Solution

Language: cpp

Status: Accepted

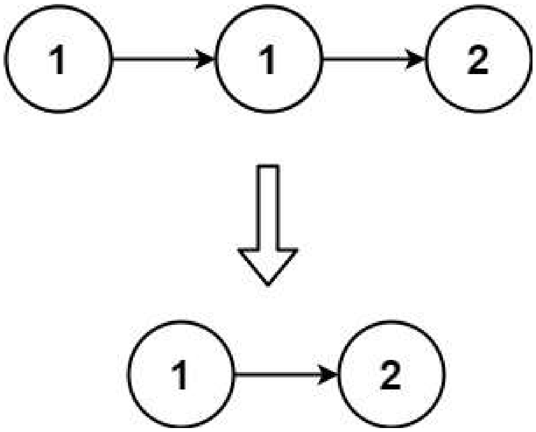
```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

    int i = 0;
    int j = 0;
    vector<int> res;

    if (m == 1 && n == 0)
      return;

    while (i < m && j < n)
    {
      if (nums1[i] < nums2[j])
      {
        res.push_back(nums1[i]);
        i++;
      }
      else if (nums1[i] >= nums2[j])
      {
        res.push_back(nums2[j]);
        j++;
      }
    }
    if (i == m)
      while (j < n)
      {
        res.push_back(nums2[j]);
        j++;
      }
    if (j == n)
      while (i < m)
      {
        res.push_back(nums1[i]);
        i++;
      }

    nums1 = res;

    }
};
```

83 Remove Duplicates from Sorted List (link)

Description

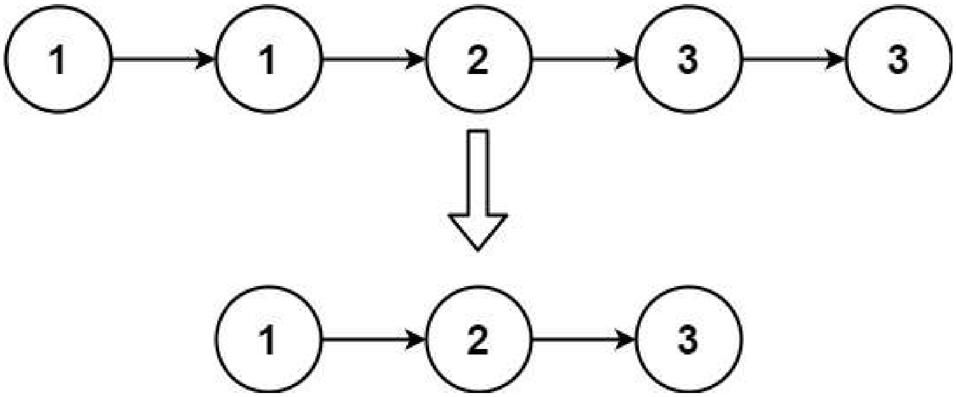
Given the `head` of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

Example 1:



```
Input: head = [1,1,2]
Output: [1,2]
```

Example 2:



```
Input: head = [1,1,2,3,3]
Output: [1,2,3]
```

Constraints:

- The number of nodes in the list is in the range `[0, 300]`.
- `-100 <= Node.val <= 100`
- The list is guaranteed to be **sorted** in ascending order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```cpp
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {

    vector<ListNode*> v;
    ListNode* ptr = head;
    int prev = -1;
    bool flag = true;

    if (head == nullptr)
      return(nullptr);

    while (ptr != nullptr)
    {
      if (ptr->val != prev || flag == true)
      {
        v.push_back(ptr);
        flag = false;
      }
      prev = ptr->val;
      ptr = ptr->next;
    }

    for (int i = 0; i < v.size() - 1; i++)
      v[i]->next = v[i+1];

    v.back()->next = nullptr;

    return (v.front());
    }
};
```

69 Sqrt(x) (link)

Description

Given a non-negative integer `x`, return *the square root of* `x` *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.

You **must not use** any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in c++ or `x ** 0.5` in python.

Example 1:

```
Input: x = 4
Output: 2
Explanation: The square root of 4 is 2, so we return 2.
```

Example 2:

```
Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is
```

Constraints:

- `0 <= x <=` 2^{31} `- 1`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int mySqrt(int x) {

      int i;
      int m,l,r;

      if (x == 0)
        return(0);
      if (x == 1)
        return(1);

      l = 0;
      r = x;

      while (l <= r)
      {
        m = (r + l) / 2;
        if (m < x / m)
          l = m + 1;
        if (m > x / m)
          r = m - 1;
        if (m == x / m)
          return(m);
      }

      if (m > x / m)
        return(--m);
      else
        return(m);
    }
};
```

67 Add Binary (link)

Description

Given two binary strings `a` and `b`, return *their sum as a binary string*.

