Leetcode Top Answers

Created By Jing Connect on LinkedIn

This document will be updated here.

Two Sum(1)

Answer 1

```
vector<int> twoSum(vector<int> &numbers, int target)
{
    //Key is the number and value is its index in the vector.
    unordered_map<int, int> hash;
    vector<int> result;
    for (int i = 0; i < numbers.size(); i++) {
        int numberToFind = target - numbers[i];

            //if numberToFind is found in map, return them
        if (hash.find(numberToFind) != hash.end()) {
                    //+1 because indices are NOT zero based
            result.push_back(hash[numberToFind] + 1);
            result.push_back(i + 1);
            return result;
        }

            //number was not found. Put it in the map.
        hash[numbers[i]] = i;
    }
    return result;
}
```

written by naveed.zafar original link here

Answer 2

Hi, this is my accepted JAVA solution. It only go through the list once. It's shorter and easier to understand. Hope this can help someone. Please tell me if you know how to make this better :)

```
public int[] twoSum(int[] numbers, int target) {
    int[] result = new int[2];
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < numbers.length; i++) {
        if (map.containsKey(target - numbers[i])) {
            result[1] = i + 1;
            result[0] = map.get(target - numbers[i]);
            return result;
        }
        map.put(numbers[i], i + 1);
    }
    return result;
}
```

written by jiaming2 original link here

Answer 3

Hello! At first glance, this can easily be solved through a quadratic algorithm BUT it

can actually be done in linear time. The idea here is to use a map to keep track of the needed RIGHT operand in order for the sum to meet its target. So, we iterate through the array, and store the index of the LEFT operand as the value in the map whereas the NEEDED RIGHT operand is used as the key. When we do encounter the right operand somewhere in the array, the answer is considered to be found! We just return the indices as instructed. :]

Feel free to let me know should you have any queries for me OR if this can be improved upon!

```
public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> tracker = new HashMap<Integer, Integer>();
        int len = nums.length;
        for(int i = 0; i < len; i++){
            if(tracker.containsKey(nums[i])){
                int left = tracker.get(nums[i]);
                return new int[]{left+1, i+1};
            }else{
                tracker.put(target - nums[i], i);
            }
        }
        return new int[2];
    }
```

written by waisuan original link here

Add Two Numbers(2)

Answer 1

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode c1 = l1;
        ListNode c2 = l2;
        ListNode sentinel = new ListNode(0);
        ListNode d = sentinel;
        int sum = 0;
        while (c1 != null || c2 != null) {
            sum /= 10;
            if (c1 != null) {
                sum += c1.val;
                c1 = c1.next;
            }
            if (c2 != null) {
                sum += c2.val;
                c2 = c2.next;
            }
            d.next = new ListNode(sum % 10);
            d = d.next;
        }
        if (sum / 10 == 1)
            d.next = new ListNode(1);
        return sentinel.next;
    }
}
```

written by potpie original link here

Answer 2

```
ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
    ListNode preHead(0), *p = &preHead;
    int extra = 0;
    while (l1 || l2 || extra) {
        int sum = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + extra;
        extra = sum / 10;
        p->next = new ListNode(sum % 10);
        p = p->next;
        l1 = l1 ? l1->next : l1;
        l2 = l2 ? l2->next : l2;
    }
    return preHead.next;
}
```

written by ce2 original link here

Answer 3

Two things to make the code simple:

1. Whenever one of the two *ListNode* is null, replace it with 0.
2. Keep the while loop going when at least one of the three conditions is met.

Let me know if there is something wrong. Thanks.

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode prev = new ListNode(0);
        ListNode head = prev;
        int carry = 0;
        while (l1 != null || l2 != null || carry != 0) {
            ListNode cur = new ListNode(0);
            int sum = ((l2 == null) ? 0 : l2.val) + ((l1 == null) ? 0 : l1.val) +
carry;
            cur.val = sum % 10;
            carry = sum / 10;
            prev.next = cur;
            prev = cur;

            l1 = (l1 == null) ? l1 : l1.next;
            l2 = (l2 == null) ? l2 : l2.next;
        }
        return head.next;
    }
}
```

written by dchen0215 original link here

Longest Substring Without Repeating Characters(3)

Answer 1

the basic idea is, keep a hashmap which stores the characters in string as keys and their positions as values, and keep two pointers which define the max substring. move the right pointer to scan through the string , and meanwhile update the hashmap. If the character is already in the hashmap, then move the left pointer to the right of the same character last found. Note that the two pointers can only move forward.

```
public int lengthOfLongestSubstring(String s) {
     if (s.length()==0) return 0;
     HashMap<Character, Integer> map = new HashMap<Character, Integer>();
     int max=0;
     for (int i=0, j=0; i<s.length(); ++i){
         if (map.containsKey(s.charAt(i))){
             j = Math.max(j,map.get(s.charAt(i))+1);
         }
         map.put(s.charAt(i),i);
         max = Math.max(max,i-j+1);
     }
     return max;
 }
```

written by cbmbbz original link here

Answer 2

```
/**
 * Solution (DP, O(n)):
 *
 * Assume L[i] = s[m...i], denotes the longest substring without repeating
 * characters that ends up at s[i], and we keep a hashmap for every
 * characters between m ... i, while storing <character, index> in the
 * hashmap.
 * We know that each character will appear only once.
 * Then to find s[i+1]:
 * 1) if s[i+1] does not appear in hashmap
 *    we can just add s[i+1] to hash map. and L[i+1] = s[m...i+1]
 * 2) if s[i+1] exists in hashmap, and the hashmap value (the index) is k
 *    let m = max(m, k), then L[i+1] = s[m...i+1], we also need to update
 *    entry in hashmap to mark the latest occurency of s[i+1].
 *
 * Since we scan the string for only once, and the 'm' will also move from
 * beginning to end for at most once. Overall complexity is O(n).
 *
 * If characters are all in ASCII, we could use array to mimic hashmap.
 */

int lengthOfLongestSubstring(string s) {
    // for ASCII char sequence, use this as a hashmap
    vector<int> charIndex(256, -1);
    int longest = 0, m = 0;

    for (int i = 0; i < s.length(); i++) {
        m = max(charIndex[s[i]] + 1, m);    // automatically takes care of -1 cas
e
        charIndex[s[i]] = i;
        longest = max(longest, i - m + 1);
    }

    return longest;
}
```

Hope you like it :)

written by dragonmigo original link here

Answer 3

if only use DP, it's an O(n*n) solution, adding a map to get O(n).

```
class Solution {
    public:
        int lengthOfLongestSubstring(string s) {
            if(s.size()<2) return s.size();
            int d=1, maxLen=1;
            unordered_map<char,int> map;
            map[s[0]]=0;
            for(int i=1;i<s.size();i++)
            {
                if(map.count(s[i])==0 || map[s[i]]<i-d)
                    d++;
                else
                    d= i- map[s[i]];
                map[s[i]]=i;
                if(d>maxLen)
                    maxLen = d;
            }
            return maxLen;
        }
    };
```

written by heiyanbin original link here

Median of Two Sorted Arrays(4)

Answer 1

Given a sorted array A with length m, we can split it into two part:

```
{ A[0], A[1], ... , A[i - 1] } | { A[i], A[i + 1], ... , A[m - 1] }
```

All elements in right part are greater than elements in left part.

The left part has "i" elements, and right part has "m - i" elements.

There are "m + 1" kinds of splits. (i = 0 ~ m)

When i = 0, the left part has "0" elements, right part has "m" elements.

When i = m, the left part has "m" elements, right part has "0" elements.

For array B, we can split it with the same way:

```
{ B[0], B[1], ... , B[j - 1] } | { B[j], B[j + 1], ... , B[n - 1] }
```

The left part has "j" elements, and right part has "n - j" elements.

Put A's left part and B's left part into one set. (Let's name this set "LeftPart")

Put A's right part and B's right part into one set. (Let's name this set"RightPart")

```
           LeftPart           |            RightPart
{ A[0], A[1], ... , A[i - 1] } | { A[i], A[i + 1], ... , A[m - 1] }
{ B[0], B[1], ... , B[j - 1] } | { B[j], B[j + 1], ... , B[n - 1] }
```

If we can ensure:

```
1) LeftPart's length == RightPart's length (or RightPart's length + 1)

2) All elements in RightPart are greater than elements in LeftPart.
```

then we split all elements in {A, B} into two parts with eqaul length, and one part is always greater than the other part. Then the median can be easily found.

To ensure these two condition, we just need to ensure:

```
(1) i + j == m - i + n - j (or: m - i + n - j + 1)

    if n >= m, we just need to set:

          i = 0 ~ m, j = (m + n + 1) / 2 - i

(2) B[j - 1] <= A[i] and A[i - 1] <= B[j]

    considering edge values, we need to ensure:

          (j == 0 or i == m or B[j - 1] <= A[i]) and

              (i == 0 or j == n or A[i - 1] <= B[j])
```

So, all we need to do is:

```
Search i from 0 to m, to find an object "i" to meet condition (1) and (2) above.
```

And we can do this search by binary search. How?

If B[j0 - 1] > A[i0], then the object "ix" can't be in [0, i0]. Why?

```
Because if ix < i0, then jx = (m + n + 1) / 2 - ix > j0,

then B[jx - 1] >= B[j0 - 1] > A[i0] >= A[ix]. This violates

the condition (2). So ix can't be less than i0.
```

And if A[i0 - 1] > B[j0], then the object "ix" can't be in [i0, m].

So we can do the binary search following steps described below:

```
1. set imin, imax = 0, m, then start searching in [imin, imax]

2. i = (imin + imax) / 2; j = (m + n + 1) / 2 - i

3. if B[j - 1] > A[i]: continue searching in [i + 1, imax]
   elif A[i - 1] > B[j]: continue searching in [imin, i - 1]
   else: bingo! this is our object "i"
```

When the object i is found, the median is:

```
max(A[i - 1], B[j - 1]) (when m + n is odd)

or (max(A[i - 1], B[j - 1]) + min(A[i], B[j])) / 2 (when m + n is even)
```

Below is the accepted code:

```
def median(A, B):
    m, n = len(A), len(B)

    if m > n:
        A, B, m, n = B, A, n, m

    imin, imax, half_len = 0, m, (m + n + 1) / 2
    while imin <= imax:
        i = (imin + imax) / 2
        j = half_len - i
        if j > 0 and i < m and B[j - 1] > A[i]:
            imin = i + 1
        elif i > 0 and j < n and A[i - 1] > B[j]:
            imax = i - 1
        else:
            if i == 0:
                num1 = B[j - 1]
            elif j == 0:
                num1 = A[i - 1]
            else:
                num1 = max(A[i - 1], B[j - 1])

            if (m + n) % 2 == 1:
                return num1

            if i == m:
                num2 = B[j]
            elif j == n:
                num2 = A[i]
            else:
                num2 = min(A[i], B[j])

            return (num1 + num2) / 2.0
```

written by MissMary original link here

Answer 2

This problem is notoriously hard to implement due to all the corner cases. Most implementations consider odd-lengthed and even-lengthed arrays as two different cases and treat them separately. As a matter of fact, with a little mind twist. These two cases can be combined as one, leading to a very simple solution where (almost) no special treatment is needed.

