**Home Work Day 1**

1.

class Solution {

public int[] sortArray(int[] nums) {

if (nums == null || nums.length <= 1) {

return nums;

}

mergeSort(nums, 0, nums.length - 1);

return nums;

}

private void mergeSort(int[] nums, int low, int high) {

if (low < high) {

int mid = low + (high - low) / 2;

mergeSort(nums, low, mid);

mergeSort(nums, mid + 1, high);

merge(nums, low, mid, high);

}

}

private void merge(int[] nums, int low, int mid, int high) {

int n1 = mid - low + 1;

int n2 = high - mid;

int[] left = new int[n1];

int[] right = new int[n2];

for (int i = 0; i < n1; ++i) {

left[i] = nums[low + i];

}

for (int j = 0; j < n2; ++j) {

right[j] = nums[mid + 1 + j];

}

int i = 0, j = 0, k = low;

while (i < n1 && j < n2) {

if (left[i] <= right[j]) {

nums[k++] = left[i++];

} else {

nums[k++] = right[j++];

}

}

while (i < n1) {

nums[k++] = left[i++];

}

while (j < n2) {

nums[k++] = right[j++];

}

}

}

2.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode sortList(ListNode head) {

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

return head;

}

ListNode middle = findMiddle(head);

ListNode nextToMiddle = middle.next;

middle.next = null;

ListNode left = sortList(head);

ListNode right = sortList(nextToMiddle);

return merge(left, right);

}

private ListNode findMiddle(ListNode head) {

if (head == null) {

return null;

}

ListNode slow = head;

ListNode fast = head.next;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

private ListNode merge(ListNode left, ListNode right) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (left != null && right != null) {

if (left.val < right.val) {

current.next = left;

left = left.next;

} else {

current.next = right;

right = right.next;

}

current = current.next;

}

if (left != null) {

current.next = left;

}

if (right != null) {

current.next = right;

}

return dummy.next;

}

}

3.

class Solution {

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

int left = 0;

int right = nums.length - 1;

while (left <= right) {

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

if (nums[mid] == target) {

return mid;

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

}

4.

class Solution {

void pushZerosToEnd(int[] arr, int n) {

int nonZeroIndex = 0;

// Traverse the array and move non-zero elements to the front

for (int i = 0; i < n; i++) {

if (arr[i] != 0) {

arr[nonZeroIndex++] = arr[i];

}

}

// Fill the remaining positions with zeros

for (int i = nonZeroIndex; i < n; i++) {

arr[i] = 0;

}

}

}

5.

class Solution {

public int longestPalindromeSubseq(String s) {

int n = s.length();

// Create a 2D array to store the length of the longest palindromic subsequence

int[][] dp = new int[n][n];

// Every individual character is a palindrome of length 1

for (int i = 0; i < n; i++) {

dp[i][i] = 1;

}

// Loop to fill the dp array

for (int len = 2; len <= n; len++) {

for (int i = 0; i <= n - len; i++) {

int j = i + len - 1;

if (s.charAt(i) == s.charAt(j)) {

dp[i][j] = 2 + dp[i + 1][j - 1];

} else {

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

}

}

}

// The length of the longest palindromic subsequence in the entire string

return dp[0][n - 1];

}

}

6.

class Solution {

public int maxProfit(int k, int[] prices) {

int n = prices.length;

// If k is large enough, it becomes a general stock transaction problem

if (k >= n / 2) {

int maxProfit = 0;

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

if (prices[i] > prices[i - 1]) {

maxProfit += prices[i] - prices[i - 1];

}

}

return maxProfit;

}

// Use dynamic programming to solve the problem

int[][] buy = new int[n][k + 1];

int[][] sell = new int[n][k + 1];

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

buy[0][j] = -prices[0];

sell[0][j] = 0;

}

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

buy[i][0] = Math.max(buy[i - 1][0], sell[i - 1][0] - prices[i]);

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

buy[i][j] = Math.max(buy[i - 1][j], sell[i - 1][j] - prices[i]);

sell[i][j] = Math.max(sell[i - 1][j], buy[i - 1][j - 1] + prices[i]);

}

}

return sell[n - 1][k];

}

}

**Home Work Day 2:**