alen + blen != clen) {

return false;

}

boolean[][] canMerge = new boolean[alen + 1][blen + 1];

for (int i = 0; i <= alen; i++) {

for (int j = 0; j <= blen; j++) {

if (i == 0 && j == 0) {

canMerge[i][j] = true;

}

if (i > 0 && a[i - 1] == c[i + j - 1]) {

canMerge[i][j] = canMerge[i][j] || canMerge[i - 1][j];

}

if (j > 0 && b[j - 1] == c[i + j - 1]) {

canMerge[i][j] = canMerge[i][j] || canMerge[i][j - 1];

}

}

}

return canMerge[alen][blen];

}

}

[Back To Index](#h.jb8elqvr1fjc)

### Class 28 - 强化练习 7

#### 1. Reconstruct Binary Tree With Preorder And Inorder Traversal Sequence

public class PreOrderInOrder {

// Assumptions: no duplicates numbers in the tree.

public TreeNode fromPreInOrder(int[] preOrder, int[] inOrder) {

assert preOrder != null && inOrder != null;

assert preOrder.length == inOrder.length;

HashMap<Integer, Integer> inOrderMap = indexMap(inOrder);

return fromPreInOrder(preOrder, 0, preOrder.length - 1, inOrder, 0, inOrder.length - 1,

inOrderMap);

}

private TreeNode fromPreInOrder(int[] preOrder, int pLeft, int pRight, int[] inOrder, int iLeft,

int iRight, HashMap<Integer, Integer> inOrderMap) {

if (pLeft > pRight) {

return null;

}

TreeNode root = new TreeNode(preOrder[pLeft]);

int iIndex = inOrderMap.get(root.key);

int leftLength = iIndex - iLeft;

int rightLength = iRight - iIndex;

root.left = fromPreInOrder(preOrder, pLeft + 1, pLeft + leftLength, inOrder, iLeft, iIndex - 1,

inOrderMap);

root.right = fromPreInOrder(preOrder, pRight - rightLength + 1, pRight, inOrder, iIndex + 1,

iRight, inOrderMap);

return root;

}

private HashMap<Integer, Integer> indexMap(int[] inOrder) {

HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

for (int i = 0; i < inOrder.length; i++) {

map.put(inOrder[i], i);

}

return map;

}

}

#### 2. Reconstruct Binary Tree With Postorder And Inorder Traversal Sequence

public class ReconstructBTInPost {

private int pIndex;

private int iIndex;

// Assumptions: no duplicate numbers in the tree;

public TreeNode reconstruct(int[] post, int[] in) {

pIndex = iIndex = post.length - 1;

return helper(post, in, Integer.MIN\_VALUE);

}

private TreeNode helper(int[] post, int[] in, int target) {

if (iIndex < 0 || in[iIndex] == target) {

return null;

}

TreeNode root = new TreeNode(post[pIndex]);

pIndex--;

root.right = helper(post, in, root.key);

iIndex--;

root.left = helper(post, in, target);

return root;

}

}

#### 3. Reconstruct Binary Tree With Levelorder And Inorder Traversal Sequence

public class ReconstructBTInLevel {

public TreeNode reconstruct(int[] level, int[] in) {

// level, in not null. no duplicates.

Map<Integer, Integer> inMap = new HashMap<Integer, Integer>();

for (int i = 0; i < in.length; i++) {

inMap.put(in[i], i);

}

List<Integer> lList = new ArrayList<Integer>();

for (int num : level) {

lList.add(num);

}

return helper(lList, inMap);

}

private TreeNode helper(List<Integer> level, Map<Integer, Integer> inMap) {

if (level.isEmpty()) {

return null;

}

TreeNode root = new TreeNode(level.remove(0));

List<Integer> left = new ArrayList<Integer>();

List<Integer> right = new ArrayList<Integer>();

for (int num : level) {

if (inMap.get(num) < inMap.get(root.key)) {

left.add(num);

} else {

right.add(num);

}

}

root.left = helper(left, inMap);

root.right = helper(right, inMap);

return root;

}

}

#### 4. Reconstruct Binary Search Tree With Preorder Traversal Sequence

public class ReconstructBSTPreorder {

public TreeNode reconstruct(int[] pre) {

// preorder is not null, no duplicates.

return helper(pre, 0, pre.length - 1);

}

private TreeNode helper(int[] pre, int left, int right) {

if (left > right) {

return null;

}

if (left == right) {

return new TreeNode(pre[left]);

}

TreeNode root = new TreeNode(pre[left]);

int mid = findMid(pre, left + 1, right, pre[left]);

root.left = helper(pre, left + 1, mid);

root.right = helper(pre, mid + 1, right);

return root;

}

private int findMid(int[] pre, int left, int right, int target) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (pre[mid] > target) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return right;

}

}

#### 5. Reconstruct Binary Search Tree With Postorder Traversal Sequence

public class ReconstructBSTPostorder {

private int index;

public TreeNode reconstruct(int[] post) {

// postorder is not null, no duplicates

index = post.length - 1;

return helper(post, Integer.MIN\_VALUE);

}

private TreeNode helper(int[] postorder, int min) {

if (index < 0 || postorder[index] <= min) {

return null;

}

TreeNode root = new TreeNode(postorder[index--]);

root.right = helper(postorder, root.key);

root.left = helper(postorder, min);

return root;

}

}

#### 6. Reconstruct Binary Search Tree With Levelorder Traversal Sequence

public class ReconstructBSTLevelorder {

public TreeNode reconstruct(int[] level) {

// level order is not null. no duplicates.