First, let's see the concept of 'MEDIAN' in a slightly unconventional way. That is:

> "**if we cut the sorted array to two halves of EQUAL LENGTHS, then median is the AVERAGE OF Min(lower*half) and Max(upper*half), i.e. the two numbers immediately next to the cut**".

For example, for [2 3 5 7], we make the cut between 3 and 5:

```
[2 3 / 5 7]
```

then the median = (3+5)/2. **Note that I'll use '/' to represent a cut, and (number / number) to represent a cut made through a number in this article**.

for [2 3 4 5 6], we make the cut right through 4 like this:

[2 3 (4/4) 5 7]

Since we split 4 into two halves, we say now both the lower and upper subarray contain 4. This notion also leads to the correct answer: (4 + 4) / 2 = 4;

For convenience, let's use L to represent the number immediately left to the cut, and R the right counterpart. In [2 3 5 7], for instance, we have L = 3 and R = 5, respectively.

We observe the index of L and R have the following relationship with the length of the array N:

```
N        Index of L / R
1              0 / 0
2              0 / 1
3              1 / 1
4              1 / 2
5              2 / 2
6              2 / 3
7              3 / 3
8              3 / 4
```

It is not hard to conclude that index of L = (N-1)/2, and R is at N/2. Thus, the median can be represented as

```
(L + R)/2 = (A[(N-1)/2] + A[N/2])/2
```

To get ready for the two array situation, let's add a few imaginary 'positions' (represented as #'s) in between numbers, and treat numbers as 'positions' as well.

```
[6 9 13 18]  ->   [# 6 # 9 # 13 # 18 #]    (N = 4)
position index     0 1 2 3 4 5  6 7  8     (N_Position = 9)

[6 9 11 13 18]->   [# 6 # 9 # 11 # 13 # 18 #]   (N = 5)
position index      0 1 2 3 4 5  6 7  8 9 10    (N_Position = 11)
```

As you can see, there are always exactly 2*N+1 'positions' regardless of length N. Therefore, the middle cut should always be made on the Nth position (0-based). Since index(L) = (N-1)/2 and index(R) = N/2 in this situation, we can infer that **index(L) = (CutPosition-1)/2, index(R) = (CutPosition)/2** .

Now for the two-array case:

```
A1: [# 1 # 2 # 3 # 4 # 5 #]    (N1 = 5, N1_positions = 11)

A2: [# 1 # 1 # 1 # 1 #]     (N2 = 4, N2_positions = 9)
```

Similar to the one-array problem, we need to find a cut that divides the two arrays each into two halves such that

> "any number in the two left halves" <= "any number in the two right halves".

We can also make the following observationsï¼š

1. There are 2*N1* + 2N2 + 2 position altogether. Therefore, there must be exactly N1 + N2 positions on each side of the cut, and 2 positions directly on the cut.
2. Therefore, when we cut at position C2 = K in A2, then the cut position in A1 must be C1 = N1 + N2 - k. For instance, if C2 = 2, then we must have C1 = 4 + 5 - C2 = 7.

   ```
   [# 1 # 2 # 3 # (4/4) # 5 #]

   [# 1 / 1 # 1 # 1 #]
   ```

3. When the cuts are made, we'd have two L's and two R's. They are

   ```
   L1 = A1[(C1-1)/2]; R1 = A1[C1/2];
   L2 = A2[(C2-1)/2]; R2 = A2[C2/2];
   ```

In the above example,

```
L1 = A1[(7-1)/2] = A1[3] = 4; R1 = A1[7/2] = A1[3] = 4;
L2 = A2[(2-1)/2] = A2[0] = 1; R2 = A1[2/2] = A1[1] = 1;
```

Now how do we decide if this cut is the cut we want? Because L1, L2 are the greatest numbers on the left halves and R1, R2 are the smallest numbers on the right, we only need

```
L1 <= R1 && L1 <= R2 && L2 <= R1 && L2 <= R2
```

to make sure that any number in lower halves <= any number in upper halves. As a matter of fact, since L1 <= R1 and L2 <= R2 are naturally guaranteed because A1 and A2 are sorted, we only need to make sure:

L1 <= R2 and L2 <= R1.

Now we can use simple binary search to find out the result.

```
If we have L1 > R1, it means there are too many large numbers on the left half of
A1, then we must move C1 to the left (i.e. move C2 to the right);
If L2 > R1, then there are too many large numbers on the left half of A2, and we
must move C2 to the left.
Otherwise, this cut is the right one.
After we find the cut, the medium can be computed as (max(L1, L2) + min(R1, R2)) /
2;
```

Two side notes:

A. since C1 and C2 can be mutually determined from each other, we might as well select the shorter array (say A2) and only move C2 around, and calculate C1 accordingly. That way we can achieve a run-time complexity of O(log(min(N1, N2)))

B. The only edge case is when a cut falls on the 0th(first) or the 2*N*th(last) position. *For instance, if C2* = 2N2, then R2 = A2[2*N2/2] = A2[N2], which exceeds the boundary of the array. To solve this problem, we can imagine that both A1 and A2 actually have two extra elements, INT_MAX at A[-1] and INT_MAX at A[N]. These additions don't change the result, but make the implementation easier: If any L falls out of the left boundary of the array, then L = INT_MIN, and if any R falls out of the right boundary, then R = INT_MAX.

I know that was not very easy to understand, but all the above reasoning eventually boils down to the following concise code:

```
 double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int N1 = nums1.size();
    int N2 = nums2.size();
    if (N1 < N2) return findMedianSortedArrays(nums2, nums1);   // Make sure A2 i
s the shorter one.

    if (N2 == 0) return ((double)nums1[(N1-1)/2] + (double)nums1[N1/2])/2;  // If
A2 is empty

    int lo = 0, hi = N2 * 2;
    while (lo <= hi) {
        int mid2 = (lo + hi) / 2;   // Try Cut 2
        int mid1 = N1 + N2 - mid2;  // Calculate Cut 1 accordingly

        double L1 = (mid1 == 0) ? INT_MIN : nums1[(mid1-1)/2];  // Get L1, R1, L2
, R2 respectively
        double L2 = (mid2 == 0) ? INT_MIN : nums2[(mid2-1)/2];
        double R1 = (mid1 == N1 * 2) ? INT_MAX : nums1[(mid1)/2];
        double R2 = (mid2 == N2 * 2) ? INT_MAX : nums2[(mid2)/2];

        if (L1 > R2) lo = mid2 + 1;     // A1's lower half is too big; need to mo
ve C1 left (C2 right)
        else if (L2 > R1) hi = mid2 - 1;    // A2's lower half too big; need to m
ove C2 left.
        else return (max(L1,L2) + min(R1, R2)) / 2; // Otherwise, that's the righ
t cut.
    }
    return -1;
}
```

If you have any suggestions to make the logic and implementation even more cleaner. Please do let me know!

written by stellari original link here

Answer 3

Binary search. Call 2 times getkth and k is about half of (m + n). Every time call getkth can reduce the scale k to its half. So the time complexity is log(m + n).

```
class Solution {
public:
    int getkth(int s[], int m, int l[], int n, int k){
        // let m <= n
        if (m > n)
            return getkth(l, n, s, m, k);
        if (m == 0)
            return l[k - 1];
        if (k == 1)
            return min(s[0], l[0]);

        int i = min(m, k / 2), j = min(n, k / 2);
        if (s[i - 1] > l[j - 1])
            return getkth(s, m, l + j, n - j, k - j);
        else
            return getkth(s + i, m - i, l, n, k - i);
        return 0;
    }

    double findMedianSortedArrays(int A[], int m, int B[], int n) {
        int l = (m + n + 1) >> 1;
        int r = (m + n + 2) >> 1;
        return (getkth(A, m ,B, n, l) + getkth(A, m, B, n, r)) / 2.0;
    }
};
```

written by vaputa original link here

Longest Palindromic Substring(5)

Answer 1

```
string longestPalindrome(string s) {
    if (s.empty()) return "";
    if (s.size() == 1) return s;
    int min_start = 0, max_len = 1;
    for (int i = 0; i < s.size();) {
      if (s.size() - i <= max_len / 2) break;
      int j = i, k = i;
      while (k < s.size()-1 && s[k+1] == s[k]) ++k; // Skip duplicate characters.
      i = k+1;
      while (k < s.size()-1 && j > 0 && s[k + 1] == s[j - 1]) { ++k; --j; } // Ex
pand.
      int new_len = k - j + 1;
      if (new_len > max_len) { min_start = j; max_len = new_len; }
    }
    return s.substr(min_start, max_len);
}
```

written by hh1985 original link here

Answer 2

The performance is pretty good, surprisingly.

```
public class Solution {
private int lo, maxLen;

public String longestPalindrome(String s) {
    int len = s.length();
    if (len < 2)
        return s;

    for (int i = 0; i < len-1; i++) {
        extendPalindrome(s, i, i);  //assume odd length, try to extend Palindrome
as possible
        extendPalindrome(s, i, i+1); //assume even length.
    }
    return s.substring(lo, lo + maxLen);
}

private void extendPalindrome(String s, int j, int k) {
    while (j >= 0 && k < s.length() && s.charAt(j) == s.charAt(k)) {
        j--;
        k++;
    }
    if (maxLen < k - j - 1) {
        lo = j + 1;
        maxLen = k - j - 1;
    }
}}
```

written by jinwu original link here

Answer 3

```
class Solution {
public:
    std::string longestPalindrome(std::string s) {
        if (s.size() < 2)
            return s;
        int len = s.size(), max_left = 0, max_len = 1, left, right;
        for (int start = 0; start < len && len - start > max_len / 2;) {
            left = right = start;
            while (right < len - 1 && s[right + 1] == s[right])
                ++right;
            start = right + 1;
            while (right < len - 1 && left > 0 && s[right + 1] == s[left - 1]) {
                ++right;
                --left;
            }
            if (max_len < right - left + 1) {
                max_left = left;
                max_len = right - left + 1;
            }
        }
        return s.substr(max_left, max_len);
    }
};
```

written by prime_tang original link here

ZigZag Conversion(6)