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```

Constraints:

- `1 <= a.length, b.length <=` 10^4
- `a` and `b` consist only of `'0'` or `'1'` characters.
- Each string does not contain leading zeros except for the zero itself.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```cpp
class Solution {
public:
    string addBinary(string a, string b) {

    string res = "";
    int i = a.length() - 1;
    int j = b.length() - 1;
    int carry = 0;


    while (i >= 0 && j >= 0)
    {
      if (a[i] - 48 + b[j] - 48 + carry == 3)
      {
        res.insert(0,"1");
        carry = 1;
      }
      else if (a[i] - 48 + b[j] - 48 + carry == 2)
      {
        res.insert(0,"0");
        carry = 1;
      }
      else if (a[i] - 48 + b[j] - 48 + carry == 1)
      {
        res.insert(0,"1");
        carry = 0;
      }
      else if (a[i] - 48 + b[j] - 48 + carry == 0)
      {
        res.insert(0,"0");
        carry = 0;
      }
      i--;
      j--;
    }

    if (j < 0)
        while (i >= 0)
        {
          if (a[i] - 48 + carry == 2)
          {
            res.insert(0, "0");
            carry = 1;
          }
          else if (a[i] - 48 + carry == 1)
          {
            res.insert(0,"1");
            carry = 0;
          }
          else if (a[i] - 48 + carry == 0)
          {
            res.insert(0,"0");
            carry = 0;
          }
          i--;
        }
    else
      if (i < 0)
        while (j >= 0)
        {
          if (b[j] - 48 + carry == 2)
          {
            res.insert(0, "0");
            carry = 1;
          }
          else if (b[j] - 48 + carry == 1)
          {
            res.insert(0,"1");
            carry = 0;
          }
          else if (b[j] - 48 + carry == 0)
          {
            res.insert(0,"0");
            carry = 0;
          }
          j--;
        }

    if (carry == 1)
        res.insert(0, "1");

    return(res);
    }
};
```

66 Plus One (link)

Description

You are given a **large integer** represented as an integer array `digits`, where each `digits[i]` is the `i`th digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading `0`'s.

Increment the large integer by one and return *the resulting array of digits*.

Example 1:

```
Input: digits = [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].
```

Example 2:

```
Input: digits = [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
Incrementing by one gives 4321 + 1 = 4322.
Thus, the result should be [4,3,2,2].
```

Example 3:

```
Input: digits = [9]
Output: [1,0]
Explanation: The array represents the integer 9.
Incrementing by one gives 9 + 1 = 10.
Thus, the result should be [1,0].
```

Constraints:

- `1 <= digits.length <= 100`
- `0 <= digits[i] <= 9`
- `digits` does not contain any leading `0`'s.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```cpp
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {

    int carry = 0;

    for (int i = digits.size() - 1; i >= 0; i--)
    {
      if (i == digits.size() - 1)
      {

        if (digits[i] + 1 < 10)
        {
          digits[i] = digits[i] + 1;
          carry = 0;
        }
        else
        {
          digits[i] = 0;
          carry = 1;
        }

      }

      else
      {

        if (digits[i] + carry < 10)
        {
          digits[i] = digits[i] + carry;
          carry = 0;
        }
        else
        {
          digits[i] = 0;
          carry = 1;
        }

      }
    }

    if (carry == 1)
      digits.insert(digits.begin(), 1);
    return(digits);

    }
};
```

41 First Missing Positive (link)

Description

Given an unsorted integer array nums. Return the *smallest positive integer* that is *not present* in nums.

You must implement an algorithm that runs in O(n) time and uses O(1) auxiliary space.

