Index

Class 1 Array and Sorting Algorithms

Selection Sort

```
public class SelectionSort {
```

```java
public void selectionSort(int[] array) {
  // sanity check before the main logic is applied.
  // conditions to consider: null? empty? ......
  if (array == null) {
    return;
  }
  for (int i = 0; i < array.length - 1; i++) {
    int min = i;
    // find the min element in the subarray of (i, array.length - 1)
    for (int j = i + 1; j < array.length; j++) {
      if (array[j] < array[min]) {
        min = j;
      }
    }
    swap(array, i, min);
  }
}

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 = null;
  solution.selectionSort(array);
  System.out.println(Arrays.toString(array));

  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));
  }
}
```

Merge Sort

```
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);
  }

  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
    for (int i = left; i <= right; i++) {
      helper[i] = array[i];
    }
    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 still have some elements 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();

    // test cases to cover possible situations.
    int[] array = null;
    solution.mergeSort(array);
    System.out.println(Arrays.toString(array));

    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));
  }
}
```

Quick Sort

```
public class QuickSort {
  public void quickSort(int[] array) {
    if (array == null) {
      return;
    }
    quickSort(array, 0, array.length - 1);
  }
```

```java
public void quickSort(int[] array, int left, int right) {
  if (left >= right) {
    return;
  }
  // define a pivot and use the pivot to partition the array.
  int pivotPos = partition(array, left, right);
  quickSort(array, left, pivotPos - 1);
  quickSort(array, pivotPos + 1, right);
}

private int partition(int[] array, int left, int right) {
  int pivotIndex = pivotIndex(left, right);
  int pivot = array[pivotIndex];
  // swap the pivot element to the rightmost position first
  swap(array, pivotIndex, right);
  int leftBound = left;
  int rightBound = right - 1;
  while (leftBound <= rightBound) {
    if (array[leftBound] < pivot) {
      leftBound++;
    } else if (array[rightBound] >= pivot) {
        rightBound--;
    } else {
      swap(array, leftBound++, rightBound--);
    }
  }
  // swap back the pivot element.
  swap(array, leftBound, right);
  return leftBound;
}

private int pivotIndex(int left, int right) {
  // sample implementation, pick random element as pivot each time.
  return left + (int) (Math.random() * (right - left + 1));
}

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) {
```

```java
    QuickSort solution = new QuickSort();

    int[] array = null;
    solution.quickSort(array);
    System.out.println(Arrays.toString(array));

    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));
  }
}
```

Array Shuffling I (move all 0s to the right end)

```java
/**
 * Move 0 to the right end of the array, no need to keep the relative order of
 * the elements in the original array.
 */
public class MoveZeroII {
  public void moveZero(int[] array) {
    if (array == null || array.length <= 1) {
      return;
    }
    int left = 0;
    int right = array.length - 1;
    while (left <= right) {
      if (array[left] != 0) {
        left++;
      } else if (array[right] == 0) {
        right--;
      } else {
```

```
        swap(array, left++, right--);
      }
    }
  }

  private void swap(int[] array, int a, int b) {
    int tmp = array[a];
    array[a] = array[b];
    array[b] = tmp;
  }
}
```

Rainbow Sort

```
/**
 * Rainbow sort implementation.
 * Assumption:
 * 1).we have three colors denoted as -1, 0, 1 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 neg is -1.
    // 2. the right side of pos is 1.
    // 3. the part between neg and zero is 0.
    // 4. the part between zero and pos is to be discovered.
    int neg = 0;
    int zero = 0;
    int pos = array.length - 1;
    while (zero <= pos) {
      if (array[zero] == 0) {
        zero++;
      } else if (array[zero] == -1) {
        swap(array, neg++, zero++);
      } else {
        swap(array, zero, pos--);
      }
    }
  }
```

```
  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 = null;
    solution.rainbowSort(array);
    System.out.println(Arrays.toString(array));

    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[] { 0, 1, 1, -1 };
    solution.rainbowSort(array);
    System.out.println(Arrays.toString(array));

    array = new int[] { 1, -1, 0, 1, 0 };
    solution.rainbowSort(array);
    System.out.println(Arrays.toString(array));
  }
}
```

[Back To Index](#)

Class 2 Recursion I and Binary Search

Fibonacci Number

```
public class FibonacciNumber {
```

```java
  // Method 1: recursion, this method will timeout on laicode.com
  public long fibonacci(int K) {
    if (K <= 0) {
      return 0;
    }
    if (K == 1) {
      return 1;
    }
    return fibonacci(K - 1) + fibonacci(K - 2);
  }

  // Method 2: dp solution with O(n) space.
  public long fibonacciI(int K) {
    if (K <= 0) {
      return 0;
    }
    long[] array = new long[K + 1];
    array[1] = 1;
    for (int i = 2; i <= K; i++) {
      array[i] = array[i - 2] + array[i - 1];
    }
    return array[K];
  }

  // Method 3: dp solution with O(1) space.
  public long fibonacciII(int K) {
    long a = 0;
    long b = 1;
    if (K <= 0) {
      return a;
    }
    while (K > 1) {
      long temp = a + b;
      a = b;
      b = temp;
      K--;
    }
    return b;
  }
}
```

a To the Power of b

```java
public class Power {
  // Assumption: a >= 0, b >= 0.
  public long power(int a, int b) {
    if (b == 0) {
      return 1;
    }
    if (a == 0) {
      return b > 0 ? 0 : Long.MAX_VALUE;
    }
    long half = power(a, b / 2);
    return b % 2 == 0 ? half * half : half * half * a;
  }
}
```

Classical Binary Search

```java
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;
  }
}
```

Binary Search In Sorted 2D Array

```java
/**
 * Search in sorted matrix, each row of the matrix is sorted in ascending order,
 * and the first element of the row is equals to or larger than the last element
```

```java
 * of the previous row.
 *
 * Return the position (row, column) if the target is found, otherwise return null.
 */
public class SearchInSortedMatrixI {
  // Assumptions: matrix is not null. return null if not found.
  // Method 1: find row first then find col.
  public int[] searchI(int[][] matrix, int target) {
    if (matrix.length == 0 || matrix[0].length == 0) {
      return null;
    }
    int row = findRow(matrix, 0, matrix.length - 1, target);
    if (row == -1) {
      return null;
    }
    int col = findCol(matrix[row], 0, matrix[row].length - 1, target);
    if (col == -1) {
      return null;
    }
    return new int[] { row, col };
  }

  private int findRow(int[][] matrix, int up, int down, int target) {
    while (up <= down) {
      int mid = up + (down - up) / 2;
      if (matrix[mid][0] > target) {
        down = mid - 1;
      } else {
        up = mid + 1;
      }
    }
    return down;
  }

  private int findCol(int[] array, int left, int right, int target) {
    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;
  }

  // Method 2: convert the 2D array to 1D array and do binary search.
  public int[] searchII(int[][] matrix, int target) {
    if (matrix.length == 0 || matrix[0].length == 0) {
      return null;
    }
    int rows = matrix.length;
    int cols = matrix[0].length;
    int left = 0;
    int right = rows * cols - 1;
    while (left <= right) {
      int mid = left + (right - left) / 2;
      int row = mid / cols;
      int col = mid % cols;
      if (matrix[row][col] == target) {
        return new int[] { row, col };
      } else if (matrix[row][col] < target) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }
    return null;
  }
}
```

First Occurrence

```
public class FirstOccurrence {
  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;
  } else if (array[right] == target) {
    return right;
  }
  return -1;
 }
}
```

Last Occurrence

```
public class LastOccurrence {
 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;
  } else if (array[left] == target) {
    return left;
  }
  return -1;
 }
}
```

Closest Number In Sorted Array

```
public class Closest {
```

```
  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;
    }
  }
}
```

Search In Unknown Sized Sorted Array

```
/**
 * 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;
    int right = 1;
    while (dictionary.get(right) != null && dictionary.get(right) < target) {
      // 1. move left to right
      // 2. double right index
      left = right;
      right = 2 * 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;
  }
}
```

Back To Index

Class 3 Queue, Stack & Linked List

Implement Queue By Two Stacks

```java
public class QueueByTwoStack {
  private Deque<Integer> in;
  private Deque<Integer> out;

  public QueueByTwoStack() {
    in = new LinkedList<Integer>();
    out = new LinkedList<Integer>();
  }

  public Integer poll() {
    move();
    return out.isEmpty() ? null : out.pollFirst();
  }

  public void offer(int value) {
    in.offerFirst(value);
  }

  public Integer peek() {
    move();
    return out.isEmpty() ? null : out.peekFirst();
  }

  // when out stack is empty, move the elements from in.
  private void move() {
    if (out.isEmpty()) {
      while (!in.isEmpty()) {
        out.offerFirst(in.pollFirst());
      }
    }
  }

  public int size() {
    return in.size() + out.size();
  }

  public boolean isEmpty() {
    return in.size() == 0 && out.size() == 0;
```

```
  }
}
```

Stack With min()

```
public class StackWithMin {

  private Deque<Integer> stack;
  private Deque<Integer> minStack;

  public StackWithMin() {
    stack = new LinkedList<Integer>();
    minStack = new LinkedList<Integer>();
  }

  public Integer min() {
    if (minStack.isEmpty()) {
      return null;
    }
    return minStack.peekFirst();
  }

  public void push(int value) {
    stack.offerFirst(value);
    // when value <= current min value in stack,
    // need to push the value to minStack.
    if (minStack.isEmpty() || value <= minStack.peekFirst()) {
      minStack.offerFirst(value);
    }
  }

  public Integer pop() {
    if (stack.isEmpty()) {
      return null;
    }
    Integer result = stack.pollFirst();
    // when the popped value is the same as top value of minStack, the value
    // need to be popped from minStack as well.
    if (minStack.peekFirst().equals(result)) {
      minStack.pollFirst();
    }
    return result;
  }
```

```java
  public Integer top() {
    if (stack.isEmpty()) {
      return null;
    }
    return stack.peekFirst();
  }
}
```

Sort Numbers In Three Stacks

```java
/**
 * The numbers are in s1 originally, after sorting, the numbers should be in s1
 * as well and from top to bottom the numbers are sorted in ascending order.
 */
public class SortArrayThreeStacks {
  // Assumptions: s1 is not null.
  public void sort(LinkedList<Integer> s1) {
    LinkedList<Integer> s2 = new LinkedList<Integer>();
    LinkedList<Integer> s3 = new LinkedList<Integer>();
    sort(s1, s2, s3, s1.size());
  }

  private void sort(LinkedList<Integer> s1, LinkedList<Integer> s2, LinkedList<Integer> s3,
      int length) {
    if (length <= 1) {
      return;
    }
    int mid1 = length / 2;
    int mid2 = length - length / 2;
    for (int i = 0; i < mid1; i++) {
      s2.offerFirst(s1.pollFirst());
    }
    // use the other stacks to sort s2/s1.
    // after sorting the numbers in s2/s1 are in ascending order from top to
    // bottom in the two stacks.
    sort(s2, s3, s1, mid1);
    sort(s1, s3, s2, mid2);
    int i = 0;
    int j = 0;
    while (i < mid1 && j < mid2) {
      if (s2.peekFirst() < s1.peekFirst()) {
        s3.offerFirst(s2.pollFirst());
```

```
        i++;
      } else {
        s3.offerFirst(s1.pollFirst());
        j++;
      }
    }
    while (i < mid1) {
      s3.offerFirst(s2.pollFirst());
      i++;
    }
    while (j < mid2) {
      s3.offerFirst(s1.pollFirst());
      j++;
    }
    // after merging, the numbers are in descending order from top to bottom in
    // s3, we need to push them back to s1 so that they are in ascending order.
    for (int index = 0; index < length; index++) {
      s1.offerFirst(s3.pollFirst());
    }
  }
}
```

Reverse Linked List

```
/**
 * Reverse singled linked list, both iterative and recursive ways.
 */
public class ReverseLinkedList {
  // Method 1: iteratively reverse linked list.
  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;
  }
```

```java
  // Method 2: recursively reverse linked list.
  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;
  }
}
```

Find Middle Node Of Linked List

```java
/**
 * Find the middle node of a singled linked list.
 * 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;
  }
}
```

Check If Linked List Has Cycle

```java
public class CheckCycle {
  public boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) {
      return false;
    }
```

```
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
      if (slow == fast) {
        return true;
      }
    }
    return false;
  }
}
```