Answer 1

Create nRows StringBuffers, and keep collecting characters from original string to corresponding StringBuffer. Just take care of your index to keep them in bound.

```
public String convert(String s, int nRows) {
    char[] c = s.toCharArray();
    int len = c.length;
    StringBuffer[] sb = new StringBuffer[nRows];
    for (int i = 0; i < sb.length; i++) sb[i] = new StringBuffer();

    int i = 0;
    while (i < len) {
        for (int idx = 0; idx < nRows && i < len; idx++) // vertically down
            sb[idx].append(c[i++]);
        for (int idx = nRows-2; idx >= 1 && i < len; idx--) // obliquely up
            sb[idx].append(c[i++]);
    }
    for (int idx = 1; idx < sb.length; idx++)
        sb[0].append(sb[idx]);
    return sb[0].toString();
}
```

written by dylan_yu original link here

Answer 2

```
/*n=numRows
Î”=2n-2    1                          2n-1                        4n-3
Î”=        2                  2n-2    2n                  4n-4    4n-2
Î”=        3              2n-3        2n+1              4n-5      .
Î”=        .          .               .                 .         .
Î”=        .      n+2                 .           3n              .
Î”=        n-1 n+1                    3n-3    3n-1                5n-5
Î”=2n-2    n                          3n-2                        5n-4
*/
```

that's the zigzag pattern the question asked! Be careful with nR=1 && nR=2

my 16ms code in c++:

```
class Solution {
public:
    string convert(string s, int numRows) {
        string result="";
        if(numRows==1)
            return s;
        int step1,step2;
        int len=s.size();
        for(int i=0;i<numRows;++i){
            step1=(numRows-i-1)*2;
            step2=(i)*2;
            int pos=i;
            if(pos<len)
                result+=s.at(pos);
            while(1){
                pos+=step1;
                if(pos>=len)
                    break;
                if(step1)
                    result+=s.at(pos);
                pos+=step2;
                if(pos>=len)
                    break;
                if(step2)
                    result+=s.at(pos);
            }
        }
        return result;
    }
};
```

written by HelloKenLee original link here

Answer 3

The problem statement itself is unclear for many. Especially for 2-row case. "ABCD", 2 --> "ACBD". The confusion most likely is from the character placement. I would like to extend it a little bit to make ZigZag easy understood.

The example can be written as follow:

1. P.....A.....H.....N
2. ..A..P....L..S....I...I....G
3. ....Y.....I.....R

Therefore, <ABCD, 2> can be arranged as:

1. A....C
2. ...B....D

My simple accepted code:

```
string convert(string s, int nRows) {

    if (nRows <= 1)
        return s;

    const int len = (int)s.length();
    string *str = new string[nRows];

    int row = 0, step = 1;
    for (int i = 0; i < len; ++i)
    {
        str[row].push_back(s[i]);

        if (row == 0)
            step = 1;
        else if (row == nRows - 1)
            step = -1;

        row += step;
    }

    s.clear();
    for (int j = 0; j < nRows; ++j)
    {
        s.append(str[j]);
    }

    delete[] str;
    return s;
}
```

written by enze98 original link here

Reverse Integer(7)

Answer 1

Only 15 lines. If overflow exists, the new result will not equal previous one. No flags needed. No hard code like 0xf7777777 needed. Sorry for my bad english.

```
public int reverse(int x)
{
    int result = 0;

    while (x != 0)
    {
        int tail = x % 10;
        int newResult = result * 10 + tail;
        if ((newResult - tail) / 10 != result)
        { return 0; }
        result = newResult;
        x = x / 10;
    }

    return result;
}
```

written by bitzhuwei original link here

Answer 2

```
public int reverse(int x) {
        long rev= 0;
        while( x != 0){
            rev= rev*10 + x % 10;
            x= x/10;
            if( rev > Integer.MAX_VALUE || rev < Integer.MIN_VALUE)
                return 0;
        }
        return (int) rev;
    }
```

written by kbakhit original link here

Answer 3

Throw an exception? Good, but what if throwing an exception is not an option? You would then have to re-design the function (ie, add an extra parameter).

written by caidiexunmeng original link here

String to Integer (atoi)(8)

Answer 1

I think we only need to handle four cases:

1. discards all leading whitespaces
2. sign of the number
3. overflow
4. invalid input

Is there any better solution? Thanks for pointing out!

```
int atoi(const char *str) {
    int sign = 1, base = 0, i = 0;
    while (str[i] == ' ') { i++; }
    if (str[i] == '-' || str[i] == '+') {
        sign = 1 - 2 * (str[i++] == '-');
    }
    while (str[i] >= '0' && str[i] <= '9') {
        if (base >  INT_MAX / 10 || (base == INT_MAX / 10 && str[i] - '0' > 7)) {
            if (sign == 1) return INT_MAX;
            else return INT_MIN;
        }
        base  = 10 * base + (str[i++] - '0');
    }
    return base * sign;
}
```

written by yuruofeifei original link here

Answer 2

```
public int myAtoi(String str) {
    int index = 0, sign = 1, total = 0;
    //1. Empty string
    if(str.length() == 0) return 0;

    //2. Remove Spaces
    while(str.charAt(index) == ' ' && index < str.length())
        index ++;

    //3. Handle signs
    if(str.charAt(index) == '+' || str.charAt(index) == '-'){
        sign = str.charAt(index) == '+' ? 1 : -1;
        index ++;
    }

    //4. Convert number and avoid overflow
    while(index < str.length()){
        int digit = str.charAt(index) - '0';
        if(digit < 0 || digit > 9) break;

        //check if total will be overflow after 10 times and add digit
        if(Integer.MAX_VALUE/10 < total || Integer.MAX_VALUE/10 == total && Integ
er.MAX_VALUE %10 < digit)
            return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;

        total = 10 * total + digit;
        index ++;
    }
    return total * sign;
}
```

written by lestrois original link here

Answer 3

```
int myAtoi(string str) {
    long result = 0;
    int indicator = 1;
    for(int i = 0; i<str.size();)
    {
        i = str.find_first_not_of(' ');
        if(str[i] == '-' || str[i] == '+')
            indicator = (str[i++] == '-')? -1 : 1;
        while('0'<= str[i] && str[i] <= '9')
        {
            result = result*10 + (str[i++]-'0');
            if(result*indicator >= INT_MAX) return INT_MAX;
            if(result*indicator <= INT_MIN) return INT_MIN;
        }
        return result*indicator;
    }
}
```

written by morning_color original link here

Palindrome Number(9)

Answer 1

compare half of the digits in x, so don't need to deal with overflow.

```
public boolean isPalindrome(int x) {
    if (x<0 || (x!=0 && x%10==0)) return false;
    int rev = 0;
    while (x>rev){
        rev = rev*10 + x%10;
        x = x/10;
    }
    return (x==rev || x==rev/10);
}
```

written by cbmbbz original link here

Answer 2

```
class Solution {
public:
    bool isPalindrome(int x) {
        if(x<0|| (x!=0 &&x%10==0)) return false;
        int sum=0;
        while(x>sum)
        {
            sum = sum*10+x%10;
            x = x/10;
        }
        return (x==sum)||(x==sum/10);
    }
};
```

written by gaurav5 original link here

Answer 3

```
public boolean isPalindrome(int x) {

    if (x < 0) return false;

    int p = x;
    int q = 0;

    while (p >= 10){
        q *=10;
        q += p%10;
        p /=10;
    }

    return q == x / 10 && p == x % 10;
}
```

// so the reversed version of int is always 1 time short in the factor of 10s .

in case of Int16, check 63556 will finally check if (6553 == 6355 && 6 == 63556%10) so there will have no concerns about the overflow.

written by evlstyle original link here

Regular Expression Matching(10)

Answer 1

Please refer to my blog post if you have any comment. Wildcard matching problem can be solved similarly.

```cpp
class Solution {
public:
    bool isMatch(string s, string p) {
        if (p.empty())    return s.empty();

        if ('*' == p[1])
            // x* matches empty string or at least one character: x* -> xx*
            // *s is to ensure s is non-empty
            return (isMatch(s, p.substr(2)) || !s.empty() && (s[0] == p[0] || '.'
== p[0]) && isMatch(s.substr(1), p));
        else
            return !s.empty() && (s[0] == p[0] || '.' == p[0]) && isMatch(s.subst
r(1), p.substr(1));
    }
};

class Solution {
public:
    bool isMatch(string s, string p) {
        /**
         * f[i][j]: if s[0..i-1] matches p[0..j-1]
         * if p[j - 1] != '*'
         *      f[i][j] = f[i - 1][j - 1] && s[i - 1] == p[j - 1]
         * if p[j - 1] == '*', denote p[j - 2] with x
         *      f[i][j] is true iff any of the following is true
         *      1) "x*" repeats 0 time and matches empty: f[i][j - 2]
         *      2) "x*" repeats >= 1 times and matches "x*x": s[i - 1] == x && f[
i - 1][j]
         * '.' matches any single character
         */
        int m = s.size(), n = p.size();
        vector<vector<bool>> f(m + 1, vector<bool>(n + 1, false));

        f[0][0] = true;
        for (int i = 1; i <= m; i++)
            f[i][0] = false;
        // p[0.., j - 3, j - 2, j - 1] matches empty iff p[j - 1] is '*' and p[0.
.j - 3] matches empty
        for (int j = 1; j <= n; j++)
            f[0][j] = j > 1 && '*' == p[j - 1] && f[0][j - 2];

        for (int i = 1; i <= m; i++)
            for (int j = 1; j <= n; j++)
                if (p[j - 1] != '*')
                    f[i][j] = f[i - 1][j - 1] && (s[i - 1] == p[j - 1] || '.' ==
p[j - 1]);
                else
                    // p[0] cannot be '*' so no need to check "j > 1" here
                    f[i][j] = f[i][j - 2] || (s[i - 1] == p[j - 2] || '.' == p[j
- 2]) && f[i - 1][j];

        return f[m][n];
    }
};
```

written by xiaohui7 original link here

Answer 2

1.'.' is easy to handle. if p has a '.', it can pass any single character in s except '\0'.

