1. Two Sum

```go
func twoSum(nums []int, target int) []int {
    mp := make(map[int]int)
    for i := range nums {
        remaining := target - nums[i]
        if idx, ok := mp[remaining]; ok {
            return []int{idx, i}
        }
        mp[nums[i]] = i
    }
    return []int{}
}
```

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> mp;
        for (int i = 0; i < nums.size(); i++) {
            int remaining = target - nums[i];
            if (mp.contains(remaining)) {
                return {mp.at(remaining), i};
            }
            mp[nums[i]] = i;
        }
        return {};
    }
};
```

2. Add Two Numbers

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := &ListNode{}
    tail := dummy
    carry := 0
    for l1 != nil || l2 != nil || carry != 0 {
        v1, v2 := 0, 0
        if l1 != nil {
            v1 = l1.Val
            l1 = l1.Next
        }
        if l2 != nil {
            v2 = l2.Val
            l2 = l2.Next
        }
        sum := v1 + v2 + carry
        carry = sum / 10
        tail.Next = &ListNode{Val: sum % 10}
        tail = tail.Next
    }
    return dummy.Next
}
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode* next) : val(x), next(next) {}
 * };

 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode();
        ListNode* tail = dummy;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr || carry != 0) {
            int v1 = 0;
            int v2 = 0;
            if (l1 != nullptr) {
                v1 = l1->val;
                l1 = l1->next;
            }
            if (l2 != nullptr) {
                v2 = l2->val;
                l2 = l2->next;
            }
            int sum = v1 + v2 + carry;
            carry = sum / 10;
            tail->next = new ListNode(sum % 10);
            tail = tail->next;
        }
        return dummy->next;
    }
};
```

3. Longest Substring Without Repeating Characters

```
func lengthOfLongestSubstring(s string) int {
    mp := make(map[byte]int)
    left := 0
    ans := 0
    for i := range s {
        if idx, ok := mp[s[i]]; ok {
            left = max(left, idx+1)
        }
        mp[s[i]] = i
        ans = max(ans, i-left+1)
    }
    return ans
}
```

```
class Solution {
```

```
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<int, int> mp;
        int left = 0;
        int ans = 0;
        for (int i = 0; i < s.length(); i++) {
            if (mp.contains(s[i])) {
                left = max(left, mp.at(s[i]) + 1);
            }
            mp[s[i]] = i;
            ans = max(ans, i - left + 1);
        }
        return ans;
    }
};
```

4. Median of Two Sorted Arrays

```
func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    sz := len(nums1) + len(nums2)
    if sz % 2 == 1 {
        return float64(getKthElement(nums1, nums2, sz/2 + 1))
    } else {
        return float64(getKthElement(nums1, nums2, sz/2) + getKthElement(nums1,
nums2, sz/2 + 1)) / 2.0
    }
}

func getKthElement(nums1 []int, nums2 []int, k int) int {
    index1, index2 := 0, 0
    for {
        if index1 == len(nums1) {
            return nums2[index2+k-1]
        }
        if index2 == len(nums2) {
            return nums1[index1+k-1]
        }
        if k == 1 {
            return min(nums1[index1], nums2[index2])
        }
```

```go
        newIndex1 := min(index1 + k/2, len(nums1)) - 1
        newIndex2 := min(index2 + k/2, len(nums2)) - 1
        if nums1[newIndex1] < nums2[newIndex2] {
            k -= newIndex1 - index1 + 1
            index1 = newIndex1 + 1
        } else {
            k -= newIndex2 - index2 + 1
            index2 = newIndex2 + 1
        }
    }
    return 0
}
```

5. Longest Palindromic Substring

```go
func longestPalindrome(s string) string {
    p, q := 0, 0
    for i := range s {
        p1, q1 := expandAroundCenter(s, i, i)
        p2, q2 := expandAroundCenter(s, i, i+1)
        if q1-p1 > q-p {
            p = p1
            q = q1
        }
        if q2-p2 > q-p {
            p = p2
            q = q2
        }
    }
    return s[p : q+1]
}

func expandAroundCenter(s string, i int, j int) (int, int) {
    for i >= 0 && j < len(s) && s[i] == s[j] {
        i--
        j++
    }
    return i + 1, j - 1
}
```

```
class Solution {
public:
    string longestPalindrome(string s) {
        int p = 0;
        int q = 0;
        for (int i = 0; i < s.length(); i++) {
            auto [p1, q1] = expandAroundCenter(s, i, i);
            auto [p2, q2] = expandAroundCenter(s, i, i + 1);
            if (q1 - p1 > q - p) {
                p = p1;
                q = q1;
            }
            if (q2 - p2 > q - p) {
                p = p2;
                q = q2;
            }
        }
        return s.substr(p, q - p + 1);
    }

    tuple<int, int> expandAroundCenter(string& s, int i, int j) {
        while (i >= 0 && j < s.length() && s[i] == s[j]) {
            i--;
            j++;
        }
        return {i + 1, j - 1};
    }
};
```