Insert In Sorted Linked List

```
public class InsertSortedList {
  public ListNode insert(ListNode head, int value) {
    ListNode newNode = new ListNode(value);
    // 1. determine if the inserted node is before head.
    if (head == null || head.value >= value) {
      newNode.next = head;
      return newNode;
    }
    // 2. insert the new node to the right position.
    // using the previous node to traverse the linked list
    // the insert position of the new node should be between prev and prev.next
    ListNode prev = head;
    while (prev.next != null && prev.next.value < value) {
      prev = prev.next;
    }
    newNode.next = prev.next;
    prev.next = newNode;
    return head;
  }
}
```

Merge Two Sorted Linked List

```
public class MergeTwoSortedList {
  public ListNode merge(ListNode one, ListNode two) {
    ListNode dummy = new ListNode(0);
    ListNode cur = dummy;
    while (one != null && two != null) {
```

```
      if (one.value <= two.value) {
        cur.next = one;
        one = one.next;
      } else {
        cur.next = two;
        two = two.next;
      }
      cur = cur.next;
    }
    if (one != null) {
      cur.next = one;
    } else {
      cur.next = two;
    }
    return dummy.next;
  }
}
```

Reorder Linked List

```
public class ReorderList {
  public ListNode reorder(ListNode head) {
    if (head == null || head.next == null) {
      return head;
    }
    // 1. find the middle node
    ListNode mid = middleNode(head);
    ListNode one = head;
    ListNode two = mid.next;
    // de-link the second half from the list.
    mid.next = null;
    // 2. reverse the second half
    // 3. merge the two halves
    return merge(one, reverse(two));
  }

  private ListNode middleNode(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    while (fast.next != null && fast.next.next != null) {
      slow = slow.next;
      fast = fast.next.next;
    }
```

```
    return slow;
  }

  private ListNode reverse(ListNode head) {
    if (head == null || head.next == null) {
      return head;
    }
    ListNode prev = null;
    while (head != null) {
      ListNode next = head.next;
      head.next = prev;
      prev = head;
      head = next;
    }
    return prev;
  }

  private ListNode merge(ListNode one, ListNode two) {
    ListNode dummy = new ListNode(0);
    ListNode cur = dummy;
    while (one != null && two != null) {
      cur.next = one;
      one = one.next;
      cur.next.next = two;
      two = two.next;
      cur = cur.next.next;
    }
    if (one != null) {
      cur.next = one;
    } else {
      cur.next = two;
    }
    return dummy.next;
  }
}
```

Partition Linked List

```
public class PartitionList {
  public ListNode partition(ListNode head, int target) {
    if (head == null || head.next == null) {
      return head;
    }
```

```
    ListNode small = new ListNode(0);
    ListNode large = new ListNode(0);
    ListNode curSmall = small;
    ListNode curLarge = large;
    while (head != null) {
      if (head.value < target) {
        curSmall.next = head;
        curSmall = curSmall.next;
      } else {
        curLarge.next = head;
        curLarge = curLarge.next;
      }
      head = head.next;
    }
    // connect the two partitions
    curSmall.next = large.next;
    // un-link the last node in large partition
    curLarge.next = null;
    return small.next;
  }
}
```

[Back To Index]

Class 4 - Binary Tree & Binary Search Tree

Binary Tree Pre-order Iterative Traversal

```
public class PreOrder {
  public List<Integer> preOrder(TreeNode root) {
    List<Integer> preorder = new ArrayList<Integer>();
    if (root == null) {
      return preorder;
    }
    Deque<TreeNode> stack = new LinkedList<TreeNode>();
    stack.offerFirst(root);
    while (!stack.isEmpty()) {
      TreeNode cur = stack.pollFirst();
      // the left subtree should be traversed before right subtree,
      // since stack is LIFO, we should push right into the stack first,
      // so for the next step the top element of the stack is the left subtree.
      if (cur.right != null) {
        stack.offerFirst(cur.right);
```

```
      }
      if (cur.left != null) {
        stack.offerFirst(cur.left);
      }
      preorder.add(cur.key);
    }
    return preorder;
  }
}
```

Binary Tree In-order Iterative Traversal

```
public class InOrder {
  public List<Integer> inOrder(TreeNode root) {
    List<Integer> inorder = new ArrayList<Integer>();
    Deque<TreeNode> stack = new LinkedList<TreeNode>();
    TreeNode cur = root;
    while (cur != null || !stack.isEmpty()) {
      // always go left if applicable.
      if (cur != null) {
        stack.offerFirst(cur);
        cur = cur.left;
      } else {
        // if can not go left, should pop top element and go right.
        cur = stack.pollFirst();
        inorder.add(cur.key);
        cur = cur.right;
      }
    }
    return inorder;
  }
}
```

Binary Tree Post-order Iterative Traversal

```
public class PostOrder {
  // Method 1: post-order is the reverse order of pre-order with traversing
  // right subtree first then left subtree.
  public List<Integer> postOrderI(TreeNode root) {
    List<Integer> result = new ArrayList<Integer>();
    if (root == null) {
      return result;
    }
```

```java
  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;
}

// Method 2: check the relation between the current node and the previous node
// to determine which direction should go next.
public List<Integer> postOrderII(TreeNode root) {
  List<Integer> list = new ArrayList<Integer>();
  if (root == null) {
    return list;
  }
  Deque<TreeNode> stack = new LinkedList<TreeNode>();
  stack.offerFirst(root);
  TreeNode prev = null;
  while (!stack.isEmpty()) {
    TreeNode cur = stack.peekFirst();
    if (prev == null || cur == prev.left || cur == prev.right) {
      if (cur.left != null) {
        stack.offerFirst(cur.left);
      } else if (cur.right != null) {
        stack.offerFirst(cur.right);
      } else {
        stack.pollFirst();
        list.add(cur.key);
      }
    } else if (prev == cur.right || prev == cur.left && cur.right == null) {
      stack.pollFirst();
      list.add(cur.key);
```

```
      } else {
        stack.offerFirst(cur.right);
      }
      prev = cur;
    }
    return list;
  }
}
```

Check If Binary Tree Is Balanced

```
public class CheckBalanced {
  public boolean isBalanced(TreeNode root) {
    if (root == null) {
      return true;
    }
    // use -1 to denote the tree is not balanced.
    // >= 0 value means the tree is balanced and the value is the height of the
    // tree.
    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;
  }
}
```

Check If Binary Tree Is Symmetric

```
public class CheckSymmetric {
  public boolean isSymmetric(TreeNode root) {
```

```
    if (root == null) {
      return true;
    }
    return isSymmetric(root.left, root.right);
  }

  private boolean isSymmetric(TreeNode one, TreeNode two) {
    if (one == null && two == null) {
      return true;
    } else if (one == null || two == null) {
      return false;
    } else if (one.key != two.key) {
      return false;
    } else {
      return isSymmetric(one.left, two.right)
          && isSymmetric(one.right, two.left);
    }
  }
}
```

Tweaked Identical Binary Trees

```
public class CheckTweakedIdentical {
  public boolean isTweakedIdentical(TreeNode one, TreeNode two) {
    if (one == null && two == null) {
      return true;
    } else if (one == null || two == null) {
      return false;
    } else if (one.key != two.key) {
      return false;
    } else {
      return isTweakedIdentical(one.left, two.left) && isTweakedIdentical(one.right, two.right)
          || isTweakedIdentical(one.left, two.right) && isTweakedIdentical(one.right, two.left);
    }
  }
}
```

Is Binary Search Tree Or Not

```
public class CheckBST {
  public boolean isBST(TreeNode root) {
    return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
  }
```

```java
  private boolean isBST(TreeNode root, int min, int max) {
    if (root == null) {
      return true;
    }
    if (root.key < min || root.key > max) {
      return false;
    }
    return isBST(root.left, min, root.key - 1)
        && isBST(root.right, root.key + 1, max);
  }
}
```

Get Keys In Binary Search Tree In Given Range

```java
public class GetRange {

  public List<Integer> getRange(TreeNode root, int min, int max) {
    List<Integer> list = new ArrayList<Integer>();
    getRange(root, min, max, list);
    return list;
  }

  private void getRange(TreeNode root, int min, int max, List<Integer> list) {
    if (root == null) {
      return;
    }
    // 1. determine if left subtree should be traversed, only when root.key >
    // min, we should traverse the left subtree.
    if (root.key > min) {
      getRange(root.left, min, max, list);
    }
    // 2. determine if root should be traversed.
    if (root.key >= min && root.key <= max) {
      list.add(root.key);
    }
    // 3. determine if right subtree should be traversed, only when root.key <
    // max, we should traverse the right subtree.
    if (root.key < max) {
      getRange(root.right, min, max, list);
    }
  }
}
```

[Back To Index](#)

Class 5 Heap & Graph Search Algorithm I (BFS)

K Smallest In Unsorted Array

```java
/**
 * Find the smallest k elements in an unsorted array. Assumptions: 1). array is
 * not null 2). k >= 0 and k <= array.length
 */
public class KSmallest {
  // Method 1: K sized max heap
  public int[] kSmallestI(int[] array, int k) {
    assert array != null;
    if (array.length == 0 || k == 0) {
      return new int[0];
    }
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k,
        new Comparator<Integer>() {
          public int compare(Integer o1, Integer o2) {
            if (o1.equals(o2)) {
              return 0;
            }
            return o1 > o2 ? -1 : 1;
          }
        });
    for (int i = 0; i < array.length; i++) {
      if (i < k) {
        // offer the first k elements into max heap.
        maxHeap.offer(array[i]);
      } else if (array[i] < maxHeap.peek()) {
        // for the other elements, only offer it into the max heap if it is
        // smaller than the max value in the max heap.
        maxHeap.poll();
        maxHeap.offer(array[i]);
      }
    }
    int[] result = new int[k];
    for (int i = k - 1; i >= 0; i--) {
      result[i] = maxHeap.poll();
    }
    return result;
  }
```

```java
// Method 2: quick select
public int[] kSmallestII(int[] array, int k) {
  if (array.length == 0 || k == 0) {
    return new int[0];
  }
  quickSelect(array, 0, array.length - 1, k - 1);
  int[] result = Arrays.copyOf(array, k);
  Arrays.sort(result);
  return result;
}

private void quickSelect(int[] array, int left, int right, int target) {
  int mid = partition(array, left, right);
  if (mid == target) {
    return;
  } else if (target < mid) {
    quickSelect(array, left, mid - 1, target);
  } else {
    quickSelect(array, mid + 1, right, target);
  }
}

private int partition(int[] array, int left, int right) {
  int pivot = array[right];
  int start = left;
  int end = right - 1;
  while (start <= end) {
    if (array[start] < pivot) {
      start++;
    } else if (array[end] >= pivot) {
      end--;
    } else {
      swap(array, start++, end--);
    }
  }
  swap(array, start, right);
  return start;
}

private void swap(int[] array, int a, int b) {
  int tmp = array[a];
  array[a] = array[b];
```

```
    array[b] = tmp;
  }
}
```

Get Keys In Binary Tree Layer By Layer

```
public class LayerByLayer {
  public List<List<Integer>> layerByLayer(TreeNode root) {
    List<List<Integer>> list = new ArrayList<List<Integer>>();
    if (root == null) {
      return list;
    }
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    while (!queue.isEmpty()) {
      List<Integer> curLayer = new ArrayList<Integer>();
      int size = queue.size();
      for (int i = 0; i < size; i++) {
        TreeNode cur = queue.poll();
        curLayer.add(cur.key);
        if (cur.left != null) {
          queue.offer(cur.left);
        }
        if (cur.right != null) {
          queue.offer(cur.right);
        }
      }
      list.add(curLayer);
    }
    return list;
  }
}
```