2.''*is a totally different problem. if p has a* ' character, it can pass any length of first-match characters in s including '\0'.

```
  class Solution {
    public:
    bool matchFirst(const char *s, const char *p){
        return (*p == *s || (*p == '.' && *s != '\0'));
    }

bool isMatch(const char *s, const char *p) {
    if (*p == '\0') return *s == '\0';  //empty

    if (*(p + 1) != '*') {//without *
        if(!matchFirst(s,p)) return false;
        return isMatch(s + 1, p + 1);
    } else { //next: with a *
        if(isMatch(s, p + 2)) return true;    //try the length of 0
        while ( matchFirst(s,p) )       //try all possible lengths
            if (isMatch(++s, p + 2))return true;
    }
}
};
```

written by enriquewang original link here

Answer 3

In the given examples, the last one `isMatch("aab", "c*a*b") â†’ true`; don't understand why these two strings matches? Can someone please help me understand this example?

written by shawnForsythe original link here

Container With Most Water(11)

Answer 1

The O(n) solution with proof by contradiction doesn't look intuitive enough to me. Before moving on, read the algorithm first if you don't know it yet.

Here's another way to see what happens in a matrix representation:

Draw a matrix where the row is the first line, and the column is the second line. For example, say `n=6`.

In the figures below, `x` means we don't need to compute the volume for that case: (1) On the diagonal, the two lines are overlapped; (2) The lower left triangle area of the matrix is symmetric to the upper right area.

We start by computing the volume at `(1,6)`, denoted by `o`. Now if the left line is shorter than the right line, then all the elements left to `(1,6)` on the first row have smaller volume, so we don't need to compute those cases (crossed by `---`).

```
  1 2 3 4 5 6
1 x ------- o
2 x x
3 x x x
4 x x x x
5 x x x x x
6 x x x x x x
```

Next we move the left line and compute `(2,6)`. Now if the right line is shorter, all cases below `(2,6)` are eliminated.

```
  1 2 3 4 5 6
1 x ------- o
2 x x       o
3 x x x     |
4 x x x x   |
5 x x x x x |
6 x x x x x x
```

And no matter how this `o` path goes, we end up only need to find the max value on this path, which contains `n-1` cases.

```
  1 2 3 4 5 6
1 x ------- o
2 x x - o o o
3 x x x o | |
4 x x x x | |
5 x x x x x |
6 x x x x x x
```

Hope this helps. I feel more comfortable seeing things this way.

written by kongweihan original link here

Answer 2

Start by evaluating the widest container, using the first and the last line. All other possible containers are less wide, so to hold more water, they need to be higher. Thus, after evaluating that widest container, skip lines at both ends that don't support a higher height. Then evaluate that new container we arrived at. Repeat until there are no more possible containers left.

C++

```
int maxArea(vector<int>& height) {
    int water = 0;
    int i = 0, j = height.size() - 1;
    while (i < j) {
        int h = min(height[i], height[j]);
        water = max(water, (j - i) * h);
        while (height[i] <= h && i < j) i++;
        while (height[j] <= h && i < j) j--;
    }
    return water;
}
```

C

A bit shorter and perhaps faster because I can use raw int pointers, but a bit longer because I don't have `min` and `max`.

```
int maxArea(int* heights, int n) {
    int water = 0, *i = heights, *j = i + n - 1;
    while (i < j) {
        int h = *i < *j ? *i : *j;
        int w = (j - i) * h;
        if (w > water) water = w;
        while (*i <= h && i < j) i++;
        while (*j <= h && i < j) j--;
    }
    return water;
}
```

written by StefanPochmann original link here

Answer 3

The idea is : to compute area, we need to take min(height[i],height[j]) as our height. Thus if `height[i]<height[j]`, then the expression `min(height[i],height[j])` will always lead to at maximum `height[i]` for all other j(i being fixed), hence no point checking them. Similarly when `height[i]>height[j]` then all the other i's can be ignored for that particular j.

```
class Solution {
public:
    int maxArea(vector<int> &height)
    {
        int j=height.size()-1,i=0,mx=0;

        while(i<j)
        {
            mx=max(mx,((j-i)*(min(height[i],height[j]))));

            if(height[i]<height[j])
             i++;
             else if(height[i]>=height[j])
             j--;
        }
        return mx;
    }
};
```

written by franticguy original link here

Integer to Roman(12)

Answer 1

```
    public static String intToRoman(int num) {
        String M[] = {"", "M", "MM", "MMM"};
        String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"}
;
        String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"}
;
        String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"}
;
        return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];
    }
```

written by fabrizio3 original link here

Answer 2

Reference: http://blog.csdn.net/beiyeqingteng/article/details/8547565

public class Solution { public String intToRoman(int num) {

```
    int[] values = {1000,900,500,400,100,90,50,40,10,9,5,4,1};
    String[] strs = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};

    StringBuilder sb = new StringBuilder();

    for(int i=0;i<values.length;i++) {
        while(num >= values[i]) {
            num -= values[i];
            sb.append(strs[i]);
        }
    }
    return sb.toString();
}
```

}

written by Lucifer27 original link here

Answer 3

String[] romanPieces={"","I","II","III","IV","V","VI","VII","VIII","IX",
"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC",
"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM",
"","M","MM","MMM","MMMM"}; return
romanPieces[num/1000+30]+romanPieces[(num/100)%10+20]
+romanPieces[(num/10)%10+10]+romanPieces[num%10];

written by bing10 original link here

Roman to Integer(13)

Answer 1

```
public class Solution {
    public int romanToInt(String s) {
        //ï¼šâ… ï¼ˆ1ï¼‰â…¤ï¼ˆ5ï¼‰â…©ï¼ˆ10ï¼‰Lï¼ˆ50ï¼‰Cï¼ˆ100ï¼‰Dï¼ˆ500ï¼‰Mï¼ˆ1000ï¼‰
        // rules:ä½ºŽå¤§æ•°çš„åŽé¢–¶å°±ä½¸ºåŠ æ•°ï¼›ä½ºŽå¤§æ•°çš„å‰é¢å°±ä¸ºæ¸ºå‡æ•°
        //egï¼šâ…¢=3,â…£=4,â…¥=6,â…©â…¨=19,â…©â…©=20,â…©Lâ…¤=45,MCMâ…©â…©C=1980
        //"DCXXI"

        if(s == null || s.length() == 0) return 0;
        int len = s.length();
        HashMap<Character,Integer> map = new HashMap<Character,Integer>();
        map.put('I',1);
        map.put('V',5);
        map.put('X',10);
        map.put('L',50);
        map.put('C',100);
        map.put('D',500);
        map.put('M',1000);
        int result = map.get(s.charAt(len -1));
        int pivot = result;
        for(int i = len -2; i>= 0;i--){
            int curr = map.get(s.charAt(i));
            if(curr >=  pivot){
                result += curr;
            }else{
                result -= curr;
            }
            pivot = curr;
        }
        return result;
    }
}
```

written by yidi original link here

Answer 2

count every Symbol and add its value to the sum, and minus the extra part of special cases.

```
public int romanToInt(String s) {
     int sum=0;
    if(s.indexOf("IV")!=-1){sum-=2;}
    if(s.indexOf("IX")!=-1){sum-=2;}
    if(s.indexOf("XL")!=-1){sum-=20;}
    if(s.indexOf("XC")!=-1){sum-=20;}
    if(s.indexOf("CD")!=-1){sum-=200;}
    if(s.indexOf("CM")!=-1){sum-=200;}

    char c[]=s.toCharArray();
    int count=0;

   for(;count<=s.length()-1;count++){
       if(c[count]=='M') sum+=1000;
       if(c[count]=='D') sum+=500;
       if(c[count]=='C') sum+=100;
       if(c[count]=='L') sum+=50;
       if(c[count]=='X') sum+=10;
       if(c[count]=='V') sum+=5;
       if(c[count]=='I') sum+=1;

   }

   return sum;

}
```

written by hongbin2 original link here

Answer 3

Problem is simpler to solve by working the string from back to front and using a map. Runtime speed is 88 ms.

```
int romanToInt(string s)
{
    unordered_map<char, int> T = { { 'I' , 1 },
                                   { 'V' , 5 },
                                   { 'X' , 10 },
                                   { 'L' , 50 },
                                   { 'C' , 100 },
                                   { 'D' , 500 },
                                   { 'M' , 1000 } };

   int sum = T[s.back()];
   for (int i = s.length() - 2; i >= 0; --i)
   {
       if (T[s[i]] < T[s[i + 1]])
       {
           sum -= T[s[i]];
       }
       else
       {
           sum += T[s[i]];
       }
   }

   return sum;
}
```

written by wsugrad77 original link here

Longest Common Prefix(14)

Answer 1

```
public String longestCommonPrefix(String[] strs) {
    if(strs == null || strs.length == 0)    return "";
    String pre = strs[0];
    int i = 1;
    while(i < strs.length){
        while(strs[i].indexOf(pre) != 0)
            pre = pre.substring(0,pre.length()-1);
        i++;
    }
    return pre;
}
```

written by desmile original link here

Answer 2

```
public class Solution {
    public String longestCommonPrefix(List<String> strs) {
        if(strs.size()==0) return "";
        StringBuilder lcp=new StringBuilder();
        for(int i=0;i<strs.get(0).length();i++){
            char c=strs.get(0).charAt(i);
            for(String s:strs){
                if(s.length()<i+1||c!=s.charAt(i)) return lcp.toString();
            }
            lcp.append(c);
        }
        return lcp.toString();
    }
}
```

written by pengfeifc original link here

Answer 3

Sort the array first, and then you can simply compare the first and last elements in the sorted array.

```java
    public String longestCommonPrefix(String[] strs) {
        StringBuilder result = new StringBuilder();

        if (strs!= null && strs.length > 0){

            Arrays.sort(strs);

            char [] a = strs[0].toCharArray();
            char [] b = strs[strs.length-1].toCharArray();

            for (int i = 0; i < a.length; i ++){
                if (b.length > i && b[i] == a[i]){
                    result.append(b[i]);
                }
                else {
                    return result.toString();
                }
            }
        return result.toString();
    }
```

written by cassandra9 original link here

3Sum(15)

Answer 1

Hi guys!

The idea is to sort an input array and then run through all indices of a possible first element of a triplet. For each possible first element we make a standard bi-directional 2Sum sweep of the remaining part of the array. Also we want to skip equal elements to avoid duplicates in the answer without making a set or smth like that.

```
public List<List<Integer>> threeSum(int[] num) {
    Arrays.sort(num);
    List<List<Integer>> res = new LinkedList<>();
    for (int i = 0; i < num.length-2; i++) {
        if (i == 0 || (i > 0 && num[i] != num[i-1])) {
            int lo = i+1, hi = num.length-1, sum = 0 - num[i];
            while (lo < hi) {
                if (num[lo] + num[hi] == sum) {
                    res.add(Arrays.asList(num[i], num[lo], num[hi]));
                    while (lo < hi && num[lo] == num[lo+1]) lo++;
                    while (lo < hi && num[hi] == num[hi-1]) hi--;
                    lo++; hi--;
                } else if (num[lo] + num[hi] < sum) lo++;
                else hi--;
           }
        }
    }
    return res;
}
```

Have a nice coding!

written by shpolsky original link here

Answer 2

the key idea is the same as the `TwoSum` problem. When we fix the `1st` number, the `2nd` and `3rd` number can be found following the same reasoning as `TwoSum`.