8. String to Integer (atoi)

```
func myAtoi(s string) int {
    if len(s) == 0 {
        return 0
    }

    i := 0
    for i < len(s) && s[i] == ' ' {
        i++
    }
    if i == len(s) {
        return 0
    }

    sign := 1
    if s[i] == '-' {
        sign = -1
        i++
    } else if s[i] == '+' {
        sign = 1
        i++
    }

    ans := 0
    for i < len(s) && s[i] >= '0' && s[i] <= '9' {
        digit := int(s[i] - '0') * sign
        if ans > math.MaxInt32 / 10 || (ans == math.MaxInt32 / 10 && digit > 
math.MaxInt32 % 10) {
            return math.MaxInt32
        }
        if ans < math.MinInt32 / 10 || (ans == math.MinInt32 / 10 && digit < 
math.MinInt32 % 10) {
            return math.MinInt32
        }
        ans = 10 * ans + digit
        i++
    }

    return ans
}
```

11. Container With Most Water

```
func maxArea(height []int) int {
    i, j := 0, len(height) - 1
    ans := 0
    for i < j {
        area := min(height[i], height[j]) * (j - i)
```

```go
        ans = max(ans, area)
        if height[i] <= height[j] {
            i++
        } else {
            j--
        }
    }
    return ans
}
```

15. 3Sum

```go
func threeSum(nums []int) [][]int {
    n := len(nums)
    sort.Ints(nums)
    ans := [][]int{}
    for i := 0; i < n; i++ {
        if i > 0 && nums[i] == nums[i-1] {
            continue
        }
        k := n - 1
        for j := i + 1; j < n; j++ {
            if j > i+1 && nums[j] == nums[j-1] {
                continue
            }
            for j < k && nums[i]+nums[j]+nums[k] > 0 {
                k--
            }
            if j == k {
                break
            }
            if nums[i]+nums[j]+nums[k] == 0 {
                ans = append(ans, []int{nums[i], nums[j], nums[k]})
            }
        }
    }
    return ans
}
```

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(), nums.end());
        vector<vector<int>> ans;
        for (int i = 0; i < n; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            int k = n - 1;
            for (int j = i + 1; j < n; j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                while (j < k && nums[i] + nums[j] + nums[k] > 0) {
                    k--;
                }
                if (j == k) {
                    break;
                }
                if (nums[i] + nums[j] + nums[k] == 0) {
                    ans.push_back({nums[i], nums[j], nums[k]});
                }
            }
        }
        return ans;
    }
};
```

19. Remove Nth Node From End of List

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
```

```go
func removeNthFromEnd(head *ListNode, n int) *ListNode {
    dummy := &ListNode{Next: head}
    left, right := dummy, head
    for i := 0; i < n; i++ {
        right = right.Next
    }
    for right != nil {
        right = right.Next
        left = left.Next
    }
    left.Next = left.Next.Next
    return dummy.Next
}
```

20. Valid Parentheses

```go
func isValid(s string) bool {
    stack := []byte{}
    for i := range s {
        if s[i] == '(' {
            stack = append(stack, ')')
        } else if s[i] == '[' {
            stack = append(stack, ']')
        } else if s[i] == '{' {
            stack = append(stack, '}')
        } else {
            if len(stack) == 0 || stack[len(stack)-1] != s[i] {
                return false
            }
            stack = stack[:len(stack)-1]
        }
    }
    return len(stack) == 0
}
```

21. Merge Two Sorted Lists

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeTwoLists(list1 *ListNode, list2 *ListNode) *ListNode {
    dummy := &ListNode{}
    tail := dummy
    for list1 != nil && list2 != nil {
        if list1.Val < list2.Val {
            tail.Next = list1
            list1 = list1.Next
        } else {
            tail.Next = list2
            list2 = list2.Next
        }
        tail = tail.Next
    }
    if list1 == nil {
        tail.Next = list2
    } else {
        tail.Next = list1
    }
    return dummy.Next
}
```

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeTwoLists(list1 *ListNode, list2 *ListNode) *ListNode {
    if list1 == nil {
        return list2
    }
    if list2 == nil {
        return list1
    }
    if list1.Val < list2.Val {
        list1.Next = mergeTwoLists(list1.Next, list2)
        return list1
    } else {
        list2.Next = mergeTwoLists(list1, list2.Next)
        return list2
    }
}
```

22. Generate Parentheses

```
func generateParenthesis(n int) []string {
    res := []string{}
    stack := []byte{}

    var backtrack func(int, int)

    backtrack = func(openN int, closedN int) {
        if openN == closedN && openN == n {
            res = append(res, string(stack))
            return
        }

        if openN < n {
            stack = append(stack, '(')
            backtrack(openN + 1, closedN)
            stack = stack[:len(stack)-1]
        }

        if closedN < openN {
            stack = append(stack, ')')
            backtrack(openN, closedN + 1)
            stack = stack[:len(stack)-1]
        }
    }
    backtrack(0, 0)
    return res
}
```