Bipartite

```
/**
 * Assumptions:
 * 1). The given graph is not null.
 */
public class Bipartite {
  public boolean isBipartite(List<GraphNode> graph) {
    assert graph != null;
    // use 0 and 1 to denote two different groups.
    // the map maintains for each node which group it belongs to.
```

```java
    HashMap<GraphNode, Integer> map = new HashMap<GraphNode, Integer>();
    // the graph can be represented by a list of nodes (if it is not guaranteed
    // to be connected). we have to do BFS from each of the nodes.
    for (GraphNode node : graph) {
      if (!BFS(node, map)) {
        return false;
      }
    }
    return true;
  }

  private boolean BFS(GraphNode node, HashMap<GraphNode, Integer> map) {
    // if this node has been traversed, no need to do BFS again.
    if (map.containsKey(node)) {
      return true;
    }
    Queue<GraphNode> queue = new LinkedList<GraphNode>();
    queue.offer(node);
    // we can assign it to any of the groups, for example, group 0.
    map.put(node, 0);
    while (!queue.isEmpty()) {
      GraphNode cur = queue.poll();
      int curSign = map.get(cur);
      int neiSign = curSign == 0 ? 1 : 0;
      for (GraphNode nei : cur.neighbors) {
        // if the neighbor has not been traversed, just put it in the queue
        // and choose the correct group.
        if (!map.containsKey(nei)) {
          map.put(nei, neiSign);
          queue.offer(nei);
        } else if (map.get(nei) != neiSign) {
          // only if the neighbor has been traversed and the group does not
          // match to the desired one, return false.
          return false;
        }
      }
    }
    return true;
  }
}
```

Check If Binary Tree Is Complete

```
public class CheckCompleted {
  public boolean isCompleted(TreeNode root) {
    if (root == null) {
      return true;
    }
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    // if the flag is set true, there should not be any child nodes afterwards.
    boolean flag = false;
    queue.offer(root);
    while (!queue.isEmpty()) {
      TreeNode cur = queue.poll();
      if (cur.left == null) {
        flag = true;
      } else if (flag) {
        return false;
      } else {
        queue.offer(cur.left);
      }
      if (cur.right == null) {
        flag = true;
      } else if (flag) {
        return false;
      } else {
        queue.offer(cur.right);
      }
    }
    return true;
  }
}
```

Kth Smallest Number In Sorted Matrix

```
/**
 * Assumptions:
 * 1). matrix is not null, N * M where N > 0 and M > 0
 * 2). k > 0 and k <= N * M
 */
public class KthSmallestInSortedMatrixII {
  public int kthSmallest(int[][] matrix, int k) {
    int len = matrix.length;
    int clen = matrix[0].length;
```

```java
    PriorityQueue<Sum> minHeap = new PriorityQueue<Sum>(k, new Comparator<Sum>() {
      @Override
      public int compare(Sum s1, Sum s2) {
        if (s1.sum == s2.sum) {
          return 0;
        }
        return s1.sum < s2.sum ? -1 : 1;
      }
    });
    // mark all the cells been visited.
    boolean[][] visited = new boolean[len][clen];
    minHeap.offer(new Sum(0, 0, matrix[0][0]));
    visited[0][0] = true;
    int result = Integer.MIN_VALUE;
    for (int i = 0; i < k; i++) {
      Sum cur = minHeap.poll();
      result = cur.sum;
      if (cur.x + 1 < len && !visited[cur.x + 1][cur.y]) {
        minHeap.offer(new Sum(cur.x + 1, cur.y, matrix[cur.x + 1][cur.y]));
        visited[cur.x + 1][cur.y] = true;
      }
      if (cur.y + 1 < clen && !visited[cur.x][cur.y + 1]) {
        minHeap.offer(new Sum(cur.x, cur.y + 1, matrix[cur.x][cur.y + 1]));
        visited[cur.x][cur.y + 1] = true;
      }
    }
    return result;
  }

static class Sum {
    int x;
    int y;
    int sum;

    Sum(int x, int y, int sum) {
      this.x = x;
      this.y = y;
      this.sum = sum;
    }
  }
}
```

Back To Index

Class 6 Graph Search Algorithm II (DFS)

All Subsets I

```
/**
 * Assumptions:
 * 1). there are no duplicate characters in the given string.
 */
public class SubSetsI {
  // method 1: DFS solution.
  public List<String> subSets(String set) {
    List<String> result = new ArrayList<String>();
    if (set == null) {
      return result;
    }
    char[] arraySet = set.toCharArray();
    StringBuilder sb = new StringBuilder();
    helper(arraySet, sb, 0, result);
    return result;
  }

  private void helper(char[] set, StringBuilder sb, int index, List<String> result) {
    if (index == set.length) {
      result.add(sb.toString());
      return;
    }
    helper(set, sb, index + 1, result);
    helper(set, sb.append(set[index]), index + 1, result);
    sb.deleteCharAt(sb.length() - 1);
  }

  // method 2: another DFS solution.
  public List<String> subSetsII(String set) {
    List<String> result = new ArrayList<String>();
    if (set == null) {
      return result;
    }
    char[] arraySet = set.toCharArray();
    StringBuilder sb = new StringBuilder();
    helperII(arraySet, sb, 0, result);
    return result;
  }
```

```
  private void helperII(char[] set, StringBuilder sb, int index, List<String> result) {
    result.add(sb.toString());
    // choose what is the index in the original set to pick.
    // maintain the ascending order of the picked indices.
    for (int i = index; i < set.length; i++) {
      sb.append(set[i]);
      helperII(set, sb, i + 1, result);
      sb.deleteCharAt(sb.length() - 1);
    }
  }
}
```

All Valid Permutations Of Parentheses I

```
public class ValidParenthesesI {
  public List<String> validParentheses(int k) {
    List<String> result = new ArrayList<String>();
    char[] cur = new char[k * 2];
    helper(cur, k, k, 0, result);
    return result;
  }

  private void helper(char[] cur, int left, int right, int index, List<String> result) {
    if (left == 0 && right == 0) {
      result.add(new String(cur));
      return;
    }
    if (left > 0) {
      cur[index] = '(';
      helper(cur, left - 1, right, index + 1, result);
    }
    if (right > left) {
      cur[index] = ')';
      helper(cur, left, right - 1, index + 1, result);
    }
  }
}
```

Combinations Of Coins

```
public class CombinationsOfCoins {
  public List<List<Integer>> combinations(int target, int[] coins) {
```

```
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> cur = new ArrayList<Integer>();
    helper(target, coins, 0, cur, result);
    return result;
  }

  private void helper(int target, int[] coins, int index, List<Integer> cur,
      List<List<Integer>> result) {
    // terminate earlier to avoid too many branches.
    if (index == coins.length - 1) {
      if (target % coins[coins.length - 1] == 0) {
        cur.add(target / coins[coins.length - 1]);
        result.add(new ArrayList<Integer>(cur));
        cur.remove(cur.size() - 1);
      }
      return;
    }
    int max = target / coins[index];
    for (int i = 0; i <= max; i++) {
      cur.add(i);
      helper(target - i * coins[index], coins, index + 1, cur, result);
      cur.remove(cur.size() - 1);
    }
  }
}
```

All Permutations I

```
public class PermutationsI {
  // 1. DFS solution with swapping
  public List<String> permutations(String set) {
    List<String> result = new ArrayList<String>();
    if (set == null) {
      return result;
    }
    char[] array = set.toCharArray();
    helper(array, 0, result);
    return result;
  }

  private void helper(char[] array, int index, List<String> result) {
    if (index == array.length) {
      result.add(new String(array));
```

```
    return;
  }
  for (int i = index; i < array.length; i++) {
    swap(array, index, i);
    helper(array, index + 1, result);
    swap(array, index, i);
  }
}

private void swap(char[] array, int left, int right) {
  char tmp = array[left];
  array[left] = array[right];
  array[right] = tmp;
}

// 2. Solution to maintain the order of all the permutations.
public List<String> permutationsWithOrder(String set) {
  List<String> result = new ArrayList<String>();
  if (set == null) {
    return result;
  }
  char[] arraySet = set.toCharArray();
  Arrays.sort(arraySet);
  // record which index has been used.
  boolean[] used = new boolean[arraySet.length];
  StringBuilder cur = new StringBuilder();
  helperWithOrder(arraySet, used, cur, result);
  return result;
}

private void helperWithOrder(char[] array, boolean[] used, StringBuilder cur, List<String>
result) {
  if (cur.length() == array.length) {
    result.add(cur.toString());
    return;
  }
  // when picking the next char, always according to the order.
  for (int i = 0; i < array.length; i++) {
    if (!used[i]) {
      used[i] = true;
      cur.append(array[i]);
      helperWithOrder(array, used, cur, result);
      used[i] = false;
```

```
        cur.deleteCharAt(cur.length() - 1);
      }
    }
  }
}
```

Back To Index

Class 7 HashTable & String 1

Top K Frequent Words

```
public class TopKFrequent {
  // Assumptions: k >= 1.
  public String[] topKFrequent(List<String> combo, int k) {
    List<String> topk = new ArrayList<String>();
    // construct the map of <word, frequency>.
    final HashMap<String, Long> freqMap = new HashMap<String, Long>();
    for (String w : combo) {
      Long count = freqMap.get(w);
      if (count != null) {
        freqMap.put(w, count + 1);
      } else {
        freqMap.put(w, 1L);
      }
    }
    // min heap based on the words' frequencies.
    PriorityQueue<String> minHeap = new PriorityQueue<String>(k, new Comparator<String>() {
      @Override
      public int compare(String w1, String w2) {
        long f1 = freqMap.get(w1);
        long f2 = freqMap.get(w2);
        if (f1 == f2) {
          return 0;
        }
        return f1 < f2 ? -1 : 1;
      }
    });
    // iterate the map and put the word into the min heap for top k.
    for (Map.Entry<String, Long> entry : freqMap.entrySet()) {
      if (minHeap.size() < k) {
        minHeap.offer(entry.getKey());
      } else if (entry.getValue() > freqMap.get(minHeap.peek())) {
```

```
        minHeap.poll();
        minHeap.offer(entry.getKey());
      }
    }
    while (!minHeap.isEmpty()) {
      topk.add(minHeap.poll());
    }
    // need to return array of words in ascending order of frequencies.
    Collections.reverse(topk);
    // convert the list to array.
    return topk.toArray(new String[0]);
  }
}
```

Missing Number I

```
public class MissingNumberI {
  // Method 1: use HashSet.
  public int missingI(int[] array) {
    int n = array.length + 1;
    HashSet<Integer> set = new HashSet<Integer>();
    for (int number : array) {
      set.add(number);
    }
    for (int i = 1; i < n; i++) {
      if (!set.contains(i)) {
        return i;
      }
    }
    return n;
  }

  // Method 2: use sum.
  public int missingII(int[] array) {
    int n = array.length + 1;
    long targetSum = (n + 0L) * (n + 1) / 2;
    long actualSum = 0L;
    for (int num : array) {
      actualSum += num;
    }
    return (int) (targetSum - actualSum);
  }
}
```

Common Numbers Of Two Sorted Arrays

```
// Assumptions: there could be duplicated elements in the given arrays.
public class CommonNumbersII {
  // Method 1: two pointers.
  public List<Integer> commonI(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;
  }

  // Method 2: use HashMap.
  public List<Integer> commonII(int[] a, int[] b) {
    // process the smaller array first.
    if (a.length > b.length) {
      return commonII(b, a);
    }
    List<Integer> common = new ArrayList<Integer>();
    HashMap<Integer, Integer> countA = new HashMap<Integer, Integer>();
    for (int num : a) {
      Integer ct = countA.get(num);
      if (ct != null) {
        countA.put(num, ct + 1);
      } else {
        countA.put(num, 1);
      }
    }
    HashMap<Integer, Integer> countB = new HashMap<Integer, Integer>();
    for (int num : b) {
```

```
      Integer ct = countB.get(num);
      if (ct != null) {
        countB.put(num, ct + 1);
      } else {
        countB.put(num, 1);
      }
    }
    for (Map.Entry<Integer, Integer> entry : countA.entrySet()) {
      Integer ctInB = countB.get(entry.getKey());
      if (ctInB != null) {
        int appear = Math.min(entry.getValue(), ctInB);
        for (int i = 0; i < appear; i++) {
          common.add(entry.getKey());
        }
      }
    }
    return common;
  }
}
```

Remove ‘u’ And ‘n’ From Word

```
public class RemoveUN {
  public String removeUN(String input) {
    if (input == null || input.length() == 0) {
      return input;
    }
    char[] array = input.toCharArray();
    int slow = 0;
    for (int fast = 0; fast < array.length; fast++) {
      if (array[fast] == 'u' || array[fast] == 'n') {
        continue;
      }
      array[slow++] = array[fast];
    }
    return new String(array, 0, slow);
  }
}
```

