1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to `target`*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

Constraints:

- `2 <= nums.length <= 10^5`
- `-10^9 <= nums[i] <= 10^9`
- `-10^9 <= target <= 10^9`
- **Only one valid answer exists.**

- Use unordered_map when the sorted map is not required.
- in maps m.insert is faster than m[key] = value.

---

3. Longest Substring Without Repeating Characters

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

Example 4:

```
Input: s = ""
Output: 0
```

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

Window technique
`vector<int> v(256, -1);` initialize the entire vector with value -1

---

5. Longest Palindromic Substring

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Note: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Example 3:

```
Input: s = "a"
Output: "a"
```

Example 4:

```
Input: s = "ac"
Output: "a"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters (lower-case and/or upper-case),

Manacher's Algorithm Used to solve palindrome substring in O(n)

---

15. 3Sum

Given an array `nums` of *n* integers, are there elements *a*, *b*, *c* in `nums` such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
```

Example 2:

```
Input: nums = []
Output: []
```

Example 3:

```
Input: nums = [0]
Output: []
```

Constraints:

- `0 <= nums.length <= 3000`
- -10^5 `<= nums[i] <=` 10^5

`ans.push_back(vector<int>{nums[i], nums[low], nums[high]});` Pushing values inside 2D vector

Fix one value and then use 2Sum using 2 pointers

17. Letter Combinations of a Phone Number

Given a string containing digits from `2–9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

Example 3:

```
Input: digits = "2"
Output: ["a","b","c"]
```

Constraints:

- `0 <= digits.length <= 4`
- `digits[i]` is a digit in the range `['2', '9']`.

We use recursion when their is variable nested FOR loops

34. Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

Follow up: Could you write an algorithm with `O(log n)` runtime complexity?

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

Example 3:

```
Input: nums = [], target = 0
Output: [-1,-1]
```

Constraints:

- `0 <= nums.length <= 10^5`
- `-10^9 <= nums[i] <= 10^9`
- `nums` is a non-decreasing array.
- `-10^9 <= target <= 10^9`

`mid = start + (end-start)/2`
This is preferrable method for finding midpoint and take care of overflow condition.

---

46. Permutations

Given a collection of **distinct** integers, return all possible permutations.

Example:

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

`map.count(key)` It returns 1 if the key is present in map else return 0.
`map.erase(key)` Delete the particular key in map.
`nums.erase(nums.begin() + i);` Erase element from position **i**.
`nums.insert(nums.begin() + i, n);` Insert element **n** at postion **i**.
`nums.empty()` Return 1 if vector is empty.

Backtracking Steps

- choose (what are the possibilities)
- explore (what are their outcome)
- un-choose (undone the changes for the next cycle also called the backtracking part)

---

49. Group Anagrams

Given an array of strings `strs` , group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

Example 2:

```
Input: strs = [""]
Output: [[""]]
```

Example 3:

```
Input: strs = ["a"]
Output: [["a"]]
```

Constraints:

- `1 <= strs.length <= 10^4`
- `0 <= strs[i].length <= 100`
- `strs[i]` consists of lower-case English letters.

```
template <typename Map> bool map_compare (Map const &lhs, Map const &rhs) { return lhs.size() == rhs.size() && equal(lhs.begin(),
lhs.end(), rhs.begin()); }
```
map_compare(a,b) ~ compare two maps

```
ans.erase(ans.begin()+count, ans.end());
```
Erase vector from **count** index to **end**.

---

56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

Constraints:

- `intervals[i][0] <= intervals[i][1]`

Intervals questions become easy after sorting

---

62. Unique Paths

A robot is located at the top-left corner of a `m x n` grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Example 1:


```
Input: m = 3, n = 7
Output: 28
```

Example 2:

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down
```

Example 3:

```
Input: m = 7, n = 3
Output: 28
```

Example 4:

```
Input: m = 3, n = 3
Output: 6
```

Constraints:

- `1 <= m, n <= 100`
- It's guaranteed that the answer will be less than or equal to `2 * 10^9`.

Backtracking and dp goes hand in hand. In most of the cases the backtracking problem can be optimised using dp.

---

70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

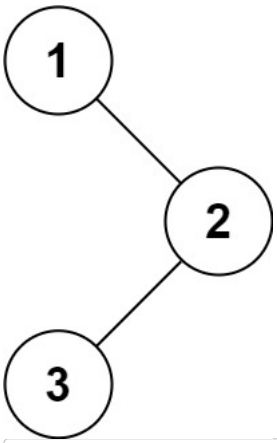
- `1 <= n <= 45`

`Binets Method solve the fibonacci in O(logn).`

94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:




```
Input: root = [1,null,2,3]
Output: [1,3,2]
```

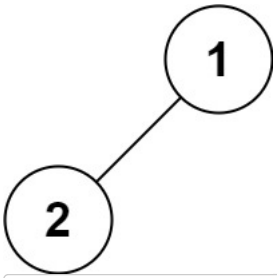
Example 2:

```
Input: root = []
Output: []
```

Example 3:

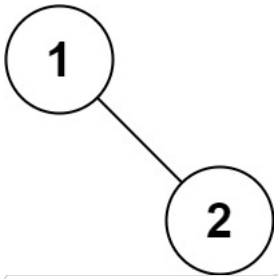
```
Input: root = [1]
Output: [1]
```

Example 4:




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

Example 5:




```
Input: root = [1,null,2]
Output: [1,2]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 100]`.
- `-100 <= Node.val <= 100`

Follow up:

Recursive solution is trivial, could you do it iteratively?

Inorder Traversal (Recursion)

```
void solve(vector<int>& ans, TreeNode* root)
{
    if(root==NULL)
        return;

    solve(ans, root->left);
    ans.push_back(root->val);

    solve(ans, root->right);

}
```

Inorder Traversal (Iterative)

```
stack<TreeNode* > s;
        TreeNode* curr = root;
        while(!s.empty() || curr!=NULL)
        {
            while(curr!=NULL)
            {
                s.push(curr);
                curr = curr->left;
            }
            curr = s.top();
            s.pop();
            ans.push_back(curr->val);
            curr = curr->right;

        }
```

---

101. Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following `[1,2,2,null,3,null,3]` is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Follow up: Solve it both recursively and iteratively.

```
else if (l && !r)
         return false;

else if (!l && r)
     return false;

// We can use xor rather than two above and statements.
else if (!l ^ !r)
     return false;
```

---

104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

DFS

```
void DFS(TreeNode* root)
{
    if(root)
    {
        cout << root << " ";
        DFS(root->left);
        DFS(root->right);
    }
}
```

BFS

```
queue<TreeNode* > q;
   q.push(root);
   while (!q.empty())
     {
      TreeNode* node = q.front();
      cout << node->data << " ";
      q.pop();
      if (node->left != NULL)
         q.push(node->left);
      if (node->right != NULL)
         q.push(node->right);
   }
```

108. Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

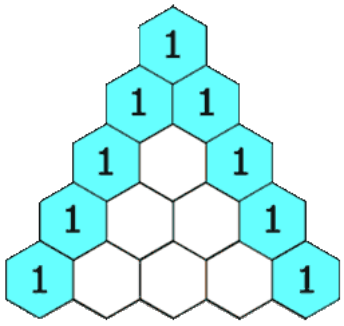
Example:

```
Given the sorted array: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

      0
     / \
   -3   9
   /   /
 -10  5
```

- AVL Tree is the binary search tree having the property of self balancing. Rotation is used for balancing the AVL tree.
- Red-Black Tree is similar to AVL but used when their is lot of insertion and deletion to reduce the rotation.

118. Pascal's Triangle

Given a non-negative integer *numRows*, generate the first *numRows* of Pascal's triangle.



In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

```
Input: 5
Output:
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

`if we use ~ vector<vector<int>> ans;`

```
 for(int i=0; i<numRows; i++)
        {
            vector<int> temp;
            for(int j=0; j<i+1; j++)
            {
                if(j==0 || j==i)
                    temp.push_back(1);
                else
                    temp.push_back(ans[i-1][j-1] + ans[i-1][j]);
            }
            ans.push_back(temp);
        }
```

`if we use ~ vector<vector<int>> ans(numRows);`

```
for(int i=0; i<numRows; i++)
        {
            for(int j=0; j<i+1; j++)
            {
                if(j==0 || j==i)
                    ans[i].push_back(1);
                else
                    ans[i].push_back(ans[i-1][j-1] + ans[i-1][j]);
            }
        }
```

---

131. Palindrome Partitioning

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of *s*.

Example:

```
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]
```

Copy substring

- `s.substr(from_index, length)`
- `s.substr(from_index to end)`

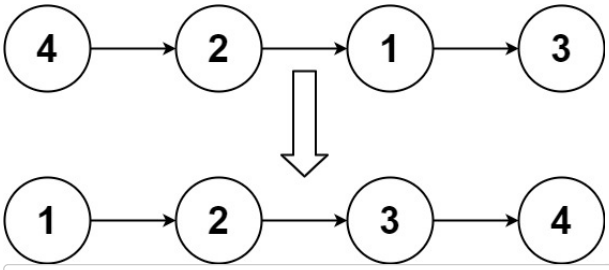
---

148. Sort List

Given the `head` of a linked list, return *the list after sorting it in **ascending order***.

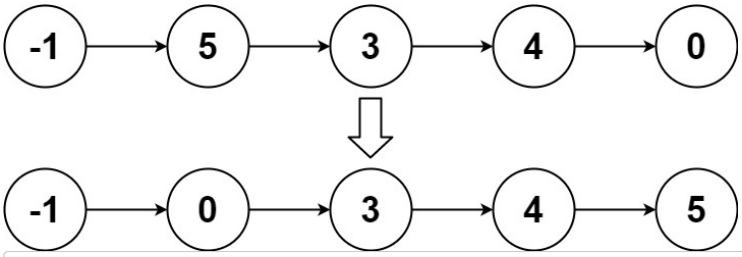
Follow up: Can you sort the linked list in `O(n logn)` time and `O(1)` memory (i.e. constant space)?

Example 1:




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

Example 2:




```
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is in the range `[0, 5 * 10^4]`.
- `-10^5 <= Node.val <= 10^5`

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

// ListNode* merge(ListNode* a, ListNode* b, int a_len, int b_len)
// {
//     ListNode* head;
//     if(a->val > b->val)
//     {
//         head = b;
//         b = b->next;
//         b_len--;
//     }
//     else
//     {
//         head = a;
//         a = a->next;
//         a_len--;
//     }

//     ListNode* temp = head;
//     while(a_len && b_len)
//     {
//         if(a->val >= b->val)
//         {
//             temp->next = b;
```

```
//             b = b->next;
//             b_len--;
//         }
//         else
//         {
//             temp->next = a;
//             a = a->next;
//             a_len--;
//         }
//         temp = temp->next;
//     }
//     while(a_len--)
//     {
//         temp->next = a;
//         a = a->next;
//         temp = temp->next;
//     }

//     while(b_len--)
//     {
//         temp->next = b;
//         b = b->next;
//         temp = temp->next;
//     }
//     temp->next = NULL;
//     return head;
// }

ListNode* merge(ListNode* a, ListNode* b)
{
    ListNode* head = new ListNode(0);
    ListNode* curr = head;

    while(a && b)
    {
        if(a->val > b->val)
        {
            curr->next = b;
            b = b->next;
        }
        else
        {
            curr->next = a;
            a = a->next;
        }
        curr = curr->next;
    }

    if(a)
        curr->next = a;

    if(b)
        curr->next = b;

    return head->next;
}

                                // ** Using length to create partition **
// ListNode* merge_sort(ListNode* node, int length)
// {
//     if(length==0 || length==1)
//         return node;

//     int mid = ceil(double(length)/2);
//     ListNode* temp = node;
//     ListNode* pre;
//     int count = mid;
//     while(count--)
//     {
//         pre = temp;
//         temp = temp->next;
//     }
//     pre->next = NULL;
```

```
//     ListNode* left = merge_sort(node, mid);
//     ListNode* right = merge_sort(temp, length-mid);
//     return merge(left, right, mid, length-mid);
// }

                                // ** Using fast and Slow pointers **
ListNode* merge_sort(ListNode* node)
{
    if(node==NULL || node->next==NULL)
        return node;

    ListNode* ahead = node;
    ListNode* curr = node;
    ListNode* pre = node;
    while(ahead != NULL && ahead->next != NULL)
    {
        cout << ahead->val << " ";
        pre = curr;
        curr = curr->next;
        ahead = ahead->next->next;
    }
    pre->next = NULL;