23. Merge k Sorted Lists

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeKLists(lists []*ListNode) *ListNode {
    return merge(lists, 0, len(lists) - 1)
}

func merge(lists []*ListNode, l int, r int) *ListNode {
    if l == r {
        return lists[l]
    }
    if l > r {
        return nil
    }
    mid := (l + r) / 2
    return mergeTwoLists(merge(lists, l, mid), merge(lists, mid + 1, r))
}

func mergeTwoLists(a *ListNode, b *ListNode) *ListNode {
    dummy := &ListNode{}
    tail := dummy
    for a != nil && b != nil {
        if a.Val < b.Val {
            tail.Next = a
            a = a.Next
        } else {
            tail.Next = b
            b = b.Next
        }
        tail = tail.Next
    }
    if a == nil {
        tail.Next = b
    } else {
        tail.Next = a
    }
    return dummy.Next
}
```

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        return merge(lists, 0, lists.size() - 1);
    }

    ListNode* merge(vector<ListNode*>& lists, int l, int r) {
        if (l == r) {
            return lists[l];
        }
        if (l > r) {
            return nullptr;
        }
        int mid = (l + r) / 2;
        return mergeTwoLists(merge(lists, l, mid), merge(lists, mid + 1, r));
    }

    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode* dummy = new ListNode();
        ListNode* tail = dummy;
        while (a != nullptr && b != nullptr) {
            if (a->val < b->val) {
                tail->next = a;
                a = a->next;
            } else {
                tail->next = b;
                b = b->next;
            }
            tail = tail->next;
        }
        if (a == nullptr) {
            tail->next = b;
        } else {
            tail->next = a;
        }
        return dummy->next;
    }
};
```

24. Swap Nodes in Pairs

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func swapPairs(head *ListNode) *ListNode {
    dummy := &ListNode{Next: head}
    prev, curr := dummy, head

    for curr != nil && curr.Next != nil {
        // save pointers
        nextPair := curr.Next.Next
        second := curr.Next

        // reverse this pair
        second.Next = curr
        curr.Next = nextPair
        prev.Next = second
        // update pointers
        prev = curr
        curr = nextPair
    }

    return dummy.Next
}
```

25. Reverse Nodes in k-Group

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseKGroup(head *ListNode, k int) *ListNode {
    dummy := &ListNode{Next: head}
    groupPrev := dummy

    for {
        kth := getKth(groupPrev, k)
        if kth == nil {
            break
        }
        groupNext := kth.Next

        // reverse current group
        prev := groupNext
        curr := groupPrev.Next
        for curr != groupNext {
            tmp := curr.Next
            curr.Next = prev
            prev = curr
            curr = tmp
        }

        tmp := groupPrev.Next
```

```go
        groupPrev.Next = kth
        groupPrev = tmp
    }

    return dummy.Next
}

func getKth(curr *ListNode, k int) *ListNode {
    for curr != nil && k > 0 {
        curr = curr.Next
        k--
    }
    return curr
}
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode(): val(0), next(nullptr) {}
 *     ListNode(int x): val(x), next(nullptr) {}
 *     ListNode(int x, ListNode* next): val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummy = new ListNode(0, head);
        ListNode* predecessor = dummy;
        ListNode* tail = dummy;

        while (true) {
            for (int i = 0; i < k; i++) {
                tail = tail->next;
                if (tail == nullptr) {
                    return dummy->next;
                }
            }
            ListNode* head = predecessor->next;
            ListNode* successor = tail->next;
            tail->next = nullptr;
            predecessor->next = reverseList(head);
            head->next = successor;
            predecessor = head;
            tail = head;
        }

        return nullptr;
    }

private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

31. Next Permutation

```
func nextPermutation(nums []int)  {
    n := len(nums)
    i := n - 2

    for i >= 0 && nums[i] >= nums[i+1] {
        i--
    }

    if i >= 0 {
        j := n - 1
        for j >= 0 && nums[j] <= nums[i] {
            j--
        }
        nums[i], nums[j] = nums[j], nums[i]
    }
```

```go
    reverse(nums, i + 1, n - 1)
}

func reverse(nums []int, i int, j int) {
    for i < j {
        nums[i], nums[j] = nums[j], nums[i]
        i++
        j--
    }
}
```

```cpp
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int n = nums.size();

        int i = n - 2;
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        if (i >= 0) {
            int j = n - 1;
            while (j >= 0 && nums[j] <= nums[i]) {
                j--;
            }
            swap(nums[i], nums[j]);
        }

        reverse(nums, i + 1, n - 1);
    }

    void reverse(vector<int>& nums, int i, int j) {
        while (i < j) {
            swap(nums[i++], nums[j--]);
        }
    }
};
```

32. Longest Valid Parentheses

```go
func longestValidParentheses(s string) int {
    ans := 0
    stack := []int{}
    stack = append(stack, -1)
    for i := range s {
        if s[i] == '(' {
            stack = append(stack, i)
        } else {
            stack = stack[:len(stack)-1]
            if len(stack) == 0 {
                stack = append(stack, i)
            } else {
                ans = max(ans, i - stack[len(stack)-1])
            }
        }
    }
    return ans
}
```

33. Search in Rotated Sorted Array

```go
func search(nums []int, target int) int {
    l, r := 0, len(nums) - 1

    for l <= r {
        mid := (l + r) / 2
        if target == nums[mid] {
            return mid
        }

        if nums[l] <= nums[mid] { // nums[l:mid+1]有序
            if target >= nums[l] && target < nums[mid] { // 可以判断target是否在
nums[l:mid+1]
                r = mid - 1
            } else {
                l = mid + 1
            }
        } else { // nums[mid+1:r+1]有序
            if target > nums[mid] && target <= nums[r] { // 可以判断target是否在
nums[mid+1:r+1]
                l = mid + 1
            } else {
                r = mid - 1
            }
        }
    }

    return -1
}
```

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n = nums.size();
        int left = 0;
        int right = n - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) {
                return mid;
            }
            if (nums[left] <= nums[mid]) { // [left, mid] 有序
                // 可以判断 target 是否在 [left, mid) 区间
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else { // [mid, right] 有序
                // 可以判断 target 是否在 (mid, right] 区间
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }
        return -1;
    }
};
```