Example 1:

```
Input: nums = [1,2,0]
Output: 3
Explanation: The numbers in the range [1,2] are all in the array.
```

Example 2:

```
Input: nums = [3,4,-1,1]
Output: 2
Explanation: 1 is in the array but 2 is missing.
```

Example 3:

```
Input: nums = [7,8,9,11,12]
Output: 1
Explanation: The smallest positive integer 1 is missing.
```

Constraints:

- 1 <= nums.length <= 10^5
- -2^{31} <= nums[i] <= 2^{31} - 1

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:

    int firstMissingPositive(vector<int>& nums) {

        int i = 0;
        int j = 0;
        int res = 0;

        if (nums.empty())
            return(1);

        if (nums.size() == 1 && nums[0] < 0)
            return(1);
        if (nums.size() == 1 && nums[0] > 1)
            return(1);
        if (nums.size() == 1 && nums[0] == 1)
            return(2);

        sort(nums.begin(), nums.end());

        if (nums[0] > 1)
            return(1);

        for (i = 0; i < nums.size(); i++)
            if (nums[i] >= 0)
                break;

        if (i == nums.size())
          return(1);

        if (nums[i] > 1)
          return(1);

        for (j = i; j < nums.size() - 1; j++)
        {
            if (nums[j+1] > nums[j] + 1)
            {
                res = nums[j] + 1;
                break;
            }
        }
        if (j == nums.size() - 1)
            res = nums[j] + 1;
        return(res);
    }
};
```

35 Search Insert Position (link)

Description

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Constraints:

- 1 <= nums.length <= 10^4
- -10^4 <= nums[i] <= 10^4
- `nums` contains **distinct** values sorted in **ascending** order.
- -10^4 <= target <= 10^4

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {

      int l, r, mid;

      l = 0;
      r = nums.size() - 1;

      while (l <= r)
      {
        mid = (r + l) / 2;
        if (target > nums[mid])
          l = mid + 1;
        if (target < nums[mid])
          r = mid - 1;
        if (target == nums[mid])
        {
          l = mid;
          break;
        }
        mid = l;
      }
      return(l);
    }
};
```

27 Remove Element (link)

Description

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place**. The order of the elements may be changed. Then return *the number of elements in* `nums` *which are not equal to* `val`.

Consider the number of elements in `nums` which are not equal to `val` be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the elements which are not equal to `val`. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int val = ...; // Value to remove
int[] expectedNums = [...]; // The expected answer with correct length.
                            // It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Example 1:

```
Input: nums = [3,2,2,3], val = 3
Output: 2, nums = [2,2,_,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 2.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Example 2:

```
Input: nums = [0,1,2,2,3,0,4,2], val = 2
Output: 5, nums = [0,1,4,0,3,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums containing 0, 0, 1,
Note that the five elements can be returned in any order.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Constraints:

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 50`
- `0 <= val <= 100`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {

      vector<int> :: iterator j;

      for (int i = 0; i < nums.size(); i++)
      {
        if (nums[i] == val)
        {
          j = nums.begin() + i;
          nums.erase(j);
          i--;
        }
      }
      return(nums.size());
    }

};
```

26 Remove Duplicates from Sorted Array (link)

Description

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

- Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
- Return k.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respec
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, a
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Constraints:

- 1 <= nums.length <= 3 * 10^4
- -100 <= nums[i] <= 100
- nums is sorted in **non-decreasing** order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
  int removeDuplicates(vector<int>& nums) {

  vector<int> :: iterator j;

  if (nums.size() == 0)
   return(0);
  if (nums.size() == 1)
   return(1);

  for (int i = 1; i < nums.size() - 1; i++)
  {
// cout << nums[i];
   if (nums[i] == nums[i-1])
   {
     j = nums.begin() + i - 1;
     nums.erase(j);
     i--;
   }
  }

  if (nums[nums.size() - 1] == nums[nums.size() - 2])
   nums.pop_back();
  return(nums.size());
  }
};
```

25 Reverse Nodes in k-Group (link)

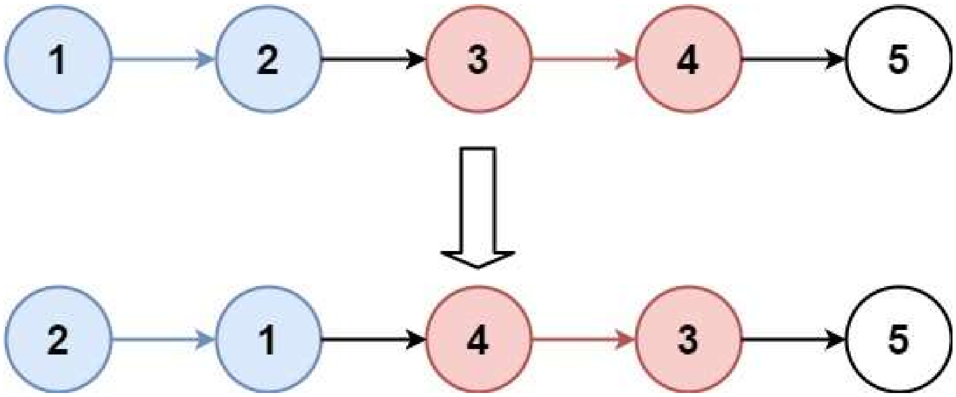
Description

Given the `head` of a linked list, reverse the nodes of the list `k` at a time, and return *the modified list*.