Remove All Leading/Trailing/Duplicate Space Characters

```
public class CharRemoval {
  public String removeSpaces(String input) {
```

```
    // input is not null.
    if (input.isEmpty()) {
      return input;
    }
    char[] array = input.toCharArray();
    int end = 0;
    for (int i = 0; i < array.length; i++) {
      if (array[i] == ' ' && (i == 0 || array[i - 1] == ' ')) {
        continue;
      }
      array[end++] = array[i];
    }
    if (end > 0 && array[end - 1] == ' ') {
      return new String(array, 0, end - 1);
    }
    return new String(array, 0, end);
  }
}
```

Remove Adjacent Repeated Characters I

```
public class RemoveDuplicateI {
  /*
   *  try to convert the string to char array,
   *  and do it in place.
   *  For the adjacent repeated characters, only retain one of them.
   */
  public String deDup(String input) {
    if (input == null) {
      return null;
    }
    char[] array = input.toCharArray();
    int end = 0;
    for (int i = 0; i < array.length; i++) {
      if (i == 0 || array[i] != array[end - 1]) {
        array[end++] = array[i];
      }
    }
    return new String(array, 0, end);
  }
}
```

Remove Adjacent Repeated Characters IV

```
public class RemoveDuplicateIV {
  public String deDup(String input) {
    /*
     *  try to convert the string to char array,
     *  and do it in place.
     *  Remove all adjacent repeated characters repeatedly.
     *  “abbbaac” → “aaac” → “c”
     */
    if (input == null || input.length() <= 1) {
      return input;
    }
    char[] array = input.toCharArray();
    int end = 0;
    for (int i = 1; i < array.length; i++) {
      if (end == -1 || array[i] != array[end]) {
        array[++end] = array[i];
      } else {
        end--;
        while (i + 1 < array.length && array[i] == array[i + 1]) {
          i++;
        }
      }
    }
    return new String(array, 0, end + 1);
  }
}
```

Determine If One String Is Another’s Substring

```
// There is no assumption about the charset used in the String.
public class Strstr {
  // Method 1: naive solution.
  public int strstrI(String large, String small) {
    if (large.length() < small.length()) {
      return -1;
    }
    // return 0 if small is empty.
    if (small.length() == 0) {
      return 0;
    }
    for (int i = 0; i <= large.length() - small.length(); i++) {
```

```java
      if (equals(large, i, small)) {
        return i;
      }
    }
    return -1;
  }

  // Method 2: RabinKarp
  public int strstrII(String large, String small) {
    if (large.length() < small.length()) {
      return -1;
    }
    // return 0 if small is empty.
    if (small.length() == 0) {
      return 0;
    }
    // large prime as module end.
    int largePrime = 101;
    // small prime to calculate the hash value.
    int prime = 31;
    int seed = 1;
    int targetHash = small.charAt(0) % largePrime;
    for (int i = 1; i < small.length(); i++) {
      seed = moduleHash(seed, 0, prime, largePrime);
      targetHash = moduleHash(targetHash, small.charAt(i), prime, largePrime);
    }
    int hash = 0;
    for (int i = 0; i < small.length(); i++) {
      hash = moduleHash(hash, large.charAt(i), prime, largePrime);
    }
    if (hash == targetHash && equals(large, 0, small)) {
      return 0;
    }
    for (int i = 1; i <= large.length() - small.length(); i++) {
      hash = nonNegative(hash - seed * large.charAt(i - 1) % largePrime, largePrime);
      hash = moduleHash(hash, large.charAt(i + small.length() - 1), prime, largePrime);
      if (hash == targetHash && equals(large, i, small)) {
        return i;
      }
    }
    return -1;
  }
```

```
  public boolean equals(String large, int start, String small) {
    for (int i = 0; i < small.length(); i++) {
      if (large.charAt(i + start) != small.charAt(i)) {
        return false;
      }
    }
    return true;
  }

  public int moduleHash(int hash, int addition, int prime, int largePrime) {
    return (hash * prime % largePrime + addition) % largePrime;
  }

  public int nonNegative(int hash, int largePrime) {
    if (hash < 0) {
      hash += largePrime;
    }
    return hash;
  }
}
```

[Back To Index]

Class 10 - String II

Reverse String

```
public class ReverseString {
  // Method 1: iterative reverse.
  public String reverse(String input) {
    if (input == null || input.length() <= 1) {
      return input;
    }
    char[] array = input.toCharArray();
    for (int left = 0, right = array.length - 1; left < right; left++, right--) {
      swap(array, left, right);
    }
    return new String(array);
  }

  // Method 2: recursive reverse.
  public String reverseRecursive(String input) {
    if (input == null || input.length() <= 1) {
```

```
    return input;
  }
  char[] array = input.toCharArray();
  reverseHelper(array, 0, array.length - 1);
  return new String(array);
}

private void reverseHelper(char[] array, int left, int right) {
  if (left >= right) {
    return;
  }
  swap(array, left, right);
  reverseHelper(array, left + 1, right - 1);
}

private void swap(char[] array, int a, int b) {
  char tmp = array[a];
  array[a] = array[b];
  array[b] = tmp;
}
}
```

Reverse Words In Sentence

```
/**
 * 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--;
    }
  }
}
```

Right Shift By N Characters

```
public class RightShift {

  public String rightShift(String input, int n) {
    if (input == null || input.length() <= 1) {
      return input;
    }
    char[] array = input.toCharArray();
    n %= array.length;
    reverse(array, array.length - n, array.length - 1);
    reverse(array, 0, array.length - n - 1);
    reverse(array, 0, array.length - 1);
    return new String(array);
  }

  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--;
    }
  }
}
```

String Replace

```
/**
 * Replace all substrings s1 in a string s with s2
 * (with possible minimum memory allocation, in-place if possible).
 */
public class StrReplace {
  // Method 1: Solution with using StringBuilder and substring().
  public String replace(String input, String s, String t) {
    // Assumptions: input, s, t are not null, s is not empty
    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();
  }

  // Method 2: Solution with char array("in place").
  public String replaceII(String input, String s, String t) {
    // Assumptions: input, s, t are not null, s is not empty
    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();) {
    if (equalSubArray(input, i, s)) {
      matches.add(i + s.length() - 1);
      i += s.length();
    } else {
      i++;
    }
  }
  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;
    }
  }
```

```java
    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));
  }
}
```

String Shuffling

```java
/**
 * Array reorder in place implementation.
 *
 * Suppose I have a array of chars, the requirement is as follow:
 * 1). [C_1, C_2, ..., C_2k]
 * --> [C_1, C_k+1, C2, C_k+2, ..., C_k, C_2k]
 *
```

```java
 * 2). [C_1, C_2, ..., C_2k+1]
 * --> [C_1, C_K+1, C2, C_K+2, ... C_K, C_2k, C_2k+1]
 */
public class ArrayReOrder {

	public void reorder(char[] array) {
		if (array == null) {
			return;
		}
		int length = array.length;
		if (length % 2 != 0) {
			reorder(array, 0, length - 2);
		} else {
			reorder(array, 0, length - 1);
		}
	}

	private void reorder(char[] array, int left, int right) {
		int length = right - left + 1;
		assert length % 2 == 0;
		if (length <= 2) {
			return;
		}
		int leftLength = 2 * (length / 4);
		int leftEnd = left + leftLength - 1;
		int leftMid = left + leftLength / 2;
		rightShift(array, leftMid, leftMid + length / 2 - 1, leftLength / 2);
		reorder(array, left, leftEnd);
		reorder(array, leftEnd + 1, right);
	}

	public void rightShift(char[] array, int left, int right, int shift) {
		reverse(array, right - shift + 1, right);
		reverse(array, left, right - shift);
		reverse(array, left, right);
	}

	public void reverse(char[] array, int left, int right) {
		while (left < right) {
			swap(array, left++, right--);
		}
	}
```

```
        public void swap(char[] array, int left, int right) {
                char temp = array[left];
                array[left] = array[right];
                array[right] = temp;
        }
}
```

All Permutations II(with duplicate characters)

```
public class PermutationsII {
  public List<String> permutations(String set) {
    List<String> result = new ArrayList<String>();
    if (set == null) {
      return result;
    }
    char[] array = set.toCharArray();
    helper(array, 0, result);
    return result;
  }

  private void helper(char[] array, int index, List<String> result) {
    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, i, index);
        helper(array, index + 1, result);
        swap(array, i, index);
      }
    }
  }

  private void swap(char[] array, int left, int right) {
    char tmp = array[left];
    array[left] = array[right];
    array[right] = tmp;
  }
}
```

Decompress String II

```
/**
 * Given a string in compressed form, decompress it to the original string. The
 * adjacent repeated characters in the original string are compressed to have
 * the character followed by the number of repeated occurrences.
 *
 * Assumptions:
 * 1. The string is not null
 * 2. The characters used in the original string are guaranteed to be ‘a’ - ‘z’
 * 3. There are no adjacent repeated characters with length > 9
 *
 * Examples:
 *
 * “a1c0b2c4” → “abbcccc”
 *
 */
public class DecompressStringII {
  // Method 1: using StringBuilder to help.
  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();
  }

  // Method 2: "in place".
  public String decompressII(String input) {
    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;
```

```
  }
}
```

Longest Substring With Only Unique Characters

```
public class LongestSubstringUnique {
  public String longest(String input) {
    // Assumptions: input is not null.
    String result = "";
    Set<Character> set = new HashSet<Character>();
    int left = 0;
    int right = 0;
    while (right < input.length()) {
      if (set.contains(input.charAt(right))) {
        set.remove(input.charAt(left++));
      } else {
        set.add(input.charAt(right++));
        if (right - left > result.length()) {
          result = input.substring(left, right);
        }
      }
    }
    return result;
  }
}
```

Find All Anagrams Of Short String In A Long String

```
// Find all anagrams of String s in String l,
// return all the starting indices.
public class AllAnagrams {
  public List<Integer> allAnagrams(String s, String l) {
    List<Integer> result = new ArrayList<Integer>();
    if (s == null || l == null || s.length() == 0 || l.length() == 0) {
      return result;
    }
    if (s.length() > l.length()) {
      return result;
    }
    Map<Character, Integer> map = countMap(s);
    int match = 0;
    for (int i = 0; i < l.length(); i++) {
      char tmp = l.charAt(i);
```

```
      Integer count = map.get(tmp);
      if (count != null) {
        map.put(tmp, count - 1);
        if (count == 1) {
          match++;
        }
      }
      if (i >= s.length()) {
        tmp = l.charAt(i - s.length());
        count = map.get(tmp);
        if (count != null) {
          map.put(tmp, count + 1);
          if (count == 0) {
            match--;
          }
        }
      }
      if (match == map.size()) {
        result.add(i - s.length() + 1);
      }
    }
    return result;
  }

  private Map<Character, Integer> countMap(String s) {
    Map<Character, Integer> map = new HashMap<Character, Integer>();
    for (char ch : s.toCharArray()) {
      Integer count = map.get(ch);
      if (count == null) {
        map.put(ch, 1);
      } else {
        map.put(ch, count + 1);
      }
    }
    return map;
  }
}
```

[Back To Index]

Class 11 - Recursion II

N Queens

```
public class NQueens {
  public List<List<Integer>> nqueens(int n) {
    // n > 0
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> cur = new ArrayList<Integer>();
    helper(0, n, cur, result);
    return result;
  }

  private void helper(int index, int n, List<Integer> cur, List<List<Integer>> result) {
    if (index == n) {
      result.add(new ArrayList<Integer>(cur));
      return;
    }
    for (int i = 0; i < n; i++) {
      cur.add(i);
      if (valid(cur)) {
        helper(index + 1, n, cur, result);
      }
      cur.remove(cur.size() - 1);
    }
  }

  private boolean valid(List<Integer> cur) {
    int size = cur.size();
    for (int i = 0; i < size - 1; i++) {
      if (cur.get(i).equals(cur.get(size - 1)) || cur.get(i) - cur.get(size - 1) == i + 1 - size
          || cur.get(i) - cur.get(size - 1) == size - 1 - i) {
        return false;
      }
    }
    return true;
  }
}
```