List<Integer> levels = new ArrayList<Integer>();

for (int num : level) {

levels.add(num);

}

return helper(levels);

}

private TreeNode helper(List<Integer> level) {

if (level.isEmpty()) {

return null;

}

TreeNode root = new TreeNode(level.remove(0));

List<Integer> left = new ArrayList<Integer>();

List<Integer> right = new ArrayList<Integer>();

for (int num : level) {

if (num < root.key) {

left.add(num);

} else {

right.add(num);

}

}

root.left = helper(left);

root.right = helper(right);

return root;

}

}

#### 7. Most Points On A Line

public class MostPointsOnLine {

public int most(Point[] points) {

// sets is not null, and set.length >= 2

int result = 0;

for (int i = 0; i < points.length; i++) {

Point seed = points[i];

int same = 1;

int sameX = 0;

int most = 0;

HashMap<Double, Integer> cnt = new HashMap<Double, Integer>();

for (int j = 0; j < points.length; j++) {

if (i == j) {

continue;

}

Point tmp = points[j];

if (tmp.x == seed.x && tmp.y == seed.y) {

same++;

} else if (tmp.x == seed.x) {

sameX++;

} else {

double slope = ((tmp.y - seed.y) + 0.0) / (tmp.x - seed.x);

if (!cnt.containsKey(slope)) {

cnt.put(slope, 1);

} else {

cnt.put(slope, cnt.get(slope) + 1);

}

most = Math.max(most, cnt.get(slope));

}

}

most = Math.max(most, sameX) + same;

result = Math.max(result, most);

}

return result;

}

static class Point {

int x;

int y;

Point(int x, int y) {

this.x = x;

this.y = y;

}

@Override

public String toString() {

return "<" + x + "," + y + ">";

}

}

}

#### 8. Largest Set Of Points With Positive Slopes

public class LargestSetPointsPositiveSlope {

public int largest(Point[] points) {

// sets is not null

Arrays.sort(points, new MyComparator());

int result = 0;

int[] longest = new int[points.length];

for (int i = 0; i < longest.length; i++) {

for (int j = 0; j < i; j++) {

if (points[j].x < points[i].x && points[j].y < points[i].y) {

longest[i] = Math.max(longest[i], longest[j]);

}

}

longest[i]++;

result = Math.max(result, longest[i]);

}

return result == 1 ? 0 : result;

}

static class MyComparator implements Comparator<Point> {

@Override

public int compare(Point p1, Point p2) {

if (p1.y < p2.y) {

return -1;

} else if (p1.y > p2.y) {

return 1;

} else if (p1.x < p2.x) {

return -1;

} else if (p1.x > p2.x) {

return 1;

}

return 0;

}

}

static class Point {

int x;

int y;

Point(int x, int y) {

this.x = x;

this.y = y;

}

}

}

#### 9. Generate Random Maze

public class GenerateRandomMaze {

public int[][] maze(int n) {

// n = 2 \* k + 1, where k > = 0,

int[][] maze = new int[n][n];

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (i == 0 && j == 0) {

maze[i][j] = 0;

} else {

maze[i][j] = 1;

}

}

}

generate(maze, 0, 0);

return maze;

}

private void generate(int[][] maze, int x, int y) {

Dir[] dirs = Dir.values();

shuffle(dirs);

for (Dir dir : dirs) {

int nextX = dir.moveX(dir.moveX(x));

int nextY = dir.moveY(dir.moveY(y));

if (valid(maze, nextX, nextY)) {

maze[dir.moveX(x)][dir.moveY(y)] = 0;

maze[nextX][nextY] = 0;

generate(maze, nextX, nextY);

}

}

}

// Get a random order of the directions.

private void shuffle(Dir[] dirs) {

for (int i = 0; i < dirs.length; i++) {

int index = (int) (Math.random() \* (dirs.length - i));

Dir tmp = dirs[i];

dirs[i] = dirs[i + index];

dirs[i + index] = tmp;

}

}

private boolean valid(int[][] maze, int x, int y) {

return x >= 0 && x < maze.length && y >= 0 && y < maze[0].length && maze[x][y] == 1;

}

enum Dir {

NORTH(-1, 0), SOUTH(1, 0), EAST(0, -1), WEST(0, 1);

int deltaX;

int deltaY;

Dir(int deltaX, int deltaY) {

this.deltaX = deltaX;

this.deltaY = deltaY;

}

public int moveX(int x) {

return x + deltaX;

}

public int moveY(int y) {

return y + deltaY;

}

}

}

#### 10. Disjoint White Objects

public class DisjointWhiteObjects {

public int whiteObjects(int[][] matrix) {

// binary matrix is not null, M \* N, M >= 0, N >= 0

int rows = matrix.length;

if (rows == 0) {

return 0;

}

int cols = matrix[0].length;

if (cols == 0) {

return 0;

}

int count = 0;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

if (matrix[i][j] == 0) {

count++;

DFS(matrix, i, j);

}

}

}

return count;

}

private void DFS(int[][] matrix, int i, int j) {

matrix[i][j] = 1;

int rows = matrix.length;

int cols = matrix[0].length;

if (i > 0 && matrix[i - 1][j] == 0) {

DFS(matrix, i - 1, j);

}

if (i < rows - 1 && matrix[i + 1][j] == 0) {

DFS(matrix, i + 1, j);

}

if (j > 0 && matrix[i][j - 1] == 0) {

DFS(matrix, i, j - 1);

}

if (j < cols - 1 && matrix[i][j + 1] == 0) {

DFS(matrix, i, j + 1);

}

}

}