The only difference is that, the `TwoSum` problem of LEETCODE has a unique solution. However, in `ThreeSum`, we have multiple duplicate solutions that can be found. Most of the OLE errors happened here because you could've ended up with a solution with so many duplicates.

The naive solution for the duplicates will be using the STL methods like below :

```
std::sort(res.begin(), res.end());
res.erase(unique(res.begin(), res.end()), res.end());
```

But according to my submissions, this way will cause you double your time consuming almostly.

A better approach is that, to jump over the number which has been scanned, no matter it is part of some solution or not.

If the three numbers formed a solution, we can safely ignore all the duplicates of them.

We can do this to all the three numbers such that we can remove the duplicates.

Here's my AC C++ Code:

```
vector<vector<int> > threeSum(vector<int> &num) {

    vector<vector<int> > res;

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

    for (int i = 0; i < num.size(); i++) {

        int target = -num[i];
        int front = i + 1;
        int back = num.size() - 1;

        while (front < back) {

            int sum = num[front] + num[back];

            // Finding answer which start from number num[i]
            if (sum < target)
                front++;

            else if (sum > target)
                back--;

            else {
                vector<int> triplet(3, 0);
                triplet[0] = num[i];
                triplet[1] = num[front];
                triplet[2] = num[back];
                res.push_back(triplet);

                // Processing duplicates of Number 2
                // Rolling the front pointer to the next different number forward
s
                while (front < back && num[front] == triplet[1]) front++;

                // Processing duplicates of Number 3
                // Rolling the back pointer to the next different number backward
s
                while (front < back && num[back] == triplet[2]) rear--;
            }

        }

        // Processing duplicates of Number 1
        while (i + 1 < num.size() && num[i + 1] == num[i])
            i++;

    }

    return res;

}
```

written by kun3 original link here

Answer 3

Sort the array, iterate through the list, and use another two pointers to approach the target. Runtime: `7ms`

```
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if(nums == null || nums.length < 3) return result;
    Arrays.sort(nums);

    int len = nums.length;
    for(int i = 0; i < len; i++) {
        if(i > 0 && nums[i] == nums[i - 1]) continue;        // Skip same results
        int target = 0 - nums[i];
        int j = i + 1, k = len - 1;
        while(j < k) {
            if(nums[j] + nums[k] == target) {
                result.add(Arrays.asList(nums[i], nums[j], nums[k]));
                while(j < k && nums[j] == nums[j + 1]) j++;  // Skip same results
                while(j < k && nums[k] == nums[k - 1]) k--;  // Skip same results
                j++; k--;
            } else if(nums[j] + nums[k] < target) {
                j++;
            } else {
                k--;
            }
        }
    }
    return result;
}
```

written by yavinci original link here

3Sum Closest(16)

Answer 1

Here is a solution in Order(N^2). I got help from this post on stackoverflow
Can we improve this time complexity ?

```
int threeSumClosest(vector<int> &num, int target) {
    vector<int> v(num.begin(), num.end()); // I didn't wanted to disturb original
array.
    int n = 0;
    int ans = 0;
    int sum;

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

    // If less then 3 elements then return their sum
    while (v.size() <= 3) {
        return accumulate(v.begin(), v.end(), 0);
    }

    n = v.size();

    /* v[0] v[1] v[2] ... v[i] .... v[j] ... v[k] ... v[n-2] v[n-1]
     *                    v[i]  <=  v[j]  <= v[k] always, because we sorted our a
rray.
     * Now, for each number, v[i] : we look for pairs v[j] & v[k] such that
     * absolute value of (target - (v[i] + v[j] + v[k]) is minimised.
     * if the sum of the triplet is greater then the target it implies
     * we need to reduce our sum, so we do K = K - 1, that is we reduce
     * our sum by taking a smaller number.
     * Simillarly if sum of the triplet is less then the target then we
     * increase out sum by taking a larger number, i.e. J = J + 1.
     */
    ans = v[0] + v[1] + v[2];
    for (int i = 0; i < n-2; i++) {
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            sum = v[i] + v[j] + v[k];
            if (abs(target - ans) > abs(target - sum)) {
                ans = sum;
                if (ans == target) return ans;
            }
            (sum > target) ? k-- : j++;
        }
    }
    return ans;
}
```

Edit: Thanks @thr for pointing out that. I have corrected it and also renamed 'mx' by 'ans'.

written by vaibhavatul47 original link here

Answer 2

Sort the vector and then no need to run*O(N^3)* algorithm as each index has a direction to move.

The code starts from this formation.

```
-----------------------------------------------------
^  ^                                                 ^
|  |                                                 |
|  +- second                                       third
+-first
```

if *nums[first] + nums[second] + nums[third]* is smaller than the *target*, we know we have to increase the sum. so only choice is moving the second index forward.

```
-----------------------------------------------------
^    ^                                               ^
|    |                                               |
|    +- second                                     third
+-first
```

if the *sum* is bigger than the *target*, we know that we need to reduce the *sum*. so only choice is moving '*third*' to backward. of course if the *sum* equals to *target*, we can immediately return the *sum*.

```
-----------------------------------------------------
^    ^                                             ^
|    |                                             |
|    +- second                                   third
+-first
```

when *second* and *third* cross, the round is done so start next round by moving '*first*' and resetting *second* and *third*.

```
-----------------------------------------------------
  ^    ^                                           ^
  |    |                                           |
  |    +- second                                 third
  +-first
```

while doing this, collect the *closest sum* of each stage by calculating and comparing delta. Compare *abs(target-newSum)* and *abs(target-closest)*. At the end of the process the three indexes will eventually be gathered at the end of the array.

```
-----------------------------------------------------
                                          ^    ^    ^
                                          |    |    `- third
                                          |    +- second
                                          +-first
```

if no exactly matching *sum* has been found so far, the value in*closest* will be the answer.

```
int threeSumClosest(vector<int>& nums, int target) {
    if(nums.size() < 3) return 0;
    int closest = nums[0]+nums[1]+nums[2];
    sort(nums.begin(), nums.end());
    for(int first = 0 ; first < nums.size()-2 ; ++first) {
        if(first > 0 && nums[first] == nums[first-1]) continue;
        int second = first+1;
        int third = nums.size()-1;
        while(second < third) {
            int curSum = nums[first]+nums[second]+nums[third];
            if(curSum == target) return curSum;
            if(abs(target-curSum)<abs(target-closest)) {
                closest = curSum;
            }
            if(curSum > target) {
                --third;
            } else {
                ++second;
            }
        }
    }
    return closest;
}
```

written by asbear original link here

Answer 3

Similar to 3 Sum problem, use 3 pointers to point current element, next element and the last element. If the sum is less than target, it means we have to add a larger element so next element move to the next. If the sum is greater, it means we have to add a smaller element so last element move to the second last element. Keep doing this until the end. Each time compare the difference between sum and target, if it is less than minimum difference so far, then replace result with it, otherwise keep iterating.

```java
public class Solution {
    public int threeSumClosest(int[] num, int target) {
        int result = num[0] + num[1] + num[num.length - 1];
        Arrays.sort(num);
        for (int i = 0; i < num.length - 2; i++) {
            int start = i + 1, end = num.length - 1;
            while (start < end) {
                int sum = num[i] + num[start] + num[end];
                if (sum > target) {
                    end--;
                } else {
                    start++;
                }
                if (Math.abs(sum - target) < Math.abs(result - target)) {
                    result = sum;
                }
            }
        }
        return result;
    }
}
```

written by chase1991 original link here

Letter Combinations of a Phone Number(17)

Answer 1

```
    public List<String> letterCombinations(String digits) {
    LinkedList<String> ans = new LinkedList<String>();
    String[] mapping = new String[] {"0", "1", "abc", "def", "ghi", "jkl", "mno",
"pqrs", "tuv", "wxyz"};
    ans.add("");
    for(int i =0; i<digits.length();i++){
        int x = Character.getNumericValue(digits.charAt(i));
        while(ans.peek().length()==i){
            String t = ans.remove();
            for(char s : mapping[x].toCharArray())
                ans.add(t+s);
        }
    }
    return ans;
}
```

written by lirensun original link here

Answer 2

This is my solution, FYI

```
vector<string> letterCombinations(string digits) {
    vector<string> res;
    string charmap[10] = {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "t
uv", "wxyz"};
    res.push_back("");
    for (int i = 0; i < digits.size(); i++)
    {
        vector<string> tempres;
        string chars = charmap[digits[i] - '0'];
        for (int c = 0; c < chars.size();c++)
            for (int j = 0; j < res.size();j++)
                tempres.push_back(res[j]+chars[c]);
        res = tempres;
    }
    return res;
}
```

written by peerlessbloom original link here

Answer 3

```
vector<string> letterCombinations(string digits) {
    vector<string> result;
    if(digits.empty()) return vector<string>();
    static const vector<string> v = {"", "", "abc", "def", "ghi", "jkl", "mno", "
pqrs", "tuv", "wxyz"};
    result.push_back("");   // add a seed for the initial case
    for(int i = 0 ; i < digits.size(); ++i) {
        int num = digits[i]-'0';
        if(num < 0 || num > 9) break;
        const string& candidate = v[num];
        if(candidate.empty()) continue;
        vector<string> tmp;
        for(int j = 0 ; j < candidate.size() ; ++j) {
            for(int k = 0 ; k < result.size() ; ++k) {
                tmp.push_back(result[k] + candidate[j]);
            }
        }
        result.swap(tmp);
    }
    return result;
}
```

Simple and efficient iterative solution.