39. Combination Sum

```go
func combinationSum(candidates []int, target int) [][]int {
    res := [][]int{}

    var dfs func(int, []int, int)
    dfs = func(i int, curr []int, total int) {
        if total == target {
            res = append(res, append([]int{}, curr...))
            return
        }
        if i >= len(candidates) || total > target {
            return
        }

        curr = append(curr, candidates[i])
        dfs(i, curr, total + candidates[i])
        curr = curr[:len(curr)-1]
        dfs(i + 1, curr, total)
    }

    dfs(0, []int{}, 0)
    return res
}
```

41. First Missing Positive

```go
func firstMissingPositive(nums []int) int {
    n := len(nums)

    for i := range nums {
        if nums[i] <= 0 {
            nums[i] = n + 1
        }
    }

    for i := range nums {
        x := abs(nums[i])
        if x <= n {
            nums[x - 1] = -abs(nums[x - 1])
        }
    }

    for i := range nums {
        if nums[i] > 0 {
            return i + 1
        }
    }
    return n + 1
}

func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
```

42. Trapping Rain Water

```
func trap(height []int) int {
    l, r := 0, len(height) - 1
    leftMax, rightMax := height[l], height[r]
    res := 0

    for l < r {
        if leftMax <= rightMax {
            l++
            leftMax = max(leftMax, height[l])
            res += leftMax - height[l]
        } else {
            r--
            rightMax = max(rightMax, height[r])
            res += rightMax - height[r]
        }
    }

    return res
}
```

```
class Solution {
public:
    int trap(vector<int>& height) {
        int i = 0;
        int j = height.size() - 1;
        int leftMax = 0;
        int rightMax = 0;
        int ans = 0;
        while (i < j) {
            leftMax = max(leftMax, height[i]);
            rightMax = max(rightMax, height[j]);
            if (height[i] < height[j]) {
                ans += leftMax - height[i++];
            } else {
                ans += rightMax - height[j--];
            }
        }
        return ans;
    }
};
```

43. Multiply Strings

```
func multiply(num1 string, num2 string) string {
    if num1 == "0" || num2 == "0" {
        return "0"
    }

    m, n := len(num1), len(num2)
    digits := make([]int, m + n)
```

```go
    for i := m - 1; i >= 0; i-- {
        x := int(num1[i] - '0')
        for j := n - 1; j >= 0; j-- {
            y := int(num2[j] - '0')
            digits[i + j + 1] += x * y
        }
    }

    for i := m + n - 1; i > 0; i-- {
        digits[i - 1] += digits[i] / 10
        digits[i] = digits[i] % 10
    }

    ans := ""
    idx := 0
    if digits[0] == 0 {
        idx = 1
    }
    for ; idx < m + n; idx++ {
        ans += strconv.Itoa(digits[idx])
    }
    return ans
}
```

46. Permutations

```go
func permute(nums []int) [][]int {
    res := [][]int{}

    var backtrack func([]int, map[int]bool)
    backtrack = func(permutation []int, used map[int]bool) {
        if len(permutation) == len(nums) {
            res = append(res, append([]int{}, permutation...))
            return
        }

        for i := 0; i < len(nums); i++ {
            if !used[nums[i]] {
                used[nums[i]] = true
                permutation = append(permutation, nums[i])
                backtrack(permutation, used)
                used[nums[i]] = false
                permutation = permutation[:len(permutation)-1]
            }
        }
    }

    backtrack([]int{}, map[int]bool{})
    return res
}
```

```
func permute(nums []int) [][]int {
    result := [][]int{}

    // base case
    if len(nums) == 1 {
        return [][]int{append([]int{}, nums...)}
    }

    for _ = range nums {
        x := nums[0]
        nums = nums[1:]

        perms := permute(nums)
        for i := range perms {
            perms[i] = append(perms[i], x)
        }
        result = append(result, perms...)
        nums = append(nums, x)
    }

    return result
}
```

48. Rotate Image

```
func rotate(matrix [][]int)  {
    n := len(matrix)

    // 水平翻转
    for i := 0; i < n/2; i++ {
        matrix[i], matrix[n-1-i] = matrix[n-1-i], matrix[i]
    }

    // 主对角线翻转
    for i := 0; i < n; i++ {
        for j := 0; j < i; j++ {
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
        }
    }
}
```

53. Maximum Subarray

```
func maxSubArray(nums []int) int {
    sum := 0
    ans := math.MinInt
    for i := range nums {
        sum += nums[i]
        ans = max(ans, sum)
        if sum < 0 {
            sum = 0
        }
    }
    return ans
}
```

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int sum = 0;
        int ans = INT_MIN;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            ans = max(ans, sum);
            if (sum < 0) {
                sum = 0;
            }
        }
        return ans;
    }
};
```

54. Spiral Matrix