Spiral Order Traverse II

```
public class SpiralPrintII {
  public List<Integer> spiral(int[][] matrix) {
    // Assumptions: matrix is not null, has size of M * N, where M, N >=0
```

```java
List<Integer> list = new ArrayList<Integer>();
int m = matrix.length;
if (m == 0) {
  return list;
}
int n = matrix[0].length;
if (n == 0) {
  return list;
}

int left = 0;
int right = n - 1;
int up = 0;
int down = m - 1;
while (left < right && up < down) {
  for (int i = left; i <= right; i++) {
    list.add(matrix[up][i]);
  }
  for (int i = up + 1; i <= down - 1; i++) {
    list.add(matrix[i][right]);
  }
  for (int i = right; i >= left; i--) {
    list.add(matrix[down][i]);
  }
  for (int i = down - 1; i >= up + 1; i--) {
    list.add(matrix[i][left]);
  }
  left++;
  right--;
  up++;
  down--;
}
if (left > right || up > down) {
  return list;
}
if (left == right) {
  for (int i = up; i <= down; i++) {
    list.add(matrix[i][left]);
  }
} else {
  for (int i = left; i <= right; i++) {
    list.add(matrix[up][i]);
  }
```

```
    }
    return list;
  }
}
```

Reverse Linked List In Pairs

```
/**
 * class ListNode {
 *   public int value;
 *   public ListNode next;
 *   public ListNode(int value) {
 *     this.value = value;
 *     next = null;
 *   }
 * }
 */
public class ReverseListInPairs {
  // Method 1: recursion.
  public ListNode reverseInPairs(ListNode head) {
    if (head == null || head.next == null) {
      return head;
    }
    ListNode newHead = head.next;
    head.next = reverseInPairs(head.next.next);
    newHead.next = head;
    return newHead;
  }

  // Method 2: iterative.
  public ListNode reverseInPairsI(ListNode head) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode cur = dummy;
    while (cur.next != null && cur.next.next != null) {
      ListNode next = cur.next.next;
      cur.next.next = cur.next.next.next;
      next.next = cur.next;
      cur.next = next;
      cur = cur.next.next;
    }
    return dummy.next;
  }
```

```
}
```

Abbreviation Matching

```
public class AbbMatch {
  public boolean match(String input, String pattern) {
    // Assumptions: input, pattern != null.
    return match(input, pattern, 0, 0);
  }

  private boolean match(String s, String t, int si, int ti) {
    if (si == s.length() && ti == t.length()) {
      return true;
    }
    if (si >= s.length() || ti >= t.length()) {
      return false;
    }
    if (t.charAt(ti) < '0' || t.charAt(ti) > '9') {
      if (s.charAt(si) == t.charAt(ti)) {
        return match(s, t, si + 1, ti + 1);
      }
      return false;
    }
    int count = 0;
    while (ti < t.length() && t.charAt(ti) >= '0' && t.charAt(ti) <= '9') {
      count = count * 10 + (t.charAt(ti) - '0');
      ti++;
    }
    return match(s, t, si + count, ti);
  }
}
```

Store Number Of Nodes In Left Subtree

```
public class NumNodesLeft {
  static class TreeNode {
    int key;
    TreeNode left;
    TreeNode right;
    int numNodesLeft;

    public TreeNode(int key) {
      this.key = key;
```

```
    }
  }

  public void numNodesLeft(TreeNode root) {
    numNodes(root);
  }

  private int numNodes(TreeNode root) {
    if (root == null) {
      return 0;
    }
    int leftNum = numNodes(root.left);
    int rightNum = numNodes(root.right);
    root.numNodesLeft = leftNum;
    return leftNum + rightNum + 1;
  }
}
```

Given a binary tree, find the node with the max difference in the total number descendents in its left subtree and right subtree

```
public class MaxDiffNode {
  public TreeNode maxDiffNode(TreeNode root) {
    if (root == null) {
      return null;
    }
    TreeNode[] node = new TreeNode[1];
    int[] diff = new int[1];
    diff[0] = -1;
    numNodes(root, node, diff);
    return node[0];
  }

  private int numNodes(TreeNode root, TreeNode[] node, int[] diff) {
    if (root == null) {
      return 0;
    }
    int leftNum = numNodes(root.left, node, diff);
    int rightNum = numNodes(root.right, node, diff);
    if (Math.abs(leftNum - rightNum) > diff[0]) {
      node[0] = root;
      diff[0] = Math.abs(leftNum - rightNum);
    }
```

```
    return leftNum + rightNum + 1;
  }
}
```

Lowest Common Ancestor I

```
public class LCAI {
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode one, TreeNode two) {
    // Assumptions: root is not null, one and two guaranteed to be in the tree.
    if (root == null) {
      return null;
    }
    if (root == one || root == two) {
      return root;
    }
    TreeNode ll = lowestCommonAncestor(root.left, one, two);
    TreeNode lr = lowestCommonAncestor(root.right, one, two);
    if (ll != null && lr != null) {
      return root;
    } else if (ll != null) {
      return ll;
    } else {
      return lr;
    }
  }
}
```

[Back To Index](#)

Class 12 - Bit Representation and Bit Operation

Determine If A Number Is Power Of 2

```
/**
 * Determine if an integer is power of 2.
 */
public class PowerOfTwo {
  // Method 1.
  public boolean isPowerOfTwoI(int number) {
    if (number <= 0) {
      return false;
    }
    while ((number & 1) == 0) {
      number >>>= 1;
```

```
    }
    return number == 1;
  }
  // Method 2.
  public boolean isPowerOfTwoII(int number) {
    if (number <= 0) {
      return false;
    }
    int count = 0;
    while (number > 0) {
      count += number & 1;
      number >>>= 1;
    }
    return count == 1;
  }
  // Method 3
  public boolean isPowerOfTwoIII(int number) {
    return number > 0 && (number & (number - 1)) == 0;
  }
}
```

Number Of Different Bits

```
public class NumberOfDiffBits {
  public int diffBits(int a, int b) {
    a ^= b;
    int count = 0;
    while (a != 0) {
      count += a & 1;
      a >>>= 1;
    }
    return count;
  }
}
```

All Unique Characters II

```
/**
 * 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.
 * The input word is not null.
```

```
 */
public class AllUniqueCharsII {
  public boolean allUnique(String word) {
    int[] vec = new int[8];
    char[] array = word.toCharArray();
    for (char c : array) {
      if ((vec[c / 32] >>> (c % 32) & 1) != 0) {
        return false;
      }
      vec[c / 32] |= 1 << (c % 32);
    }
    return true;
  }
}
```

Hexadecimal Representation

```
public class HexRepresentation {
  // Assumptions: number >= 0
  public String hex(int number) {
    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();
  }
}
```

Reverse Bits

```
public class ReverseBits {
  public int reverseI(int num) {
    for (int offset = 0; offset < 16; ++offset) {
```

```
      int right = (num >> offset) & 1;
      int left = (num >> (31 - offset)) & 1;
      if (left != right) {
        num ^= (1 << offset);
        num ^= (1 << (31 - offset));
      }
    }
    return num;
  }

  // merge sort way of reversing all the bits.
  public int reverseII(int num) {
    num = ((num & 0xFFFF0000) >>> 16) | ((num & 0x0000FFFF) << 16);
    num = ((num & 0xFF00FF00) >>> 8) | ((num & 0x00FF00FF) << 8);
    num = ((num & 0xF0F0F0F0) >>> 4) | ((num & 0x0F0F0F0F) << 4);
    num = ((num & 0xCCCCCCCC) >>> 2) | ((num & 0x33333333) << 2);
    num = ((num & 0xAAAAAAAA) >>> 1) | ((num & 0x55555555) << 1);
    return num;
  }
}
```

Back To Index

Class 13 - Dynamic Programming I

Longest Ascending Subarray

```
public class LongestAscendingSubArray {
  public int longest(int[] array) {
    // Assumptions: the given array is not null.
    if (array.length == 0) {
      return 0;
    }
    int result = 1;
    int cur = 1;
    for (int i = 1; i < array.length; i++) {
      // if array[i] > array[i - 1], the current ascending subarray can add one element.
      if (array[i] > array[i - 1]) {
        cur++;
        result = Math.max(result, cur);
      } else {     // otherwise, we need to start a new ascending subarray.
        cur = 1;
      }
```

```
    }
    return result;
  }
}
```

Max Product Of Cutting Rope

```
public class MaxProductOfCuttingRope {
  public int maxProduct(int length) {
    // Assumptions: length >= 2
    if (length == 2) {
      return 1;
    }
    int[] array = new int[length + 1];
    array[1] = 1;
    array[2] = 1;
    for (int i = 3; i < array.length; i++) {
      for (int j = 1; j <= i / 2; j++) {
        // after cutting, one of the partition is length j,
        // for the other partition, we can take the max of (i - j) and array[i - j]
        // (no cut or cut at least once).
        array[i] = Math.max(array[i], j * Math.max(i - j, array[i - j]));
      }
    }
    return array[length];
  }
}
```

Jump Game I

```
/**
 * 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) {
    // Assumptions: array is not null and is not empty.
    // Method 1: DP, canJump[i] means from index 0, can jump to index i.
    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];
}

// Method 2: DP, canJump[i] means from index i, can jump to index array.length - 1
public boolean canJumpII(int[] array) {
  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];
}

// Method 3: Greedy solution.
public boolean canJumpIII(int[] array) {
  // Assumptions: array is not null and array.length >= 1.
  if (array.length == 1) {
    return true;
  }
  // the max index cur jump can reach
  int cur = 0;
  // the max index next jump can reach
  int next = 0;
```

```
    for (int i = 0; i < array.length; i++) {
      if (i > cur) {
        // if i > cur, we need a jump from cur to next.
        if (cur == next) {
          // cur == next means there is no progress,
          // if that is the case, we can never reach end of array.
          return false;
        }
        cur = next;
      }
      next = Math.max(next, i + array[i]);
    }
    return true;
  }
}
```

Back To Index

Class 14 - Dynamic Programming II

Jump Game II

```
/**
 * 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 {
  // Method 1: DP
  public int minJumpI(int[] array) {
    // Assumptions: array is not null and is not empty.
    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];
  }

  // Method 2: Greedy solution.
  public int minJumpII(int[] array) {
    if (array.length == 1) {
      return 0;
    }
    // # of jumps currently
    int jump = 0;
    // max index by current # of jumps
    int cur = 0;
    // max index by current # of jumps + 1
    int next = 0;
    for (int i = 0; i < array.length; i++) {
      if (i > cur) {
        jump++;
        if (cur == next) {
          return array.length;
        }
        cur = next;
      }
      next = Math.max(next, array[i] + i);
    }
    return jump;
  }
}
```

Dictionary Word

```
public class DictionaryWord {
  public boolean canBreak(String input, Set<String> dict) {
    // Assumptions :
    // input != null && input.length() > 0
    // dict not null/empty string
    boolean[] canBreak = new boolean[input.length() + 1];
    canBreak[0] = true;
    for (int i = 1; i < canBreak.length; i++) {
      for (int j = 0; j < i; j++) {
        if (dict.contains(input.substring(j, i)) && canBreak[j]) {
```

```
          canBreak[i] = true;
          break;
        }
      }
    }
    return canBreak[canBreak.length - 1];
  }
}
```

Edit Distance

```
public class EditDistance {
  public int editDistance(String one, String two) {
    // Assumptions: one,two are not null
    int[][] distance = new int[one.length() + 1][two.length() + 1];
    for (int i = 0; i <= one.length(); i++) {
      for (int j = 0; j <= two.length(); j++) {
        if (i == 0) {
          distance[i][j] = j;
        } else if (j == 0) {
          distance[i][j] = i;
        } else if (one.charAt(i - 1) == two.charAt(j - 1)) {
          distance[i][j] = distance[i - 1][j - 1];
        } else {
          distance[i][j] = Math.min(distance[i - 1][j], distance[i][j - 1]);
          distance[i][j] = Math.min(distance[i - 1][j - 1], distance[i][j]);
          distance[i][j] += 1;
        }
      }
    }
    return distance[one.length()][two.length()];
  }
}
```

Largest Square Of “1” s

```
public class LargestSquareOfOnes {
  public int largest(int[][] matrix) {
    // all 0 or 1, N * N, not null, N >= 0
    int N = matrix.length;
    if (N == 0) {
      return 0;
    }
```

```
    int result = 0;
    int[][] largest = new int[N][N];
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        if (i == 0 || j == 0) {
          largest[i][j] = matrix[i][j] == 1 ? 1 : 0;
        } else if (matrix[i][j] == 1) {
          largest[i][j] = Math.min(largest[i][j - 1], largest[i - 1][j]);
          largest[i][j] = Math.min(largest[i - 1][j - 1], largest[i][j]);
          largest[i][j]++;
        }
        result = Math.max(result, largest[i][j]);
      }
    }
    return result;
  }
}
```