Explanation with sample input "123"

Initial state:

- result = {""}

Stage 1 for number "1":

- result has {""}
- candiate is "abc"
- generate three strings "" + "a", ""+"b", ""+"c" and put into tmp, tmp = {"a", "b","c"}
- swap result and tmp (swap does not take memory copy)
- Now result has {"a", "b", "c"}

Stage 2 for number "2":

- result has {"a", "b", "c"}
- candidate is "def"
- generate nine strings and put into tmp, "a" + "d", "a"+"e", "a"+"f", "b" + "d", "b"+"e", "b"+"f", "c" + "d", "c"+"e", "c"+"f"
- so tmp has {"ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf" }
- swap result and tmp
- Now result has {"ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf" }

Stage 3 for number "3":

- result has {"ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf" }
- candidate is "ghi"
- generate 27 strings and put into tmp,

- add "g" for each of "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"
- add "h" for each of "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"
- add "h" for each of "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"
- so, tmp has {"adg", "aeg", "afg", "bdg", "beg", "bfg", "cdg", "ceg", "cfg" "adh", "aeh", "afh", "bdh", "beh", "bfh", "cdh", "ceh", "cfh" "adi", "aei", "afi", "bdi", "bei", "bfi", "cdi", "cei", "cfi" }
- swap result and tmp
- Now result has {"adg", "aeg", "afg", "bdg", "beg", "bfg", "cdg", "ceg", "cfg" "adh", "aeh", "afh", "bdh", "beh", "bfh", "cdh", "ceh", "cfh" "adi", "aei", "afi", "bdi", "bei", "bfi", "cdi", "cei", "cfi" }

Finally, return result.

written by asbear original link here

4Sum(18)

Answer 1

For the reference, please have a look at my explanation of `3Sum` problem because the algorithm are exactly the same. The link is as blow.

My 3Sum problem answer

The key idea is to downgrade the problem to a `2Sum` problem eventually. And the same algorithm can be expand to `NSum` problem.

After you had a look at my explanation of `3Sum`, the code below will be extremely easy to understand.

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int target) {

        vector<vector<int> > res;

        if (num.empty())
            return res;

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

        for (int i = 0; i < num.size(); i++) {

            int target_3 = target - num[i];

            for (int j = i + 1; j < num.size(); j++) {

                int target_2 = target_3 - num[j];

                int front = j + 1;
                int back = num.size() - 1;

                while(front < back) {

                    int two_sum = num[front] + num[back];

                    if (two_sum < target_2) front++;

                    else if (two_sum > target_2) back--;

                    else {

                        vector<int> quadruplet(4, 0);
                        quadruplet[0] = num[i];
                        quadruplet[1] = num[j];
                        quadruplet[2] = num[front];
                        quadruplet[3] = num[back];
                        res.push_back(quadruplet);

                        // Processing the duplicates of number 3
                        while (front < back && num[front] == quadruplet[2]) ++fro
```

```
nt;

                        // Processing the duplicates of number 4
                        while (front < back && num[back] == quadruplet[3]) --back
;

                    }
                }

                // Processing the duplicates of number 2
                while(j + 1 < num.size() && num[j + 1] == num[j]) ++j;
            }

            // Processing the duplicates of number 1
            while (i + 1 < num.size() && num[i + 1] == num[i]) ++i;

        }

        return res;

    }
};
```

written by kun3 original link here

Answer 2

My idea is to sort num first, then build a hashtable with the key as the sum of the pair and the value as a vector storing all pairs of index of num that having the same sum. In this way, all elements stored in hashtable has a order that duplicate pairs are neighbors. Therefore scanning the vector in the hashtable we only put non duplicate elements into the final answer vvi.

Is this method O(n^2) ? or Does anyone can improve it to O(n^2);

```
    class Solution{ //using hashtable, avg O(n^2)

public:

    vector<vector<int> > fourSum(vector<int> &num, int target){
        vector<vector<int> > vvi;
        int n = num.size();
        if(n < 4) return  vvi;

        sort(num.begin(), num.end());
        unordered_map<int, vector<pair<int, int>> > mp;
        for(int i = 0; i < n; i++){
            for(int j = i + 1; j < n; j++){
                mp[num[i]+num[j]].push_back(make_pair(i,j));
            }
        }

        for(int i = 0; i < n; i++){
            if(i>0 && num[i] == num[i-1]) continue;
            for(int j = i + 1; j < n; j++){
                if(j > i + 1 && num[j] == num[j-1]) continue;
                int res = target - num[i] - num[j];
                if(mp.count(res)){
                    for(auto it = mp[res].begin(); it != mp[res].end(); it++){
                        int k = (*it).first, l = (*it).second;
                        if(k > j){ // k>j make sure that the second pair has bigg
er values than the first pair.
                            if(!vvi.empty() && num[i]==vvi.back()[0] && num[j]==v
vi.back()[1]
                            && num[k]==vvi.back()[2] && num[l] == vvi.back()[3]){
                                continue; //if the obtained 4 elements are the sa
me as previous one continue to next
                            }
                            vector<int> vi={num[i], num[j], num[k], num[l]};
                            vvi.push_back(vi);
                        } // if k>j

                    }//for it
                }//if
            }// forj
        }//for i
        return vvi;
    }
  };
```

written by asafeather original link here

Answer 3

```
public class Solution {
    public List<List<Integer>> fourSum(int[] num, int target) {
        ArrayList<List<Integer>> ans = new ArrayList<>();
        if(num.length<4)return ans;
        Arrays.sort(num);
        for(int i=0; i<num.length-3; i++){
            if(i>0&&num[i]==num[i-1])continue;
            for(int j=i+1; j<num.length-2; j++){
                if(j>i+1&&num[j]==num[j-1])continue;
                int low=j+1, high=num.length-1;
                while(low<high){
                    int sum=num[i]+num[j]+num[low]+num[high];
                    if(sum==target){
                        ans.add(Arrays.asList(num[i], num[j], num[low], num[high]
));
                        while(low<high&&num[low]==num[low+1])low++;
                        while(low<high&&num[high]==num[high-1])high--;
                        low++;
                        high--;
                    }
                    else if(sum<target)low++;
                    else high--;
                }
            }
        }
        return ans;
    }
}
```

written by casualhero original link here

Remove Nth Node From End of List(19)

Answer 1

A one pass solution can be done using pointers. Move one pointer **fast** --> **n+1** places forward, to maintain a gap of n between the two pointers and then move both at the same speed. Finally, when the fast pointer reaches the end, the slow pointer will be **n+1** places behind - just the right spot for it to be able to skip the next node.

Since the question gives that **n** is valid, not too many checks have to be put in place. Otherwise, this would be necessary.

```
   public ListNode removeNthFromEnd(ListNode head, int n) {

    ListNode start = new ListNode(0);
    ListNode slow = start, fast = start;
    slow.next = head;

    //Move fast in front so that the gap between slow and fast becomes n
    for(int i=1; i<=n+1; i++)   {
        fast = fast.next;
    }
    //Move fast to the end, maintaining the gap
    while(fast != null) {
        slow = slow.next;
        fast = fast.next;
    }
    //Skip the desired node
    slow.next = slow.next.next;
    return start.next;
}
```

written by TMS original link here

Answer 2

```
class Solution
{
public:
    ListNode* removeNthFromEnd(ListNode* head, int n)
    {
        ListNode** t1 = &head, *t2 = head;
        for(int i = 1; i < n; ++i)
        {
            t2 = t2->next;
        }
        while(t2->next != NULL)
        {
            t1 = &((*t1)->next);
            t2 = t2->next;
        }
        *t1 = (*t1)->next;
        return head;
    }
};
```

written by taotaou original link here

Answer 3

```
public ListNode RemoveNthFromEnd(ListNode head, int n) {
    ListNode h1=head, h2=head;
    while(n-->0) h2=h2.next;
    if(h2==null)return head.next;  // The head need to be removed, do it.
    h2=h2.next;

    while(h2!=null){
        h1=h1.next;
        h2=h2.next;
    }
    h1.next=h1.next.next;   // the one after the h1 need to be removed
    return head;
}
```

written by xiangyucao original link here

Valid Parentheses(20)

Answer 1

```
public class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<Character>();
        // Iterate through string until empty
        for(int i = 0; i<s.length(); i++) {
            // Push any open parentheses onto stack
            if(s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{')
                stack.push(s.charAt(i));
            // Check stack for corresponding closing parentheses, false if not va
lid
            else if(s.charAt(i) == ')' && !stack.empty() && stack.peek() == '(')
                stack.pop();
            else if(s.charAt(i) == ']' && !stack.empty() && stack.peek() == '[')
                stack.pop();
            else if(s.charAt(i) == '}' && !stack.empty() && stack.peek() == '{')
                stack.pop();
            else
                return false;
        }
        // return true if no open parentheses left in stack
        return stack.empty();
    }
}
```

written by KyleAsaff original link here

Answer 2

```
class Solution:
    # @return a boolean
    def isValid(self, s):
        stack = []
        dict = {"]":"[", "}":"{", ")":"("}
        for char in s:
            if char in dict.values():
                stack.append(char)
            elif char in dict.keys():
                if stack == [] or dict[char] != stack.pop():
                    return False
            else:
                return False
        return stack == []
```

It's quite obvious.

written by xiaoying10101 original link here

Answer 3

```java
public class Solution {
    public boolean isValid(String s) {
        Stack<Integer> p = new Stack<>();
        for(int i = 0; i < s.length(); i++) {
            int q = "(){}[]".indexOf(s.substring(i, i + 1));
            if(q % 2 == 1) {
                if(p.isEmpty() || p.pop() != q - 1) return false;
            } else p.push(q);
        }
        return p.isEmpty();
    }
}
```

written by wchiang original link here

Merge Two Sorted Lists(21)

Answer 1

```
class Solution {
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        if(l1 == NULL) return l2;
        if(l2 == NULL) return l1;

        if(l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l2->next, l1);
            return l2;
        }
    }
};
```

This solution is not a tail-recursive, the stack will overflow while the list is too long :) http://en.wikipedia.org/wiki/Tail_call

written by GZShi original link here

Answer 2

```
class Solution {
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode dummy(INT_MIN);
        ListNode *tail = &dummy;

        while (l1 && l2) {
            if (l1->val < l2->val) {
                tail->next = l1;
                l1 = l1->next;
            } else {
                tail->next = l2;
                l2 = l2->next;
            }
            tail = tail->next;
        }

        tail->next = l1 ? l1 : l2;
        return dummy.next;
    }
};
```

written by xiaohui7 original link here

Answer 3

Hello every one, here is my code, simple but works well:

```java
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null){
            return l2;
        }
        if(l2 == null){
            return l1;
        }

        ListNode mergeHead;
        if(l1.val < l2.val){
            mergeHead = l1;
            mergeHead.next = mergeTwoLists(l1.next, l2);
        }
        else{
            mergeHead = l2;
            mergeHead.next = mergeTwoLists(l1, l2.next);
        }
        return mergeHead;
    }
}
```

written by zwangbo original link here

Generate Parentheses(22)

Answer 1

The idea is intuitive. Use two integers to count the remaining left parenthesis (n) and the right parenthesis (m) to be added. At each function call add a left parenthesis if n >0 and add a right parenthesis if m>0. Append the result and terminate recursive calls when both m and n are zero.

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        addingpar(res, "", n, 0);
        return res;
    }
    void addingpar(vector<string> &v, string str, int n, int m){
        if(n==0 && m==0) {
            v.push_back(str);
            return;
        }
        if(m > 0){ addingpar(v, str+")", n, m-1); }
        if(n > 0){ addingpar(v, str+"(", n-1, m+1); }
    }
};
```

written by klycok original link here

Answer 2

My method is DP. First consider how to get the result f(n) from previous result f(0)...f(n-1). Actually, the result f(n) will be put an extra () pair to f(n-1). Let the "(" always at the first position, to produce a valid result, we can only put ")" in a way that there will be i pairs () inside the extra () and n - 1 - i pairs () outside the extra pair.