```
func spiralOrder(matrix [][]int) []int {
    nr, nc := len(matrix), len(matrix[0])
    visited := make([][]bool, nr)
    for i := 0; i < nr; i++ {
        visited[i] = make([]bool, nc)
    }

    var (
        total = nr * nc
        order = make([]int, total)
        r, c = 0, 0
        dir = [][]int{[]int{0, 1}, []int{1, 0}, []int{0, -1}, []int{-1, 0}}
        dirIdx = 0
    )
.
    for i := 0; i < total; i++ {
        order[i] = matrix[r][c]
        visited[r][c] = true
        nextR, nextC := r + dir[dirIdx][0], c + dir[dirIdx][1]
        if nextR < 0 || nextR >= nr || nextC < 0 || nextC >= nc ||
visited[nextR][nextC] {
            dirIdx = (dirIdx + 1) % 4
        }
        r += dir[dirIdx][0]
        c += dir[dirIdx][1]
    }

    return order
}
```

```cpp
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int nr = matrix.size();
        int nc = matrix[0].size();
        vector<vector<bool>> visited(nr, vector<bool>(nc, false));
        int n = nr * nc;
        vector<int> ans(n);
        int r = 0;
        int c = 0;
        vector<vector<int>> dir = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        int dirIdx = 0;

        for (int i = 0; i < n; i++) {
            ans[i] = matrix[r][c];
            visited[r][c] = true;
            int nextR = r + dir[dirIdx][0];
            int nextC = c + dir[dirIdx][1];
            if (nextR < 0 || nextR >= nr || nextC < 0 || nextC >= nc ||
visited[nextR][nextC]) {
                dirIdx = (dirIdx + 1) % 4;
            }
            r += dir[dirIdx][0];
            c += dir[dirIdx][1];
        }

        return ans;
    }
};
```

56. Merge Intervals

```go
func merge(intervals [][]int) [][]int {
    sort.Slice(intervals, func(i int, j int) bool {
        return intervals[i][0] < intervals[j][0]
    })

    merged := [][]int{}

    for i := range intervals {
        L, R := intervals[i][0], intervals[i][1]
        if len(merged) == 0 || merged[len(merged)-1][1] < L {
            merged = append(merged, []int{L, R})
        } else {
            merged[len(merged)-1][1] = max(merged[len(merged)-1][1], R)
        }
    }

    return merged
}
```

62. Unique Paths

```go
func uniquePaths(m int, n int) int {
    dp := make([][]int, m)
    for i := range dp {
        dp[i] = make([]int, n)
        dp[i][0] = 1
    }
    for j := 0; j < n; j++ {
        dp[0][j] = 1
    }

    for i := 1; i < m; i++ {
        for j := 1; j < n; j++ {
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
        }
    }
    return dp[m-1][n-1]
}
```

```go
func uniquePaths(m int, n int) int {
    dp := make([]int, n)
    for i := range dp {
        dp[i] = 1
    }

    for i := 1; i < m; i++ {
        for j := 1; j < n; j++ {
            dp[j] += dp[j-1]
        }
    }

    return dp[n-1]
}
```

88. Merge Sorted Array

```go
func merge(nums1 []int, m int, nums2 []int, n int)  {
    tail := m + n - 1
    i, j := m - 1, n - 1
    for i >= 0 && j >= 0 {
        if nums1[i] >= nums2[j] {
            nums1[tail] = nums1[i]
            i--
        } else {
            nums1[tail] = nums2[j]
            j--
        }
        tail--
    }
    for i >= 0 {
        nums1[tail] = nums1[i]
        i--
        tail--
    }
    for j >= 0 {
        nums1[tail] = nums2[j]
        j--
        tail--
    }
}
```

92. Reverse Linked List II

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode(): val(0), next(nullptr) {}
 *     ListNode(int x): val(x), next(nullptr) {}
 *     ListNode(int x, ListNode* next): val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        ListNode dummy_head(0, head);
        ListNode* predecressor = &dummy_head;
        for (int i = 0; i < left - 1; i++) {
            predecressor = predecressor->next;
        }
        ListNode* target_tail = predecressor;
        for (int i = 0; i < right - left + 1; i++) {
            target_tail = target_tail->next;
        }
        ListNode* target_head = predecressor->next;
        ListNode* successor = target_tail->next;
        predecressor->next = nullptr;
        target_tail->next = nullptr;
        predecressor->next = reverseLinkedList(target_head);
        target_head->next = successor;
        return dummy_head.next;
    }

private:
    ListNode* reverseLinkedList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
};
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode(): val(0), next(nullptr) {}
 *     ListNode(int x): val(x), next(nullptr) {}
 *     ListNode(int x, ListNode* next): val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        ListNode dummy_head(0, head);
        ListNode* predecessor = &dummy_head;
        for (int i = 0; i < left - 1; i++) {
            predecessor = predecessor->next;
        }
        ListNode* curr = predecessor->next;
        for (int i = 0; i < right - left; i++) {
            // 把 curr->next 挖出来放到 predecessor 的后面
            ListNode* temp = curr->next;
            curr->next = temp->next;
            temp->next = predecessor->next;
            predecessor->next = temp;
        }
        return dummy_head.next;
    }
};
```