Back To Index

Class 15 - Dynamic Programming III

Largest Subarray Sum

```
public class LargestSubArraySum {
  public int largestSum(int[] array) {
    // Assumptions: array != null && length >= 1
    // The subarray must at least contain one element.
    int result = array[0];
    int cur = array[0];
    for (int i = 1; i < array.length; i++) {
      cur = Math.max(cur + array[i], array[i]);
      result = Math.max(result, cur);
    }
    return result;
  }
}
```

Longest Consecutive “1”s

```
public class LongestOnes {
  public int longest(int[] array) {
    // array is not null
    int result = 0;
```

```
    int cur = 0;
    for (int i = 0; i < array.length; i++) {
      if (array[i] == 1) {
        if (i == 0 || array[i - 1] == 0) {
          cur = 1;
        } else {
          cur++;
        }
        result = Math.max(result, cur);
      }
    }
    return result;
  }
}
```

Largest Cross With All “1”s

```
public class LargestCrossOfOnes {
  public int largest(int[][] matrix) {
    // matrix is not null, N * M,
    int N = matrix.length;
    if (N == 0) {
      return 0;
    }
    int M = matrix[0].length;
    if (M == 0) {
      return 0;
    }

    int[][] leftUp = leftUp(matrix, N, M);
    int[][] rightDown = rightDown(matrix, N, M);
    return merge(leftUp, rightDown, N, M);
  }

  private int merge(int[][] leftUp, int[][] rightDown, int N, int M) {
    int result = 0;
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < M; j++) {
        leftUp[i][j] = Math.min(leftUp[i][j], rightDown[i][j]);
        result = Math.max(result, leftUp[i][j]);
      }
    }
    return result;
```

```java
}

private int[][] leftUp(int[][] matrix, int N, int M) {
  int[][] left = new int[N][M];
  int[][] up = new int[N][M];
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      if (matrix[i][j] == 1) {
        if (i == 0 && j == 0) {
          up[i][j] = 1;
          left[i][j] = 1;
        } else if (i == 0) {
          up[i][j] = 1;
          left[i][j] = left[i][j - 1] + 1;
        } else if (j == 0) {
          up[i][j] = up[i - 1][j] + 1;
          left[i][j] = 1;
        } else {
          up[i][j] = up[i - 1][j] + 1;
          left[i][j] = left[i][j - 1] + 1;
        }
      }
    }
  }
  merge(left, up, N, M);
  return left;
}

private int[][] rightDown(int[][] matrix, int N, int M) {
  int[][] right = new int[N][M];
  int[][] down = new int[N][M];
  for (int i = N - 1; i >= 0; i--) {
    for (int j = M - 1; j >= 0; j--) {
      if (matrix[i][j] == 1) {
        if (i == N - 1 && j == M - 1) {
          down[i][j] = 1;
          right[i][j] = 1;
        } else if (i == N - 1) {
          down[i][j] = 1;
          right[i][j] = right[i][j + 1] + 1;
        } else if (j == M - 1) {
          down[i][j] = down[i + 1][j] + 1;
          right[i][j] = 1;
```

```
        } else {
          down[i][j] = down[i + 1][j] + 1;
          right[i][j] = right[i][j + 1] + 1;
        }
      }
    }
  }
  merge(right, down, N, M);
  return right;
 }
}
```

Largest X With All “1”s

```
public class LargestXOfOnes {
 public int largest(int[][] matrix) {
  // matrix is not null
  int N = matrix.length;
  if (N == 0) {
   return 0;
  }
  int M = matrix[0].length;
  if (M == 0) {
   return 0;
  }
  int[][] leftUp = leftUp(matrix, N, M);
  int[][] rightDown = rightDown(matrix, N, M);
  return merge(leftUp, rightDown, N, M);
 }

 private int merge(int[][] leftUp, int[][] rightDown, int N, int M) {
  int result = 0;
  for (int i = 0; i < N; i++) {
   for (int j = 0; j < M; j++) {
    leftUp[i][j] = Math.min(leftUp[i][j], rightDown[i][j]);
    result = Math.max(result, leftUp[i][j]);
   }
  }
  return result;
 }

 private int[][] leftUp(int[][] matrix, int N, int M) {
  int[][] left = new int[N][M];
```

```
    int[][] up = new int[N][M];
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < M; j++) {
        if (matrix[i][j] == 1) {
          left[i][j] = getNumber(left, i - 1, j - 1, N, M) + 1;
          up[i][j] = getNumber(up, i - 1, j + 1, N, M) + 1;
        }
      }
    }
    merge(left, up, N, M);
    return left;
  }

  private int[][] rightDown(int[][] matrix, int N, int M) {
    int[][] right = new int[N][M];
    int[][] down = new int[N][M];
    for (int i = N - 1; i >= 0; i--) {
      for (int j = M - 1; j >= 0; j--) {
        if (matrix[i][j] == 1) {
          right[i][j] = getNumber(right, i + 1, j + 1, N, M) + 1;
          down[i][j] = getNumber(down, i + 1, j - 1, N, M) + 1;
        }
      }
    }
    merge(right, down, N, M);
    return right;
  }

  private int getNumber(int[][] number, int x, int y, int N, int M) {
    if (x < 0 || x >= N || y < 0 || y >= M) {
      return 0;
    }
    return number[x][y];
  }
}
```

Given a matrix where every element is either ‘0’ or ‘1’, find the largest subsquare surrounded by ‘1’.

```
public class LargestSquareSurroundedByOne {
  // Return the length of the largest square.
  public int largest(int[][] matrix) {
    // matrix is not null, size of M * N, where M, N >= 0
```

```
    // the elements in the matrix are either 0 or 1
    if (matrix.length == 0 || matrix[0].length == 0) {
      return 0;
    }
    int result = 0;
    int M = matrix.length;
    int N = matrix[0].length;
    int[][] left = new int[M + 1][N + 1];
    int[][] up = new int[M + 1][N + 1];
    for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++) {
        if (matrix[i][j] == 1) {
          left[i + 1][j + 1] = left[i + 1][j] + 1;
          up[i + 1][j + 1] = up[i][j + 1] + 1;
          for (int maxLength = Math.min(left[i + 1][j + 1], up[i + 1][j + 1]); maxLength >= 1;
maxLength--) {
            if (left[i + 2 - maxLength][j + 1] >= maxLength
                && up[i + 1][j + 2 - maxLength] >= maxLength) {
              result = Math.max(result, maxLength);
              break;
            }
          }
        }
      }
    }
    return result;
  }
}
```

Largest Submatrix Sum

```
public class LargestSubMatrixSum {
  public int largest(int[][] matrix) {
    // matrix is not null and N, M >= 1
    int N = matrix.length;
    int M = matrix[0].length;
    int result = Integer.MIN_VALUE;
    for (int i = 0; i < N; i++) {
      int[] cur = new int[M];
      for (int j = i; j < N; j++) {
        add(cur, matrix[j]);
        result = Math.max(result, max(cur));
      }
```

```java
    }
    return result;
  }

  private void add(int[] cur, int[] add) {
    for (int i = 0; i < cur.length; i++) {
      cur[i] += add[i];
    }
  }

  private int max(int[] cur) {
    int result = cur[0];
    int tmp = cur[0];
    for (int i = 1; i < cur.length; i++) {
      tmp = Math.max(tmp + cur[i], cur[i]);
      result = Math.max(result, tmp);
    }
    return result;
  }
}
```

Cutting Wood I

```java
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](#)

Class 16 - Probability, Sampling, Randomization

Shuffle

```
public class PerfectShuffle {
 public void shuffle(int[] array) {
  if (array == null || array.length <= 1) {
    return;
  }
  for (int i = 0; i < array.length; i++) {
    int idx = (int) (Math.random() * (array.length - i)) + i;
    swap(array, i, idx);
  }
 }

 private void swap(int[] array, int left, int right) {
  int tmp = array[left];
  array[left] = array[right];
  array[right] = tmp;
 }
}
```

Reservoir Sampling

```
public class ReservoirSampling {
 // how many numbers have been read so far.
 private int count;
 // only need to maintain the current sample.
 private Integer sample;

 public ReservoirSampling() {
  this.count = 0;
  this.sample = null;
 }
```

```
  public void read(int value) {
    count++;
    int prob = (int) (Math.random() * count);
    // the current read value has the probability of 1 / count to be as the
    // current sample.
    if (prob == 0) {
      sample = value;
    }
  }

  public Integer sample() {
    return sample;
  }
}
```

Random7 Using Random5

```
public class RandomSeven {
  public int random7() {
    while (true) {
      int random = 5 * RandomFive.random5() + RandomFive.random5();
      if (random < 21) {
        return random % 7;
      }
    }
  }
}
```

Median Tracker Of Data Flow

```
public class MedianTracker {
  private PriorityQueue<Integer> min;
  private PriorityQueue<Integer> max;

  public MedianTracker() {
    max = new PriorityQueue<Integer>();
    min = new PriorityQueue<Integer>(11, Collections.reverseOrder());
  }

  public void read(int value) {
    if (min.isEmpty() || value <= min.peek()) {
      min.offer(value);
    } else {
```

```
      max.offer(value);
    }
    if (min.size() - max.size() >= 2) {
      max.offer(min.poll());
    } else if (max.size() > min.size()) {
      min.offer(max.poll());
    }
  }

  public Double median() {
    int size = size();
    if (size == 0) {
      return null;
    } else if (size % 2 != 0) {
      return (double) (min.peek());
    } else {
      return (min.peek() + max.peek()) / 2.0;
    }
  }

  private int size() {
    return min.size() + max.size();
  }
}
```

95th Percentile

```
public class NinetyFivePercentile {
  public int percentile95(List<Integer> lengths) {
    // lengths is not null and size >= 1 with non null
    int[] count = new int[4097];
    for (int len : lengths) {
      count[len]++;
    }
    int sum = 0;
    int len = 4097;
    while (sum <= 0.05 * lengths.size()) {
      sum += count[--len];
    }
    return len;
  }
}
```

[Back To Index]

Class 18 - 加强练习 I

Array Deduplication I(sorted array, duplicate element only retain one)

```
public class ArrayDeduplicationI {
  public int dedup(int[] array) {
    // array is not null
    if (array.length <= 1) {
      return array.length;
    }
    int end = 0;
    for (int i = 1; i < array.length; i++) {
      if (array[i] != array[end]) {
        array[++end] = array[i];
      }
    }
    return end + 1;
  }
}
```

Array Deduplication II(sorted array, duplicate element only retain two)

```
public class ArrayDeduplicationII {
  public int dedup(int[] array) {
    // array is not null
    if (array.length <= 2) {
      return array.length;
    }
    int end = 1;
    for (int i = 2; i < array.length; i++) {
      if (array[i] != array[end - 1]) {
        array[++end] = array[i];
      }
    }
    return end + 1;
  }
}
```

Array Deduplication III(sorted array, duplicate element not retain any)

```
public class ArrayDeduplicationIII {
  public int dedup(int[] array) {
```

```
    // array is not null;
    if (array == null || array.length <= 1) {
      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 == true) {
        array[end] = array[i];
        flag = false;
      } else {
        array[++end] = array[i];
      }
    }
    return flag ? end : end + 1;
  }
}
```

Array Deduplication IV(unsorted array, repeatedly deduplication)

```
public class ArrayDeduplicationIV {
  public int dedup(int[] array) {
    // array is not null.
    int end = -1;
    for (int i = 0; i < array.length; i++) {
      if (end == -1 || array[end] != array[i]) {
        array[++end] = array[i];
      } else {
        while (i + 1 < array.length && array[i + 1] == array[end]) {
          i++;
        }
        end--;
      }
    }
    return end + 1;
  }
}
```

Largest And Smallest