Let us consider an example to get clear view:

f(0): ""

f(1): "("f(0)")"

f(2): "("f(0)")"f(1), "("f(1)")"

f(3): "("f(0)")"f(2), "("f(1)")"f(1), "("f(2)")"

So f(n) = "("f(0)")"f(n-1) , "("f(1)")"f(n-2) "("f(2)")"f(n-3) ... "("f(i)")"f(n-1-i) ... "(f(n-1)")"

Below is my code:

```
public class Solution
{
    public List<String> generateParenthesis(int n)
    {
        List<List<String>> lists = new ArrayList<>();
        lists.add(Collections.singletonList(""));

        for (int i = 1; i <= n; ++i)
        {
            final List<String> list = new ArrayList<>();

            for (int j = 0; j < i; ++j)
            {
                for (final String first : lists.get(j))
                {
                    for (final String second : lists.get(i - 1 - j))
                    {
                        list.add("(" + first + ")" + second);
                    }
                }
            }

            lists.add(list);
        }

        return lists.get(lists.size() - 1);
    }
}
```

written by left.peter original link here

Answer 3

```
 public List<String> generateParenthesis(int n) {
        List<String> list = new ArrayList<String>();
        backtrack(list, "", 0, 0, n);
        return list;
    }

    public void backtrack(List<String> list, String str, int open, int close, int
max){

        if(str.length() == max*2){
            list.add(str);
            return;
        }

        if(open < max)
            backtrack(list, str+"(", open+1, close, max);
        if(close < open)
            backtrack(list, str+")", open, close+1, max);
    }
```

The idea here is to only add '(' and ')' that we know will guarantee us a solution

(instead of adding 1 too many close). Once we add a '(' we will then discard it and try a ')' which can only close a valid '('. Each of these steps are recursively called.

written by brobins9 original link here

Merge k Sorted Lists(23)

Answer 1

If someone understand how priority queue works, then it would be trivial to walk through the codes.

My question: is that possible to solve this question under the same time complexity without implementing the priority queue?

```
public class Solution {
    public ListNode mergeKLists(List<ListNode> lists) {
        if (lists==null||lists.size()==0) return null;

        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.size(),n
ew Comparator<ListNode>(){
            @Override
            public int compare(ListNode o1,ListNode o2){
                if (o1.val<o2.val)
                    return -1;
                else if (o1.val==o2.val)
                    return 0;
                else
                    return 1;
            }
        });

        ListNode dummy = new ListNode(0);
        ListNode tail=dummy;

        for (ListNode node:lists)
            if (node!=null)
                queue.add(node);

        while (!queue.isEmpty()){
            tail.next=queue.poll();
            tail=tail.next;

            if (tail.next!=null)
                queue.add(tail.next);
        }
        return dummy.next;
    }
}
```

written by reeclapple original link here

Answer 2

```
ListNode *mergeKLists(vector<ListNode *> &lists) {
    if(lists.empty()){
        return nullptr;
    }
    while(lists.size() > 1){
        lists.push_back(mergeTwoLists(lists[0], lists[1]));
        lists.erase(lists.begin());
        lists.erase(lists.begin());
    }
    return lists.front();
}
ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
    if(l1 == nullptr){
        return l2;
    }
    if(l2 == nullptr){
        return l1;
    }
    if(l1->val <= l2->val){
        l1->next = mergeTwoLists(l1->next, l2);
        return l1;
    }
    else{
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
}
```

The second function is from Merge Two Sorted Lists.

The basic idea is really simple. We can merge first two lists and then push it back. Keep doing this until there is only one list left in vector. Actually, we can regard this as an iterative divide-and-conquer solution.

written by zxyperfect original link here

Answer 3

```
public static ListNode mergeKLists(ListNode[] lists){
    return partion(lists,0,lists.length-1);
}

public static ListNode partion(ListNode[] lists,int s,int e){
    if(s==e)  return lists[s];
    if(s<e){
        int q=(s+e)/2;
        ListNode l1=partion(lists,s,q);
        ListNode l2=partion(lists,q+1,e);
        return merge(l1,l2);
    }else
        return null;
}

//This function is from Merge Two Sorted Lists.
public static ListNode merge(ListNode l1,ListNode l2){
    if(l1==null) return l2;
    if(l2==null) return l1;
    if(l1.val<l2.val){
        l1.next=merge(l1.next,l2);
        return l1;
    }else{
        l2.next=merge(l1,l2.next);
        return l2;
    }
}
```

written by mouqi123 original link here

Swap Nodes in Pairs(24)

Answer 1

```
public class Solution {
    public ListNode swapPairs(ListNode head) {
        if ((head == null)||(head.next == null))
            return head;
        ListNode n = head.next;
        head.next = swapPairs(head.next.next);
        n.next = head;
        return n;
    }
}
```

written by whoji original link here

Answer 2

Three different implementations of the same algorithm, taking advantage of different strengths of the three languages. I suggest reading all three, even if you don't know all three languages.

All three of course work swap the current node with the next node by rearranging pointers, then move on to the next pair, and repeat until the end of the list.

C++

Pointer-pointer `pp` points to the pointer to the current node. So at first, `pp` points to `head`, and later it points to the `next` field of ListNodes. Additionally, for convenience and clarity, pointers `a` and `b` point to the current node and the next node.

We need to go from `*pp == a -> b -> (b->next)` to `*pp == b -> a -> (b->next)`. The first three lines inside the loop do that, setting those three pointers (from right to left). The fourth line moves `pp` to the next pair.

```
ListNode* swapPairs(ListNode* head) {
    ListNode **pp = &head, *a, *b;
    while ((a = *pp) && (b = a->next)) {
        a->next = b->next;
        b->next = a;
        *pp = b;
        pp = &(a->next);
    }
    return head;
}
```

Python

Here, `pre` is the previous node. Since the head doesn't have a previous node, I just use `self` instead. Again, `a` is the current node and `b` is the next node.

To go from `pre -> a -> b -> b.next` to `pre -> b -> a -> b.next`, we need to change those three references. Instead of thinking about in what order I change them, I just change all three at once.

```
def swapPairs(self, head):
    pre, pre.next = self, head
    while pre.next and pre.next.next:
        a = pre.next
        b = a.next
        pre.next, b.next, a.next = b, a, b.next
        pre = a
    return self.next
```

Ruby

Again, `pre` is the previous node, but here I create a dummy as previous node of the head. And again, `a` is the current node and `b` is the next node. This time I go one node further and call it `c`.

To go from `pre -> a -> b -> c` to `pre -> b -> a -> c`, we need to change those three references. Here I chain the assignments, pretty much directly saying "`pre` points to `b`, which points to `a`, which points to `c`".

```
def swap_pairs(head)
    pre = dummy = ListNode.new 0
    pre.next = head
    while a = pre.next and b = a.next
        c = b.next
        ((pre.next = b).next = a).next = c
        pre = a
    end
    dummy.next
end
```

written by StefanPochmann original link here

Answer 3

My solution is quite simple. Just find the reverse job is the same for every 2 nodes.

```
public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode newhd = head.next;
        head.next = swapPairs(newhd.next);
        newhd.next = head;
        return newhd;
}
```

written by ulyx original link here

Reverse Nodes in k-Group(25)

Answer 1

Hi, guys! Despite the fact that the approach is recursive, the code is less than 20 lines. :)

```
public ListNode reverseKGroup(ListNode head, int k) {
    ListNode curr = head;
    int count = 0;
    while (curr != null && count != k) { // find the k+1 node
        curr = curr.next;
        count++;
    }
    if (count == k) { // if k+1 node is found
        curr = reverseKGroup(curr, k); // reverse list with k+1 node as head
        // head - head-pointer to direct part,
        // curr - head-pointer to reversed part;
        while (count-- > 0) { // reverse current k-group:
            ListNode tmp = head.next; // tmp - next head in direct part
            head.next = curr; // preappending "direct" head to the reversed list
            curr = head; // move head of reversed part to a new node
            head = tmp; // move "direct" head to the next node in direct part
        }
        head = curr;
    }
    return head;
}
```

Hope it helps!

written by shpolsky original link here

Answer 2

```
-1 -> 1 -> 2 -> 3 -> 4 -> 5
 |    |    |    |
pre  cur  nex  tmp

-1 -> 2 -> 1 -> 3 -> 4 -> 5
 |         |    |    |
pre       cur  nex  tmp

-1 -> 3 -> 2 -> 1 -> 4 -> 5
 |              |    |    |
pre            cur  nex  tmp
```

Above is how it works inside one group iteration(for example, k=3)

```
class Solution {
public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        if(head==NULL||k==1) return head;
        int num=0;
        ListNode *preheader = new ListNode(-1);
        preheader->next = head;
        ListNode *cur = preheader, *nex, *tmp, *pre = preheader;
        while(cur = cur->next)
            num++;
        while(num>=k) {
            cur = pre->next;
            nex = cur->next;
            for(int i=1;i<k;i++) {
                tmp= nex->next;
                nex->next = pre->next;
                pre->next = nex;
                cur->next = tmp;
                nex = tmp;
            }
            pre = cur;
            num-=k;
        }
        return preheader->next;
    }
};
```