103. Binary Tree Zigzag Level Order Traversal

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func zigzagLevelOrder(root *TreeNode) [][]int {
    result := [][]int{}
    if root == nil {
        return result
    }
    q := []*TreeNode{root}
    leftToRight := true
    for len(q) > 0 {
        size := len(q)
        row := make([]int, size)
        for i := 0; i < size; i++ {
            node := q[0]
            q = q[1:]
            var index int
            if leftToRight {
                index = i
            } else {
                index = size - 1 - i
            }
            row[index] = node.Val
            if node.Left != nil {
                q = append(q, node.Left)
            }
            if node.Right != nil {
                q = append(q, node.Right)
            }
        }
        leftToRight = !leftToRight
        result = append(result, row)
    }
    return result
}
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode* left;
 *     TreeNode* right;
 *     TreeNode(): val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode* left, TreeNode* right): val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        if (root == nullptr) {
            return {};
        }
        vector<vector<int>> ans;
        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;
        while (!q.empty()) {
            int n = q.size();
            vector<int> row(n);
            for (int i = 0; i < n; i++) {
                TreeNode* node = q.front();
                q.pop();
                int index = leftToRight ? i : (n - 1 - i);
                row[index] = node->val;
                if (node->left != nullptr) {
                    q.push(node->left);
                }
                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }
            leftToRight = !leftToRight;
            ans.push_back(row);
        }
        return ans;
    }
};
```

121. Best Time to Buy and Sell Stock

```go
func maxProfit(prices []int) int {
    minPrice := math.MaxInt
    ans := 0
    for i := range prices {
        ans = max(ans, prices[i] - minPrice)
        minPrice = min(minPrice, prices[i])
    }
    return ans
}
```

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minPrice = INT_MAX;
        int maxProfit = 0;
        for (int price : prices) {
            maxProfit = max(maxProfit, price - minPrice);
            minPrice = min(minPrice, price);
        }
        return max_profit;
    }
};
```

141. Linked List Cycle

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func hasCycle(head *ListNode) bool {
    visited := map[*ListNode]bool{}
    for head != nil {
        if visited[head] {
            return true
        }
        visited[head] = true
        head = head.Next
    }
    return false
}
```

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func hasCycle(head *ListNode) bool {
    slow, fast := head, head
    for fast != nil && fast.Next != nil {
        slow = slow.Next
        fast = fast.Next.Next
        if fast == slow {
            return true
        }
    }
    return false
}
```

142. Linked List Cycle II

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode(int x): val(x), next(nullptr) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode* head) {
        unordered_set<ListNode*> visited;
        while (head != nullptr) {
            if (visited.find(head) != visited.end()) {
                return head;
            }
            visited.insert(head);
            head = head->next;
        }
        return nullptr;
    }
};
```

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode* next;
 *     ListNode(int x): val(x), next(nullptr) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
            if (fast == slow) {
                ListNode* ptr = head;
                while (ptr != slow) {
                    ptr = ptr->next;
                    slow = slow->next;
                }
                return ptr;
            }
        }
        return nullptr;
    }
};
```

146. LRU Cache

```
type Node struct {
    Key, Value int
    Prev, Next *Node
}

type LRUCache struct {
    keys       map[int]*Node
    head, tail *Node
    capacity   int
}

func Constructor(capacity int) LRUCache {
    return LRUCache{
        keys:     make(map[int]*Node),
        capacity: capacity,
    }
}

func (this *LRUCache) Get(key int) int {
    if node, ok := this.keys[key]; ok {
        this.remove(node)
        this.add(node)
        return node.Value
    }
    return -1
}

func (this *LRUCache) Put(key int, value int) {
    if node, ok := this.keys[key]; ok {
        this.remove(node)
        node.Value = value
        this.add(node)
        return
    }
    node := &Node{Key: key, Value: value}
    this.keys[key] = node
    this.add(node)
    if len(this.keys) > this.capacity {
        delete(this.keys, this.tail.Key)
        this.remove(this.tail)
    }
}

func (this *LRUCache) add(node *Node) {
    node.Prev = nil
    node.Next = this.head
    if this.head != nil {
        this.head.Prev = node
    }
    this.head = node
    if this.tail == nil {
        this.tail = node
    }
}

func (this *LRUCache) remove(node *Node) {
    if node.Prev != nil {
        node.Prev.Next = node.Next
    }
    if node.Next != nil {
        node.Next.Prev = node.Prev
    }
    if node == this.head {
        this.head = node.Next
    }
    if node == this.tail {
        this.tail = node.Prev
    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * obj := Constructor(capacity);
 * param_1 := obj.Get(key);
 * obj.Put(key,value);
 */
```

```
struct Node {
    int key;
    int value;
    Node* prev;
    Node* next;
    Node(int key, int value) : key(key), value(value) {}
};

class LRUCache {
public:
    LRUCache(int capacity) : keys(unordered_map<int, Node*>()),
capacity(capacity), head(nullptr), tail(nullptr) {}

    int get(int key) {
        if (keys.contains(key)) {
            Node* node = keys.at(key);
            remove(node);
            add(node);
            return node->value;
        }
        return -1;
    }

    void put(int key, int value) {
        if (keys.contains(key)) {
            Node* node = keys.at(key);
            remove(node);
            node->value = value;
            add(node);
            return;
        }
        Node* node = new Node(key, value);
        keys[key] = node;
        add(node);
        if (keys.size() > capacity) {
            keys.erase(tail->key);
            remove(tail);
        }
    }

private:
    unordered_map<int, Node*> keys;
    Node* head;
    Node* tail;
    int capacity;

    void add(Node* node) {
        node->prev = nullptr;
        node->next = head;
        if (head != nullptr) {
            head->prev = node;
        }
        head = node;
        if (tail == nullptr) {
            tail = node;
        }
    }

    void remove(Node* node) {
        if (node->prev != nullptr) {
            node->prev->next = node->next;
        }
        if (node->next != nullptr) {
            node->next->prev = node->prev;
        }
        if (node == head) {
            head = node->next;
        }
        if (node == tail) {
            tail = node->prev;
        }
    }
};

/**
```

```
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

160. Intersection of Two Linked Lists

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func getIntersectionNode(headA, headB *ListNode) *ListNode {
    visited := map[*ListNode]bool{}
    for tmp := headA; tmp != nil; tmp = tmp.Next {
        visited[tmp] = true
    }
    for tmp := headB; tmp != nil; tmp = tmp.Next {
        if visited[tmp] {
            return tmp
        }
    }
    return nil
}
```