```
public class LargestAndSmallest {
```

```
  public Pair largestAndSmallest(int[] array) {
    // array is not null & array.length >=1
    List<Integer> larger = new ArrayList<Integer>();
    List<Integer> smaller = new ArrayList<Integer>();
    for (int i = 0; i < array.length; i += 2) {
      if (i + 1 == array.length) {
        larger.add(array[i]);
        smaller.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]);
      }
    }
    return new Pair(largest(larger), smallest(smaller));
  }

  private int largest(List<Integer> larger) {
    int largest = larger.get(0);
    for (int num : larger) {
      if (num > largest) {
        largest = num;
      }
    }
    return largest;
  }

  private int smallest(List<Integer> smaller) {
    int smallest = smaller.get(0);
    for (int num : smaller) {
      if (num < smallest) {
        smallest = num;
      }
    }
    return smallest;
  }
}
```

Largest And Second Largest

```
public class LargestAndSecondLargest {
```

```java
public Pair largestAndSecond(int[] array) {
  // array is not null, array.length >= 2
  List<Pair> list = new ArrayList<Pair>();
  for (int i = 0; i < array.length; i++) {
    list.add(new Pair(i, array[i]));
  }
  HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();
  while (list.size() > 1) {
    List<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);
        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);
        }
      } else {
        nextRound.add(list.get(i));
      }
    }
    list = nextRound;
  }
  return new Pair(list.get(0).second, max(map.get(list.get(0).first)));
}

private int max(List<Integer> list) {
  int max = list.get(0);
  for (int num : list) {
    if (num > max) {
      max = num;
    }
  }
  return max;
```

```
  }
}
```

Spiral Order Print

```
public class SpiralPrintII {
  public List<Integer> spiral(int[][] matrix) {
    // Assumptions: matrix is not null, has size of M * N, where M, N >=0
    List<Integer> list = new ArrayList<Integer>();
    int m = matrix.length;
    if (m == 0) {
      return list;
    }
    int n = matrix[0].length;
    if (n == 0) {
      return list;
    }

    int left = 0;
    int right = n - 1;
    int up = 0;
    int down = m - 1;
    while (left < right && up < down) {
      for (int i = left; i <= right; i++) {
        list.add(matrix[up][i]);
      }
      for (int i = up + 1; i <= down - 1; i++) {
        list.add(matrix[i][right]);
      }
      for (int i = right; i >= left; i--) {
        list.add(matrix[down][i]);
      }
      for (int i = down - 1; i >= up + 1; i--) {
        list.add(matrix[i][left]);
      }
      left++;
      right--;
      up++;
      down--;
    }
    if (left > right || up > down) {
      return list;
    }
```

```
    if (left == right) {
      for (int i = up; i <= down; i++) {
        list.add(matrix[i][left]);
      }
    } else {
      for (int i = left; i <= right; i++) {
        list.add(matrix[up][i]);
      }
    }
    return list;
  }
}
```

Rotate Matrix By 90 Degree Clockwise

```
public class RotateMatrix {
  public void rotate(int[][] matrix) {
    // matrix is not null and N * N, N >= 0
    int n = matrix.length;
    if (n <= 1) {
      return;
    }
    int round = n / 2;
    for (int level = 0; level < round; level++) {
      int left = level;
      int right = n - 2 - level;
      for (int i = left; i <= right; i++) {
        int tmp = matrix[left][i];
        matrix[left][i] = matrix[n - 1 - i][left];
        matrix[n - 1 - i][left] = matrix[n - 1 - left][n - 1 - i];
        matrix[n - 1 - left][n - 1 - i] = matrix[i][n - 1 - left];
        matrix[i][n - 1 - left] = tmp;
      }
    }
  }
}
```

Zig-Zag Order Print Binary Tree

```
public class ZigZagLayerByLayer {
  public List<Integer> zigZag(TreeNode root) {
    // write your solution here
    if (root == null) {
```

```java
      return new LinkedList<Integer>();
    }
    Deque<TreeNode> deque = new LinkedList<TreeNode>();
    List<Integer> list = new LinkedList<Integer>();
    deque.offerFirst(root);
    int layer = 0;
    while (!deque.isEmpty()) {
      int sz = deque.size();
      for (int i = 0; i < sz; i++) {
        if (layer == 0) {
          TreeNode tmp = deque.pollLast();
          list.add(tmp.key);
          if (tmp.right != null) {
            deque.offerFirst(tmp.right);
          }
          if (tmp.left != null) {
            deque.offerFirst(tmp.left);
          }
        } else {
          TreeNode tmp = deque.pollFirst();
          list.add(tmp.key);
          if (tmp.left != null) {
            deque.offerLast(tmp.left);
          }
          if (tmp.right != null) {
            deque.offerLast(tmp.right);
          }
        }
      }
      layer = 1 - layer;
    }
    return list;
  }
}
```

Lowest Common Ancestor(without parent pointer)

```java
public class LCAI {
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode one, TreeNode two) {
    // root is not null, one and two guaranteed to be in the tree and not null
    if (root == null) {
      return null;
    }
```

```
    if (root == one || root == two) {
      return root;
    }
    TreeNode ll = lowestCommonAncestor(root.left, one, two);
    TreeNode lr = lowestCommonAncestor(root.right, one, two);
    if (ll != null && lr != null) {
      return root;
    } else if (ll != null) {
      return ll;
    } else {
      return lr;
    }
  }
}
```

Lowest Common Ancestor Of K Nodes

```
public class LCAIV {
  public TreeNode lowestCommonAncestor(TreeNode root, List<TreeNode> nodes) {
    // nodes not null, not empty, guaranteed to be in the tree.
    if (root == null) {
      return null;
    }
    for (TreeNode node : nodes) {
      if (root == node) {
        return root;
      }
    }
    TreeNode l = lowestCommonAncestor(root.left, nodes);
    TreeNode r = lowestCommonAncestor(root.right, nodes);
    if (l != null && r != null) {
      return root;
    }
    return l != null ? l : r;
  }
}
```

Lowest Common Ancestor(with parent pointer)

```
public class LCAII {
  public TreeNodeP lowestCommonAncestor(TreeNodeP one, TreeNodeP two) {
    int l1 = length(one);
    int l2 = length(two);
```

```
    if (l1 <= l2) {
      return llc(one, two, l2 - l1);
    } else {
      return llc(two, one, l1 - l2);
    }
  }

  private TreeNodeP llc(TreeNodeP small, TreeNodeP large, int diff) {
    while (diff > 0) {
      large = large.parent;
      diff--;
    }
    while (large != small) {
      large = large.parent;
      small = small.parent;
    }
    return large;
  }

  private int length(TreeNodeP node) {
    int length = 0;
    while (node != null) {
      length++;
      node = node.parent;
    }
    return length;
  }
}
```

[Back To Index]

Class 19 - 强化练习 II

Deep Copy Of List With Random Pointer

```
public class DeepCopyLinkedListRandom {
  public RandomListNode copy(RandomListNode head) {
    if (head == null) {
      return null;
    }
    RandomListNode dummy = new RandomListNode(0);
    RandomListNode cur = dummy;
```

```
    HashMap<RandomListNode, RandomListNode> map = new HashMap<RandomListNode,
RandomListNode>();
    while (head != null) {
      if (!map.containsKey(head)) {
        map.put(head, new RandomListNode(head.value));
      }
      cur.next = map.get(head);
      if (head.random != null) {
        if (!map.containsKey(head.random)) {
          map.put(head.random, new RandomListNode(head.random.value));
        }
        cur.next.random = map.get(head.random);
      }
      head = head.next;
      cur = cur.next;
    }
    return dummy.next;
  }
}
```

Deep Copy Of Graph(with possible 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));
    }
  }
}
```

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;
    }
  }
}
```

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;
  }
```

```java
  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;
    }
  }
}
```

Binary Search Tree Closest To Target

```java
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;
  }
}
```

Binary Search Tree Largest Number Smaller Than Target

```java
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;
  }
}
```

Binary Search Tree Delete

```
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;
  }
}
```

Wood Cut

```
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];
  }
}
```

Merge Stones

```
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];
  }
}
```

[Back To Index](#)

Class 20 - Midterm II

All Permutations(with duplicate characters)

```
public class PermutationsII {
  public List<String> permutations(String set) {
    List<String> result = new ArrayList<String>();
```

```
    if (set == null) {
      return result;
    }
    char[] array = set.toCharArray();
    helper(array, 0, result);
    return result;
  }

  private void helper(char[] array, int index, List<String> result) {
    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.add(array[i])) {
        swap(array, i, index);
        helper(array, index + 1, result);
        swap(array, i, index);
      }
    }
  }

  private void swap(char[] array, int left, int right) {
    char tmp = array[left];
    array[left] = array[right];
    array[right] = tmp;
  }
}
```

Max Path Sum From One Leaf Node To Another In Binary Tree

```
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;
  }
}
```

Min Cuts Of Palindrome Partitions

```
public class MinimumCutsPalindromes {
  public int minCuts(String input) {
    // input is not null.
    char[] array = input.toCharArray();
    int len = array.length;
    if (len == 0) {
      return 0;
    }
    boolean[][] isP = new boolean[len + 1][len + 1];
    int[] minCuts = new int[len + 1];
    for (int end = 1; end <= len; end++) {
      minCuts[end] = end;
      for (int start = end; start >= 1; start--) {
        if (start == end) {
          isP[start][end] = true;
        } else if (start == end - 1) {
          isP[start][end] = array[start - 1] == array[end - 1];
        } else {
          isP[start][end] = array[start - 1] == array[end - 1] ? isP[start + 1][end - 1] : false;
        }
        if (isP[start][end]) {
          minCuts[end] = Math.min(minCuts[end], 1 + minCuts[start - 1]);
        }
      }
    }
```

```
    return minCuts[len] - 1;
  }
}
```

Valid If Blocks

```
class Solution {
  public void validIfBlocks(int n) {
    if (n <= 0) {
      return;
    }
    List<String> blocks = new ArrayList<String>();
    helper(blocks, n, n);
  }

  private void helper(List<String> blocks, int left, int right) {
    if (left == 0 && right == 0) {
      print(blocks);
      return;
    }
    StringBuilder builder = new StringBuilder();
    if (left > 0) {
      for (int i = 0; i < right - left; i++) {
        builder.append("  ");
      }
      blocks.add(builder.append("if {").toString());
      helper(blocks, left - 1, right);
      blocks.remove(blocks.size() - 1);
    }
    builder.setLength(0);
    if (right > left) {
      for (int i = 0; i < right - left - 1; i++) {
        builder.append("  ");
      }
      blocks.add(builder.append("}").toString());
      helper(blocks, left, right - 1);
      blocks.remove(blocks.size() - 1);
    }
  }

  private void print(List<String> blocks) {
    for (String s : blocks) {
      System.out.println(s);
```

```java
    }
    System.out.println("=================================");
  }
}
```

[Back To Index](#)

Class 21 - 强化练习 III

Determine If Binary Tree Is Balanced

```java
public class CheckBalanced {
  public boolean isBalanced(TreeNode root) {
    if (root == null) {
      return true;
    }
    // use -1 to denote the tree is not balanced.
    // >= 0 value means the tree is balanced and it is the height of the tree.
    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;
  }
}
```

Max Path Sum Binary Tree II(path from any node to any node)

```java
public class MaxPathSumBinaryTreeII {
  public int maxPathSum(TreeNode root) {
```

```java
    // root is not null.
    return helper(root).dou;
  }

  private Result helper(TreeNode root) {
    if (root == null) {
      return new Result(0, Integer.MIN_VALUE);
    }
    Result left = helper(root.left);
    Result right = helper(root.right);
    int sin = Math.max(root.key, Math.max(root.key + left.sin, root.key + right.sin));
    int dou = Math.max(sin, root.key + left.sin + right.sin);
    left.sin = sin;
    left.dou = Math.max(dou, Math.max(left.dou, right.dou));
    return left;
  }

  static class Result {
    int sin;
    int dou;

    Result(int sin, int dou) {
      this.sin = sin;
      this.dou = dou;
    }
  }
}
```

Max Path Sum Binary Tree(path from leaf to root)

```java
public class MaxPathSumLeafToRoot {
  // root != null
  public int maxPathSum(TreeNode root) {
    return maxPathSum(root, 0);
  }

  private int maxPathSum(TreeNode root, int sum) {
    sum += root.key;
    if (root.left == null && root.right == null) {
      return sum;
    } else if (root.left == null) {
      return maxPathSum(root.right, sum);
    } else if (root.right == null) {
```

```
    return maxPathSum(root.left, sum);
  }
  return Math.max(maxPathSum(root.left, sum), maxPathSum(root.right, sum));
 }
}
```