`k` is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of `k` then left-out nodes, in the end, should remain as it is.

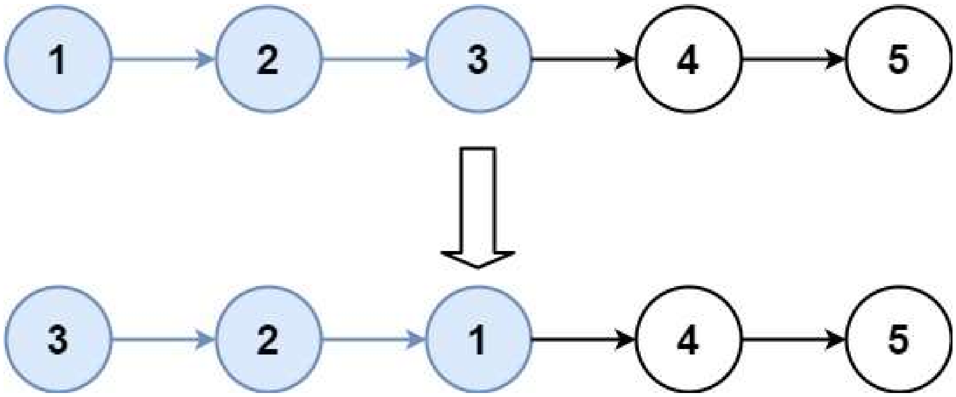
You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:



```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Example 2:



```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
```

Constraints:

- The number of nodes in the list is `n`.
- `1 <= k <= n <= 5000`
- `0 <= Node.val <= 1000`

Follow-up: Can you solve the problem in `O(1)` extra memory space?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/**
 * 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:
    void reversek(vector<ListNode*> &v, vector<ListNode*> &res)
    {
      vector<ListNode*> reverse;

      for (int i = v.size() - 1; i > -1; i--)
      {
        if (res.size() != 0)
          res[res.size() - 1]->next = v[i];
        res.push_back(v[i]);
      }
    }

    ListNode* reverseKGroup(ListNode* head, int k) {

    ListNode *p = head;
    vector<ListNode*> v, res;
    int index = 0;

    while (p != nullptr)
    {
      v.push_back(p);
      index++;
      p = p->next;
      if (index == k)
      {
        reversek(v, res);
        v.clear();
        index = 0;
      }
    }

    if (v.size() != 0)
      for (int i = 0; i < v.size(); i++)
      {
        if (res.size() != 0)
          res[res.size() - 1]->next = v[i];
        res.push_back(v[i]);
      }

    res[res.size() - 1]->next = nullptr;

    return(res[0]);
  }
};
```