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
```

```
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        unordered_map<ListNode*, bool> visited;
        for (ListNode* tmp = headA; tmp != nullptr; tmp = tmp->next) {
            visited[tmp] = true;
        }
        for (ListNode* tmp = headB; tmp != nullptr; tmp = tmp->next) {
            if (visited[tmp]) {
                return tmp;
            }
        }
        return nullptr;
    }
};
```

200. Number of Islands

```
func numIslands(grid [][]byte) int {
    nr := len(grid)
    if nr == 0 {
        return 0
    }
    nc := len(grid[0])

    ans := 0
    for r := 0; r < nr; r++ {
        for c := 0; c < nc; c++ {
            if grid[r][c] == '1' {
                ans++
                dfs(grid, r, c)
            }
        }
    }

    return ans
}

func dfs(grid [][]byte, r int, c int) {
    nr := len(grid)
    nc := len(grid[0])
    grid[r][c] = '0'
    if r - 1 >= 0 && grid[r - 1][c] == '1' {
        dfs(grid, r - 1, c)
    }
    if r + 1 <= nr - 1 && grid[r + 1][c] == '1' {
        dfs(grid, r + 1, c)
    }
    if c - 1 >= 0 && grid[r][c - 1] == '1' {
        dfs(grid, r, c - 1)
    }
    if c + 1 <= nc - 1 && grid[r][c + 1] == '1' {
        dfs(grid, r, c + 1)
    }
}
```

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int nr = grid.size();
        if (nr == 0) {
            return 0;
        }
        int nc = grid[0].size();

        int ans = 0;
        for (int r = 0; r < nr; r++) {
            for (int c = 0; c < nc; c++) {
                if (grid[r][c] == '1') {
                    ans++;
                    dfs(grid, r, c);
                }
            }
        }

        return ans;
    }

    void dfs(vector<vector<char>>& grid, int r, int c) {
        int nr = grid.size();
        int nc = grid[0].size();
        grid[r][c] = '0';
        if (r - 1 >= 0 && grid[r - 1][c] == '1') {
            dfs(grid, r - 1, c);
        }
        if (r + 1 < nr && grid[r + 1][c] == '1') {
            dfs(grid, r + 1, c);
        }
        if (c - 1 >= 0 && grid[r][c - 1] == '1') {
            dfs(grid, r, c - 1);
        }
        if (c + 1 < nc && grid[r][c + 1] == '1') {
            dfs(grid, r, c + 1);
        }
    }
};
```

206. Reverse Linked List

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func reverseList(head *ListNode) *ListNode {
    var prev *ListNode = nil
    curr := head
    for curr != nil {
        tmp := curr.Next
        curr.Next = prev
        prev = curr
        curr = tmp
    }
    return prev
}
```

```
/**
```

```
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
};
```

215. Kth Largest Element in an Array

```
func findKthLargest(nums []int, k int) int {
    return quickSelect(nums, 0, len(nums) - 1, k - 1)
}

func quickSelect(nums []int, p int, r int, index int) int {
    q := randomizedPartition(nums, p, r)
    if index == q {
        return nums[q]
    } else if index < q {
        return quickSelect(nums, p, q - 1, index)
    } else {
        return quickSelect(nums, q + 1, r, index)
    }
}
```

```go
func randomizedPartition(nums []int, p int, r int) int {
    i := rand.Intn(r - p + 1) + p
    nums[i], nums[r] = nums[r], nums[i]
    return partition(nums, p, r)
}

func partition(nums []int, p int, r int) int {
    i := p - 1
    for j := p; j < r; j++ {
        if nums[j] >= nums[r] {
            i += 1
            nums[i], nums[j] = nums[j], nums[i]
        }
    }
    nums[i + 1], nums[r] = nums[r], nums[i + 1]
    return i + 1
}
```

```go
func findKthLargest(nums []int, k int) int {
    n := len(nums)
    return quickSelect(nums, 0, n-1, n-k)
}

func quickSelect(nums []int, lo int, hi int, k int) int {
    for lo < hi {
        pivot := partition(nums, lo, hi)
        if pivot < k {
            lo = pivot + 1
        } else if pivot > k {
            hi = pivot - 1
        } else {
            break
        }
    }
    return nums[k]
}

func partition(nums []int, lo int, hi int) int {
    i, j := lo, hi+1
    for {
        for i++; i < hi && nums[i] < nums[lo]; i++ {
        }
        for j--; j > lo && nums[lo] < nums[j]; j-- {
        }
        if i >= j {
            break
        }
        nums[i], nums[j] = nums[j], nums[i]
    }
    nums[lo], nums[j] = nums[j], nums[lo]
    return j
}
```