Binary Tree Path Sum To Target(the two nodes can be the same node and they can only be on the path from root to one of the leaf nodes)

```
public class BinaryTreePathSumToTarget {
 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;
 }
}
```

Max Path Sum Binary Tree III(the two nodes can be the same node and they can only be on the path from root to one of the leaf nodes)

```
public class MaxPathSumBinaryTreeIII {
 private int max = Integer.MIN_VALUE;

 public int maxPathSum(TreeNode root) {
  // root is not null.
  max = Integer.MIN_VALUE;
  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 sin = Math.max(root.key, Math.max(root.key + left, root.key + right));
  max = Math.max(max, sin);
  return sin;
 }
}
```

Reconstruct Binary Tree With Preorder And Inorder

```
public class ReconstructBTInPre {
 private int pi;
 private int ii;
 public TreeNode reconstruct(int[] pre, int[] in) {
  // pre, in not null, no duplicates, length equals valid.
  pi = 0;
  ii = 0;
  return helper(pre, in, Integer.MAX_VALUE);
 }

 private TreeNode helper(int[] pre, int[] in, int target) {
  if (ii >= in.length || in[ii] == target) {
   return null;
  }
  TreeNode root = new TreeNode(pre[pi]);
  pi++;
```

```
    root.left = helper(pre, in, root.key);
    ii++;
    root.right = helper(pre, in, target);
    return root;
  }
}
```

Reconstruct Binary Search Tree With Postorder

```
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;
  }
}
```

Reconstruct Binary Tree With Levelorder And Inorder

```
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;
  }
}
```

[Back To Index]

Class 23 - 强化练习 IV

Reverse Binary Tree Upside Down

```
public class BinaryTreeUpsideDown {
  // Method 1: Recursion
  public TreeNode reverse(TreeNode root) {
    if (root == null || root.left == null) {
      return root;
    }
    TreeNode newRoot = reverse(root.left);
    root.left.right = root.right;
    root.left.left = root;
    root.left = null;
    root.right = null;
    return newRoot;
  }

  // Method 2: Iterative
  public TreeNode reverseI(TreeNode root) {
    TreeNode prev = null;
```

```
    TreeNode prevRight = null;
    while (root != null) {
      TreeNode next = root.left;
      TreeNode right = root.right;
      root.right = prevRight;
      root.left = prev;
      prevRight = right;
      prev = root;
      root = next;
    }
    return prev;
  }
}
```

All Valid Permutations Of Parentheses II(L pairs of (), M pairs of [], N pairs of{})

```
public class ValidParenthesesII {
  private static final char[] PS = new char[] { '(', ')', '[', ']', '{', '}' };

  public List<String> validParentheses(int l, int m, int n) {
    // l, m, n >= 0
    int[] remain = new int[] { l, l, m, m, n, n };
    int targetLen = 2 * l + 2 * m + 2 * n;
    StringBuilder cur = new StringBuilder();
    Deque<Character> stack = new LinkedList<Character>();
    List<String> result = new ArrayList<String>();
    helper(cur, stack, remain, targetLen, result);
    return result;
  }

  private void helper(StringBuilder cur, Deque<Character> stack, int[] remain, int targetLen,
      List<String> result) {
    if (cur.length() == targetLen) {
      result.add(cur.toString());
      return;
    }
    for (int i = 0; i < remain.length; i++) {
      if (i % 2 == 0) {
        if (remain[i] > 0) {
          cur.append(PS[i]);
          stack.offerFirst(PS[i]);
          remain[i]--;
          helper(cur, stack, remain, targetLen, result);
```

```
        cur.deleteCharAt(cur.length() - 1);
        stack.pollFirst();
        remain[i]++;
      }
    } else {
      if (!stack.isEmpty() && stack.peekFirst() == PS[i - 1]) {
        cur.append(PS[i]);
        stack.pollFirst();
        remain[i]--;
        helper(cur, stack, remain, targetLen, result);
        cur.deleteCharAt(cur.length() - 1);
        stack.offerFirst(PS[i - 1]);
        remain[i]++;
      }
    }
   }
  }
}
```

N Queens

```
public class NQueens {
  public List<List<Integer>> nqueens(int n) {
    // n > 0
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> cur = new ArrayList<Integer>();
    helper(0, n, cur, result);
    return result;
  }

  private void helper(int index, int n, List<Integer> cur, List<List<Integer>> result) {
    if (index == n) {
      result.add(new ArrayList<Integer>(cur));
      return;
    }
    for (int i = 0; i < n; i++) {
      cur.add(i);
      if (valid(cur)) {
        helper(index + 1, n, cur, result);
      }
      cur.remove(cur.size() - 1);
    }
  }
```

```
  private boolean valid(List<Integer> cur) {
    int size = cur.size();
    for (int i = 0; i < size - 1; i++) {
      if (cur.get(i).equals(cur.get(size - 1)) || cur.get(i) - cur.get(size - 1) == i + 1 - size
          || cur.get(i) - cur.get(size - 1) == size - 1 - i) {
        return false;
      }
    }
    return true;
  }
}
```

All Subsequences Of Sorted String

```
public class SubSetsII {
  public List<String> subSets(String set) {
    List<String> result = new ArrayList<String>();
    if (set == null) {
      return result;
    }
    char[] arraySet = set.toCharArray();
    Arrays.sort(arraySet);
    StringBuilder sb = new StringBuilder();
    helper(arraySet, sb, 0, result);
    return result;
  }

  private void helper(char[] set, StringBuilder sb, int index, List<String> result) {
    result.add(sb.toString());
    for (int i = index; i < set.length; i++) {
      if (i == index || set[i] != set[i - 1]) {
        sb.append(set[i]);
        helper(set, sb, i + 1, result);
        sb.deleteCharAt(sb.length() - 1);
      }
    }
  }
}
```

Two Sum

```
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;
  }
}
```

Three Sum

```
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;
  }
}
```

Four Sum

```
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;
  Arrays.sort(array);
  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
```

```java
      // 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;
  }
}
```

Back To Index

Class 24 - 强化练习 V

Common Elements In Three Sorted Arrays

```
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;
  }
}
```

一个字典有给一系列strings， 要求找两个string,使得它们没有共同字符，并且长度乘积最大. (Assumption: all letters in the word is from ‘a-z’ in ASCII)

```
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;
  }
}
```

How to find the k-th smallest number in the f(x,y,z) = 3^x * 5^y * 7^z (int 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;
  }
}
```

Kth Closest Point To <0,0,0>

```
public class KthClosestPoint {
  public List<Integer> closest(final int[] a, final int[] b, final int[] c, int k) {
    // a, b, c is not null and length >= 1, k >= 1 && k <= a.length * b.length *
    // c.length
    PriorityQueue<List<Integer>> minHeap = new PriorityQueue<List<Integer>>(2 * k,
        new Comparator<List<Integer>>() {
          @Override
          public int compare(List<Integer> o1, List<Integer> o2) {
            long d1 = distance(o1, a, b, c);
            long d2 = distance(o2, a, b, c);
            if (d1 == d2) {
              return 0;
            }
            return d1 < d2 ? -1 : 1;
          }
        });
    HashSet<List<Integer>> visited = new HashSet<List<Integer>>();
    List<Integer> cur = Arrays.asList(0, 0, 0);
    visited.add(cur);
```

```
    minHeap.offer(cur);
    while (k > 0) {
      cur = minHeap.poll();
      List<Integer> n = Arrays.asList(cur.get(0) + 1, cur.get(1), cur.get(2));
      if (n.get(0) < a.length && visited.add(n)) {
        minHeap.offer(n);
      }
      n = Arrays.asList(cur.get(0), cur.get(1) + 1, cur.get(2));
      if (n.get(1) < b.length && visited.add(n)) {
        minHeap.offer(n);
      }
      n = Arrays.asList(cur.get(0), cur.get(1), cur.get(2) + 1);
      if (n.get(2) < c.length && visited.add(n)) {
        minHeap.offer(n);
      }
      k--;
    }
    cur.set(0, a[cur.get(0)]);
    cur.set(1, b[cur.get(1)]);
    cur.set(2, c[cur.get(2)]);
    return cur;
  }

  private long distance(List<Integer> point, int[] a, int[] b, int[] c) {
    long dis = 0;
    dis += a[point.get(0)] * a[point.get(0)];
    dis += b[point.get(1)] * b[point.get(1)];
    dis += c[point.get(2)] * c[point.get(2)];
    return dis;
  }
}
```

Place To Put Chair I

```
/**
 * Given a gym with k equipments, and some obstacles.
 * Lets 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.
```

```java
 * 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;
  }
}
```

Largest Rectangle In Histogram

```
public class LargestRectangleHistogram {
  public int largest(int[] array) {
    // array is not null, length >= 1, non-negative
    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;
  }
}
```

Max Water Trapped

```
public class MaxWaterTrappedI {
  public int maxTrapped(int[] array) {
    // array is not null,
    if (array.length == 0) {
      return 0;
```

```
    }
    int left = 0;
    int right = array.length - 1;
    int result = 0;
    int lmax = array[left];
    int rmax = array[right];
    while (left < right) {
      if (array[left] <= array[right]) {
        result += Math.max(0, lmax - array[left]);
        lmax = Math.max(lmax, array[left]);
        left++;
      } else {
        result += Math.max(0, rmax - array[right]);
        rmax = Math.max(rmax, array[right]);
        right--;
      }
    }
    return result;
  }
}
```

Max Water Trapped II

```
public class MaxWaterTrappedII {
  public int maxTrapped(final int[][] matrix) {
    // matrix is not null, M * N, M > 0 & N > 0, non-negative integers.
    int M = matrix.length;
    int N = matrix[0].length;
    if (M < 3 || N < 3) {
      return 0;
    }

    PriorityQueue<Pair> minHeap = new PriorityQueue<Pair>();
    boolean[][] visited = new boolean[M][N];
    for (int j = 0; j < N; j++) {
      minHeap.offer(new Pair(0, j, matrix[0][j]));
      minHeap.offer(new Pair(M - 1, j, matrix[M - 1][j]));
      visited[0][j] = true;
      visited[M - 1][j] = true;
    }

    for (int i = 1; i < M - 1; i++) {
      minHeap.offer(new Pair(i, 0, matrix[i][0]));
```

```
    minHeap.offer(new Pair(i, N - 1, matrix[i][N - 1]));
    visited[i][0] = true;
    visited[i][N - 1] = true;
  }

  int result = 0;
  while (!minHeap.isEmpty()) {
    Pair cur = minHeap.poll();
    System.out.println(cur.x + " " + cur.y);
    visited[cur.x][cur.y] = true;
    result += DFS(cur, matrix, visited, minHeap, matrix[cur.x][cur.y]);
  }
  return result;
}

private int DFS(Pair cur, int[][] matrix, boolean[][] visited, PriorityQueue<Pair> minHeap,
    int height) {
  List<Pair> neis = neis(cur, visited);
  int result = height - matrix[cur.x][cur.y];
  for (Pair nei : neis) {
    if (!visited[nei.x][nei.y]) {
      nei.height = matrix[nei.x][nei.y];
      visited[nei.x][nei.y] = true;
      if (matrix[nei.x][nei.y] < height) {
        result += DFS(nei, matrix, visited, minHeap, height);
      } else {
        minHeap.offer(nei);
      }
    }
  }
  return result;
}

private List<Pair> neis(Pair cur, boolean[][] visited) {
  List<Pair> neis = new ArrayList<Pair>();
  if (cur.x + 1 < visited.length) {
    neis.add(new Pair(cur.x + 1, cur.y, 0));
  }
  if (cur.x - 1 >= 0) {
    neis.add(new Pair(cur.x - 1, cur.y, 0));
  }
  if (cur.y + 1 < visited[0].length) {
    neis.add(new Pair(cur.x, cur.y + 1, 0));
```

```
    }
    if (cur.y - 1 >= 0) {
      neis.add(new Pair(cur.x, cur.y - 1, 0));
    }
    return neis;
  }

  static class Pair implements Comparable<Pair> {
    int x;
    int y;
    int height;

    Pair(int x, int y, int height) {
      this.x = x;
      this.y = y;
      this.height = height;
    }

    @Override
    public int compareTo(Pair another) {
      if (this.height == another.height) {
        return 0;
      }
      return this.height < another.height ? -1 : 1;
    }
  }
}
```

[Back To Index]