21 Merge Two Sorted Lists (link)

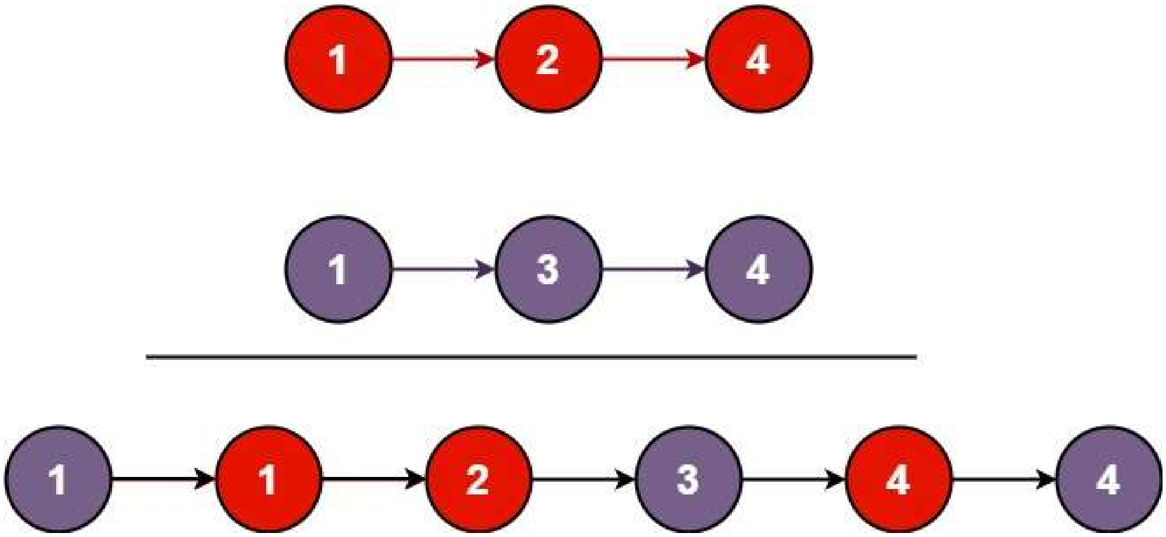
Description

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2:

```
Input: list1 = [], list2 = []
Output: []
```

Example 3:

```
Input: list1 = [], list2 = [0]
Output: [0]
```

Constraints:

- The number of nodes in both lists is in the range `[0, 50]`.
- `-100 <= Node.val <= 100`
- Both `list1` and `list2` are sorted in **non-decreasing** order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/**
 * 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* mergeTwoLists(ListNode* l1, ListNode* l2) {

    ListNode *ptr1;
    ListNode *ptr2;
    vector<ListNode*> v;

    ptr1 = l1;
    ptr2 = l2;



    if (l1 == nullptr && l2 == nullptr)
      return(l1);

    while (ptr1 != nullptr && ptr2 != nullptr)
    {
//    cout << ptr1->val << ptr2->val << " ";

      if (ptr1->val <= ptr2->val)
      {
        v.push_back(ptr1);
        ptr1 = ptr1->next;
        continue;
      }
      else
      {
        v.push_back(ptr2);
        ptr2 = ptr2->next;
        continue;
      }
    }

    if (ptr1 != nullptr)
      while (ptr1 != nullptr)
      {
        v.push_back(ptr1);
        ptr1 = ptr1->next;
      }

    if (ptr2 != nullptr)
      while (ptr2 != nullptr)
      {
        v.push_back(ptr2);
        ptr2 = ptr2->next;
      }

    for (int i = 0; i < v.size() - 1; i++)
      v[i]->next = v[i+1];

    v[v.size()-1]->next = nullptr;

    l1 = v[0];
    return(l1);

    }
};
```

20 Valid Parentheses (link)

Description

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]{}"
Output: true
```

Example 3:

```
Input: s = "(]"
Output: false
```

Constraints:

- `1 <= s.length <=` 10^4
- `s` consists of parentheses only `'()[]{}'`.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```cpp
class Solution {
public:

    char comp(char c)
    {
      if (c == '(')
        return(')');
      if (c == '{')
        return('}');
      if (c == '[')
        return(']');

      if (c == ')')
        return('(');
      if (c == '}')
        return('{');
      if (c == ']')
        return('[');

    return(0);
    }

    bool isValid(string s) {

      stack<char> r;

      for (int i = 0; i < s.size(); i++)
      {
        if (s[i] == '(' || s[i] == '{' || s[i] == '[')
          r.push(s[i]);

        if (s[i] == ')' || s[i] == '}' || s[i] == ']')
        {
          if (r.empty() == true)
            return(false);
//        cout << s[i] << comp(r.top());
          if (s[i] != comp(r.top()))
            return(false);
          else
            r.pop();
        }
      }

    if (r.empty() == false)
      return(false);

    return(true);

    }
};
```

13 Roman to Integer (link)

Description

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

```
Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, `2` is written as `II` in Roman numeral, just two ones added together. `12` is written as `XII`, which is simply `X + II`. The number `27` is written as `XXVII`, which is `XX + V + II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2:

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.
```

Example 3:

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Constraints:

- `1 <= s.length <= 15`
- `s` contains only the characters `('I', 'V', 'X', 'L', 'C', 'D', 'M')`.
- It is **guaranteed** that `s` is a valid roman numeral in the range `[1, 3999]`.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int romanToInt(string s) {

      int sum = 0;
      int i = 0;

      while (i < s.size())
      {
        if (s[i] == 'M')
        {
          sum = sum + 1000;
          i++;
          continue;
        }

        if (s[i] == 'C' && s[i+1] == 'M')
        {
          sum = sum + 900;
          i = i + 2;
          continue;
        }

        if (s[i] == 'D')
        {
          sum = sum + 500;
          i++;
          continue;
        }

        if (s[i] == 'C' && s[i+1] == 'D')
        {
          sum = sum + 400;
          i = i + 2;
          continue;
        }

        if (s[i] == 'C')
        {
          sum = sum + 100;
          i++;
          continue;
        }

        if (s[i] == 'X' && s[i+1] == 'C')
        {
          sum = sum + 90;
          i = i + 2;
          continue;
        }

        if (s[i] == 'L')
        {
          sum = sum + 50;
          i++;
          continue;
        }

        if (s[i] == 'X' && s[i+1] == 'L')
        {
          sum = sum + 40;
          i = i + 2;
          continue;
        }
        if (s[i] == 'X')
        {
          sum = sum + 10;
          i++;
          continue;
        }

        if (s[i] == 'I' && s[i+1] == 'X')
        {
          sum = sum + 9;
          i = i + 2;
          continue;
        }

        if (s[i] == 'V')
        {
          sum = sum + 5;
          i++;
          continue;
        }

        if (s[i] == 'I' && s[i+1] == 'V')
        {
          sum = sum + 4;
          i = i + 2;
          continue;
        }

        if (s[i] == 'I')
        {
          sum = sum + 1;
          i++;
          continue;
        }
      }

    return(sum);
    }
};
```

14 Longest Common Prefix (link)

Description

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string `""`.

Example 1:

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```

Example 2:

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

Constraints:

- `1 <= strs.length <= 200`
- `0 <= strs[i].length <= 200`
- `strs[i]` consists of only lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {

      string res;
      bool flag;
      char t;

      if (strs.empty())
        return("");
      if (strs.size() == 1)
        return(strs[0]);

//    Loop: Start loop with the letters of the first string at strs[0]

      for (int j = 0; j < strs[0].size(); j++) {

//    Loop: Start loop by comparing strs[0] with every string in the vector

        for (int i = 1; i < strs.size(); i++) {
          if (strs[0][j] == strs[i][j]) {
            flag = true;
            t = strs[0][j];
          }
          else {
            flag = false;
            break;
          }
        }
        if (flag == true) {
          res = res + t;
          flag = false;
        }
        else
          break;
      }
      return(res);
    }
};
```

9 Palindrome Number (link)

Description

Given an integer `x`, return `true` *if* `x` *is a* ***palindrome****, and* `false` *otherwise*.

Example 1:

```
Input: x = 121
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.
```

Example 2:

```
Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not
```

Example 3:

```
Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.
```

Constraints:

- $-2^{31} <= x <= 2^{31} - 1$

Follow up: Could you solve it without converting the integer to a string?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
   int raisePower(int x, int y)
    {
      int val = 1;
      for (int i = 0; i < y; i++)
        val = val * x;
      return(val);
    }

    bool isPalindrome(int x) {

 //   x = 1221;
      int length = 0;
      int i = 0;
      bool res = true;
      int y = x;

      if (x < 0)
        return(false);

      for (i = 9; i >= 0; i--)
      {
//      cout << x / (int)pow(10, i) << " ";
        if (x / raisePower(10, i) != 0)
          break;
      }

      length = i + 1;
//    cout << x << " length: " << length << endl;

      int a,b;

      for (int j = 0; j < length; j++)
      {
//      cout << x % 10 << ":" << y / (int)pow(10,length - j - 1) << " ";

        a = x % 10;
        b = y / raisePower(10,length - j - 1);

        x = x / 10;
        y = y % raisePower(10, length - j - 1);

        if (a != b)
        {
//        cout << res;
          return(false);
        }
      }
//    cout << res;
      return(true);
    }
};
```

4 Median of Two Sorted Arrays (link)

Description

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Constraints:

- nums1.length == m
- nums2.length == n
- 0 <= m <= 1000
- 0 <= n <= 1000
- 1 <= m + n <= 2000
- -10^6 <= nums1[i], nums2[i] <= 10^6

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

      int max = nums1.size() + nums2.size();
      vector<int> v;

      nums1.insert(nums1.end(),nums2.begin(),nums2.end());

      std :: sort(nums1.begin(), nums1.end());


 //     cout << nums1.size();
      if (nums1.size() % 2 == 1)
        return(nums1[nums1.size()/2]);
      else
      {
        double a = nums1[nums1.size()/2];
        double b = nums1[(nums1.size()/2) - 1];
        double c = (a+b)/2;
        cout << a << b;
        return(c);
      }


      return(0);
    }
};

// [1,3,5]
// [2,4]
```