```go
func findKthLargest(nums []int, k int) int {
    n := len(nums)
    return quickselect(nums, 0, n-1, n-k)
}

func quickselect(nums []int, l, r, k int) int {
    if l == r {
        return nums[k]
    }

    partition := nums[l]
    i := l - 1
    j := r + 1

    for i < j {
        i++
        for nums[i] < partition {
            i++
        }

        j--
        for nums[j] > partition {
            j--
        }

        if i < j {
            nums[i], nums[j] = nums[j], nums[i]
        }
    }
    if k <= j {
        return quickselect(nums, l, j, k)
    } else {
        return quickselect(nums, j+1, r, k)
    }
}
```

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();
        return quickselect(nums, 0, n - 1, n - k);
    }

    int quickselect(vector<int>& nums, int l, int r, int kthSmallest) {
        if (l == r) {
            return nums[kthSmallest];
        }
        int pivot = nums[l];
        int i = l - 1;
        int j = r + 1;
        while (i < j) {
            do {
                i++;
            } while (nums[i] < pivot);
            do {
                j--;
            } while (nums[j] > pivot);
            if (i < j) {
                swap(nums[i], nums[j]);
            }
        }
        if (kthSmallest <= j) {
            return quickselect(nums, l, j, kthSmallest);
        } else {
            return quickselect(nums, j + 1, r, kthSmallest);
        }
    }
};
```

```go
func findKthLargest(nums []int, k int) int {
    return quickSelect(nums, 0, len(nums)-1, len(nums)-k)
}

func quickSelect(nums []int, p int, r int, idx int) int {
    lt, gt := randomizedThreePartition(nums, p, r)
    if idx < lt {
        return quickSelect(nums, p, lt, idx)
    } else if idx > gt {
        return quickSelect(nums, gt, r, idx)
    } else {
        return nums[lt]
    }
}

func randomizedThreePartition(nums []int, p int, r int) (int, int) {
    i := rand.Intn(r-p+1) + p
    nums[i], nums[r] = nums[r], nums[i]
    return threeWayPartition(nums, p, r)
}

func threeWayPartition(nums []int, p int, r int) (int, int) {
    lt, gt := p-1, r
    pivot := nums[r]
    i := p
    for i < gt {
        if nums[i] < pivot {
            lt++
            nums[i], nums[lt] = nums[lt], nums[i]
            i++
        } else if nums[i] > pivot {
            gt--
            nums[i], nums[gt] = nums[gt], nums[i]
        } else {
            i++
        }
    }
    nums[r], nums[gt] = nums[gt], nums[r]
    return lt, gt + 1
}
```

```go
func findKthLargest(nums []int, k int) int {
    h := Constructor(nums)
    for i := 0; i < k - 1; i++ {
        h.ExtractMax()
    }
    return h.ExtractMax()
}

type MaxHeap struct {
    nums []int
    size int
}

func Constructor(nums []int) *MaxHeap {
    h := &MaxHeap{nums: nums, size: len(nums)}
    h.buildMaxHeap()
    return h
}

func (h *MaxHeap) ExtractMax() int {
    h.nums[0], h.nums[h.size - 1] = h.nums[h.size - 1], h.nums[0]
    h.size--
    h.maxHeapify(0)
    return h.nums[h.size]
}

func (h *MaxHeap) buildMaxHeap() {
    for i := h.size / 2 - 1; i >= 0; i-- {
        h.maxHeapify(i)
    }
}

func (h *MaxHeap) maxHeapify(i int) {
    l := left(i)
    r := right(i)
    largest := i
    if l < h.size && h.nums[l] > h.nums[largest] {
        largest = l
    }
    if r < h.size && h.nums[r] > h.nums[largest] {
        largest = r
    }
    if largest != i {
        h.nums[i], h.nums[largest] = h.nums[largest], h.nums[i]
        h.maxHeapify(largest)
    }
}

func left(i int) int {
    return i * 2 + 1
}

func right(i int) int {
    return i * 2 + 2
}
```

236. Lowest Common Ancestor of a Binary Tree

```
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
 func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
     if root == nil || root == p || root == q {
         return root
     }
     left := lowestCommonAncestor(root.Left, p, q)
     right := lowestCommonAncestor(root.Right, p, q)
     if left == nil {
         return right
     }
     if right == nil {
         return left
     }
     return root
}
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == nullptr || root == p || root == q) {
            return root;
        }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left == nullptr) {
            return right;
        }
        if (right == nullptr) {
            return left;
        }
        return root;
    }
};
```

415. Add Strings

```go
func addStrings(num1 string, num2 string) string {
    i := len(num1) - 1
    j := len(num2) - 1
    carry := 0
    ans := ""
    for i >= 0 || j >= 0 || carry > 0 {
        x := 0
        if i >= 0 {
            x = int(num1[i] - '0')
        }
        y := 0
        if j >= 0 {
            y = int(num2[j] - '0')
        }
        sum := x + y + carry
        ans = strconv.Itoa(sum % 10) + ans
        carry = sum / 10
        i--
        j--
    }
    return ans
}
```