    ListNode* left = merge_sort(node);
    ListNode* right = merge_sort(curr);
    return merge(left, right);
}


class Solution {
public:
    ListNode* sortList(ListNode* head) {
                                    // ** Space O(n) and Time O(nlogn) **
        // vector<int> hash;
        // while(head)
        // {
        //     hash.push_back(head->val);
        //     head = head->next;
        // }
        // sort(hash.begin(), hash.end());
        // ListNode* ans = NULL;
        // ListNode* pre = NULL;
        // ListNode* curr;
        // for(int i=0; i<hash.size(); i++)
        // {
        //     curr = new ListNode(hash[i]);
        //     if(i==0)
        //         ans = curr;
        //     if(pre)
        //         pre->next = curr;
        //     pre = curr;
        // }
        // return ans;

                                    // ** Space O(1) and Time O(nlogn) **
        return merge_sort(head);
    }
};
```

155. Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- getMin() -- Retrieve the minimum element in the stack.

Example 1:

```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Constraints:

- Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.

We cannot insert duplicate items in set

Give minimum element in vector with O(n) time complexity `min_element(v.begin(), v.end())`

Using Stack O(1)

```
class MinStack {
public:
    stack<pair<int, int>> st;

    MinStack() {
    }

    void push(int x) {
        int mini;
        if(st.empty())
            mini = x;
        else
            mini = min(st.top().first, x);
        st.push({mini, x});

    }

    void pop() {
        st.pop();
    }

    int top() {
        return st.top().second;
    }

    int getMin() {
        return st.top().first;
    }
};
```

Using Two Vectors O(1) || Using Single Vector O(n)

```
class MinStack {
public:
    /** initialize your data structure here. */
    vector<int> stack;
    vector<int> small;

    MinStack() {
    }

    void push(int x) {
        stack.push_back(x);
        if(!small.size())
            small.push_back(x);
        else if(small[small.size()-1]>=x)
            small.push_back(x);
    }

    void pop() {
        if(small[small.size()-1] == stack[stack.size()-1])
            small.pop_back();
        stack.pop_back();
    }

    int top() {
        return stack[stack.size()-1];
    }

    int getMin() {
        return small[small.size()-1];

                //Time complexity is O(n)
        // int min = *min_element(stack.begin(), stack.end());
        // return min;
    }
};
```

169. Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** `⌊ n/2 ⌋` times.

You may assume that the array is non-empty and the majority element always exist in the array.

Example 1:

```
Input: [3,2,3]
Output: 3
```

Example 2:

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

`Boyer–Moore Voting Algorithm`

189. Rotate Array

Given an array, rotate the array to the right by *k* steps, where *k* is non-negative.

Follow up:

- Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
- Could you do it in-place with O(1) extra space?

Example 1:

```
Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Constraints:

- `1 <= nums.length <= 2 * 10^4`
- `-2^31 <= nums[i] <= 2^31 - 1`
- `0 <= k <= 10^5`

`reverse(nums.begin(), nums.end());` Here, first index is included but the last index is not included.

---

190. Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation (https://en.wikipedia.org/wiki/Two%27s_complement). Therefore, in **Example 2** above, the input represents the signed integer `-3` and the output represents the signed integer `-1073741825`.

Follow up:

If this function is called many times, how would you optimize it?

Example 1:

```
Input: n = 00000010100101000001111010011100
Output:    964176192 (00111001011110000010100101000000)
Explanation: The input binary string 00000010100101000001111010011100 represents the unsigned integer 43261596, so return 964176
92 which its binary representation is 00111001011110000010100101000000.
```

Example 2:

```
Input: n = 11111111111111111111111111111101
Output:   3221225471 (10111111111111111111111111111111)
Explanation: The input binary string 11111111111111111111111111111101 represents the unsigned integer 4294967293, so return 3221
25471 which its binary representation is 10111111111111111111111111111111.
```

Constraints:

- The input must be a **binary string** of length `32`

`n>>1` is faster than `n/=2`

Reversing bits

```
unsigned int ans = 0;
        int temp = 0;
        for(int i=0; i<32; i++)
        {
            temp = n%2;
            ans *= 2;
            ans += temp;
            n = n>>1;
        }
        return ans;
```

191. Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight (http://en.wikipedia.org/wiki/Hamming_weigh

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation (https://en.wikipedia.org/wiki/Two%27s_complement). Therefore, in **Example 3** above, the input represents the signed integer. −3 .

Follow up: If this function is called many times, how would you optimize it?

Example 1:

```
Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total of three '1' bits.
```

Example 2:

```
Input: n = 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000000010000000 has a total of one '1' bit.
```

Example 3:

```
Input: n = 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111111111111101 has a total of thirty one '1' bits.
```

Constraints:

- The input must be a **binary string** of length 32

`__builtin_popcount(n)` returns number of set bits in x

201. Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

Example 1:

```
Input: [5,7]
Output: 4
```

Example 2:

```
Input: [0,1]
Output: 0
```

`m&(1<<i)` returns 2^i if ith bit is set in m, else return 0

202. Happy Number

Write an algorithm to determine if a number `n` is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat tl process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1. Those numbers for which this process **ends in 1** are hap numbers.

Return True if `n` is a happy number, and False if not.

Example:

```
Input: 19
Output: true
Explanation:
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1
```

Tortoise and Hare(Floyd's Cycle Detection) Algorithm

204. Count Primes

Count the number of prime numbers less than a non-negative number, `n`.

Example 1:

```
Input: n = 10
Output: 4
Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5, 7.
```

Example 2:

```
Input: n = 0
Output: 0
```

Example 3:

```
Input: n = 1
Output: 0
```

Constraints:

- `0 <= n <= 5 * 10^6`

`memset(visited, 0, sizeof(visited))` initialize entire array with 0

- array is faster than vector

Seive

```
        if(n<=1)
            return 0;
        vector<int> visited(n+1,0);
        visited[1] = -1;
        int count = 1;

        for(int i=2; i<=sqrt(n); i++)
        {
            if(visited[i] == 0)
            {
                for(int j=2*i; j<n; j=j+i)
                {
                    if(visited[j]==0)
                    {
                        visited[j] = -1;
                        count++;
                    }
                }
            }
        }
        return n-count-1;
```

Better version of seive

```
        if(n<=2)
            return 0;
        vector<int> visited(n/2, 0);
        int non_prime;
        if(n%2)
            non_prime = n/2 + 1;
        else
            non_prime = n/2;

        for(int i=3; i<sqrt(n); i=i+2)
        {
            if(visited[i/2]==0)
            {
                for(int j=3*i; j<n; j=j+2*i)
                {
                    if(!visited[j/2])
                    {
                        visited[j/2]=1;
                        non_prime++;
                    }
                }
            }
        }

        return n-non_prime;
```

207. Course Schedule

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-1` .

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Constraints:

- The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented (https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
- You may assume that there are no duplicate edges in the input prerequisites.
- `1 <= numCourses <= 10^5`

Topological Sort

208. Implement Trie (Prefix Tree)

Implement a trie with `insert`, `search`, and `startsWith` methods.

Example:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // returns true
trie.search("app");     // returns false
trie.startsWith("app"); // returns true
trie.insert("app");
trie.search("app");     // returns true
```

Note:

- You may assume that all inputs are consist of lowercase letters `a-z`.
- All inputs are guaranteed to be non-empty strings.

Check for the prefix

```
auto c = mismatch(prefix.begin(), prefix.end(), s.begin());
            if(c.first == prefix.end())
                return true;
```

TRIE

- Creating complexity O(n*m)
- Seach word complexity O(m)
- Search prefix complexity O(m)

Dual Mapping

- Creating complexity O(n)
- Seach word complexity O(n*m)
- Search prefix complexity O(n*m)

215. Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

```
Input: [3,2,1,5,6,4] and k = 2
Output: 5
```

Example 2:

```
Input: [3,2,3,1,2,4,5,5,6] and k = 4
Output: 4
```

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

O(nlogn) Sort entire array

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

O(nlog(k)) Sort first k elements

```
partial_sort(nums.begin(), nums.begin()+k, nums.end());
```

O(n) Set the position of nth element(every element before n is smaller than nth element and all elements after n are larger than n
element)

```
nth_element(nums.begin(), nums.end()-k, nums.end());
```

---

217. Contains Duplicate

Given an array of integers, find if the array contains any duplicates.

Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

Example 1:

```
Input: [1,2,3,1]
Output: true
```

Example 2:

```
Input: [1,2,3,4]
Output: false
```

Example 3:

```
Input: [1,1,1,3,3,4,3,2,4,2]
Output: true
```

Fast I/O

```
ios_base :: sync_with_stdio(false);
        cin.tie(NULL);
        cout.tie(NULL);
```

---

350. Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.

Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2,2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [4,9]
```

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?
- What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

```
map<int, int> hash;
cout << hash[2];
```

`Output – 0` **Because by default every value in map is set to 0(zero)**