Thanks to ciaoliang1992, the tmp pointer is no necessary, so the more concise solution is

```
class Solution {
public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        if(head==NULL||k==1) return head;
        int num=0;
        ListNode *preheader = new ListNode(-1);
        preheader->next = head;
        ListNode *cur = preheader, *nex, *pre = preheader;
        while(cur = cur->next)
            num++;
        while(num>=k) {
            cur = pre->next;
            nex = cur->next;
            for(int i=1;i<k;++i) {
                cur->next=nex->next;
                nex->next=pre->next;
                pre->next=nex;
                nex=cur->next;
            }
            pre = cur;
            num-=k;
        }
        return preheader->next;
    }
};
```

written by luming.zhang.75 original link here

Answer 3

```java
public class Solution {
      public ListNode reverseKGroup(ListNode head, int k) {
          if (head==null||head.next==null||k<2) return head;

          ListNode dummy = new ListNode(0);
          dummy.next = head;

          ListNode tail = dummy, prev = dummy,temp;
          int count;
          while(true){
              count =k;
              while(count>0&&tail!=null){
                  count--;
                  tail=tail.next;
              }
              if (tail==null) break;//Has reached the end


              head=prev.next;//for next cycle
          // prev-->temp-->...--->....--->tail-->....
          // Delete @temp and insert to the next position of @tail
          // prev-->...-->...-->tail-->head-->...
          // Assign @temp to the next node of @prev
          // prev-->temp-->...-->tail-->...-->...
          // Keep doing until @tail is the next node of @prev
              while(prev.next!=tail){
                  temp=prev.next;//Assign
                  prev.next=temp.next;//Delete

                  temp.next=tail.next;
                  tail.next=temp;//Insert

              }

              tail=head;
              prev=head;

          }
          return dummy.next;

      }
  }
```

written by reeclapple original link here

Remove Duplicates from Sorted Array(26)

Answer 1

```
class Solution {
    public:
    int removeDuplicates(int A[], int n) {
        if(n < 2) return n;
        int id = 1;
        for(int i = 1; i < n; ++i)
            if(A[i] != A[i-1]) A[id++] = A[i];
        return id;
    }
};
```

written by liyangguang1988 original link here

Answer 2

```
int count = 0;
for(int i = 1; i < n; i++){
    if(A[i] == A[i-1]) count++;
    else A[i-count] = A[i];
}
return n-count;
```

written by jasusy original link here

Answer 3

```
int removeDuplicates(vector<int>& nums) {
    int pos = 0;

    for (int i = 0; i < nums.size(); ++i) {
        if (i == 0 || nums[i] != nums[pos - 1])
            nums[pos++] = nums[i];
    }

    return pos;
}
```

written by RichyMong original link here

Remove Element(27)

Answer 1

```
int removeElement(int A[], int n, int elem) {
    int begin=0;
    for(int i=0;i<n;i++) if(A[i]!=elem) A[begin++]=A[i];
    return begin;
}
```

written by daxianji007 original link here

Answer 2

The basic idea is when elem is found at index i, let A[i] = the last element in the modifying array, then repeat searching until elem is not found.

```
public int removeElement(int[] A, int elem) {
    int len = A.length;
    for (int i = 0 ; i< len; ++i){
        while (A[i]==elem && i< len) {
            A[i]=A[--len];
        }
    }
    return len;
}
```

written by cbmbbz original link here

Answer 3

public class Solution {

```
public int removeElement(int[] A, int elem) {
   int m = 0;
   for(int i = 0; i < A.length; i++){

       if(A[i] != elem){
           A[m] = A[i];
           m++;
       }
   }

   return m;
}
```

}

written by vy7Sun original link here

Implement strStr()(28)

Answer 1

```
int strStr(char *haystack, char *needle) {
        if (!haystack || !needle) return -1;
        for (int i = 0; ; ++i) {
            for (int j = 0; ; ++j) {
                if (needle[j] == 0) return i;
                if (haystack[i + j] == 0) return -1;
                if (haystack[i + j] != needle[j]) break;
            }
        }
    }
```

written by shichaotan original link here

Answer 2

Well, the problem does not aim for an advanced algorithm like KMP but only a clean brute-force algorithm. So we can traverse all the possible starting points of `haystack` (from `0` to `haystack.length() - needle.length()`) and see if the following characters in `haystack` match those of `needle`.

The code is as follows.

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        int m = haystack.length(), n = needle.length();
        if (!n) return 0;
        for (int i = 0; i < m - n + 1; i++) {
            int j = 0;
            for (; j < n; j++)
                if (haystack[i + j] != needle[j])
                    break;
            if (j == n) return i;
        }
        return -1;
    }
};
```

Of course, you may challenge yourself implementing the KMP algorithm for this problem.

KMP is a classic and yet notoriously hard-to-understand algorithm. However, I think the following two links give nice explanations. You may refer to them.

1. KMP on jBoxer's blog;
2. KMP on geeksforgeeks, with a well-commented C code.

I am sorry that I am still unable to give a personal explanation of the algorithm. I only read it from the two links above and mimic the code in the second link.

My accepted C++ code using KMP is as follows. Well, it also takes 4ms -_-

```
class Solution {
public:
    int strStr(string haystack, string needle) {
        int m = haystack.length(), n = needle.length();
        if (!n) return 0;
        vector<int> lps = kmpProcess(needle);
        for (int i = 0, j = 0; i < m; ) {
            if (haystack[i] == needle[j]) {
                i++;
                j++;
            }
            if (j == n) return i - j;
            if (i < m && haystack[i] != needle[j]) {
                if (j) j = lps[j - 1];
                else i++;
            }
        }
        return -1;
    }
private:
    vector<int> kmpProcess(string& needle) {
        int n = needle.length();
        vector<int> lps(n, 0);
        for (int i = 1, len = 0; i < n; ) {
            if (needle[i] == needle[len])
                lps[i++] = ++len;
            else if (len) len = lps[len - 1];
            else lps[i++] = 0;
        }
        return lps;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```java
public class Solution {
    public int strStr(String haystack, String needle) {
        int l1 = haystack.length(), l2 = needle.length();
        if (l1 < l2) {
            return -1;
        } else if (l2 == 0) {
            return 0;
        }
        int threshold = l1 - l2;
        for (int i = 0; i <= threshold; ++i) {
            if (haystack.substring(i,i+l2).equals(needle)) {
                return i;
            }
        }
        return -1;
    }
}
```

written by iziang original link here

Divide Two Integers(29)

Answer 1

In this problem, we are asked to divide two integers. However, we are not allowed to use division, multiplication and mod operations. So, what else can we use? Yeah, bit manipulations.

Let's do an example and see how bit manipulations work.

Suppose we want to divide `15` by `3`, so `15` is `dividend` and `3` is `divisor`. Well, division simply requires us to find how many times we can subtract the `divisor` from the the `dividend` without making the `dividend` negative.

Let's get started. We subtract `3` from `15` and we get `12`, which is positive. Let's try to subtract more. Well, we **shift** `3` to the left by `1` bit and we get `6`. Subtracting `6` from `15` still gives a positive result. Well, we shift again and get `12`. We subtract `12` from `15` and it is still positive. We shift again, obtaining `24` and we know we can at most subtract `12`. Well, since `12` is obtained by shifting `3` to left twice, we know it is `4` times of `3`. How do we obtain this `4`? Well, we start from `1` and shift it to left twice at the same time. We add `4` to an answer (initialized to be `0`). In fact, the above process is like `15 = 3 * 4 + 3`. We now get part of the quotient (`4`), with a remainder `3`.

Then we repeat the above process again. We subtract `divisor = 3` from the remaining `dividend = 3` and obtain `0`. We know we are done. No shift happens, so we simply add `1 << 0` to the answer.

Now we have the full algorithm to perform division.

According to the problem statement, we need to handle some exceptions, such as overflow.

Well, two cases may cause overflow:

1. `divisor = 0`;
2. `dividend = INT_MIN` and `divisor = -1` (because `abs(INT_MIN) = INT_MAX + 1`).

Of course, we also need to take the sign into considerations, which is relatively easy.

Putting all these together, we have the following code.

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (!divisor || (dividend == INT_MIN && divisor == -1))
            return INT_MAX;
        int sign = ((dividend < 0) ^ (divisor < 0)) ? -1 : 1;
        long long dvd = labs(dividend);
        long long dvs = labs(divisor);
        int res = 0;
        while (dvd >= dvs) {
            long long temp = dvs, multiple = 1;
            while (dvd >= (temp << 1)) {
                temp <<= 1;
                multiple <<= 1;
            }
            dvd -= temp;
            res += multiple;
        }
        return sign == 1 ? res : -res;
    }
};
```

written by jianchao.li.fighter original link here

Answer 2

Long division in binary: The outer loop reduces n by at least half each iteration. So It has O(log N) iterations. The inner loop has at most log N iterations. So the overall complexity is O((log N)^2)

```
typedef long long ll;

int divide(int n_, int d_) {
    ll ans=0;
    ll n=abs((ll)n_);
    ll d=abs((ll)d_);
    while(n>=d){
        ll a=d;
        ll m=1;
        while((a<<1) < n){a<<=1;m<<=1;}
        ans+=m;
        n-=a;
    }
    if((n_<0&&d_>=0)||(n_>=0&&d_<0))
        return -ans;
    return ans;
}
```

written by lucastan original link here

Answer 3

```python
class Solution:
# @return an integer
def divide(self, dividend, divisor):
    positive = (dividend < 0) is (divisor < 0)
    dividend, divisor = abs(dividend), abs(divisor)
    res = 0
    while dividend >= divisor:
        temp, i = divisor, 1
        while dividend >= temp:
            dividend -= temp
            res += i
            i <<= 1
            temp <<= 1
    if not positive:
        res = -res
    return min(max(-2147483648, res), 2147483647)
```

written by tusizi original link here

Substring with Concatenation of All Words(30)

Answer 1

```cpp
// travel all the words combinations to maintain a window
// there are wl(word len) times travel
// each time, n/wl words, mostly 2 times travel for each word
// one left side of the window, the other right side of the window
// so, time complexity O(wl * 2 * N/wl) = O(2N)
vector<int> findSubstring(string S, vector<string> &L) {
    vector<int> ans;
    int n = S.size(), cnt = L.size();
    if (n <= 0 || cnt <= 0) return ans;

    // init word occurence
    unordered_map<string, int> dict;
    for (int i = 0; i < cnt; ++i) dict[L[i]]++;

    // travel all sub string combinations
    int wl = L[0].size();
    for (int i = 0; i < wl; ++i) {
        int left = i, count = 0;
        unordered_map<string, int> tdict;
        for (int j = i; j <= n - wl; j += wl) {
            string str = S.substr(j, wl);
            // a valid word, accumulate results
            if (dict.count(str)) {
                tdict[str]++;
                if (tdict[str] <= dict[str])
                    count++;
                else {
                    // a more word, advance the window left side possiablly
                    while (tdict[str] > dict[str]) {
                        string str1 = S.substr(left, wl);
                        tdict[str1]--;
                        if (tdict[str1] < dict[str1]) count--;
                        left += wl;
                    }
                }
                // come to a result
                if (count == cnt) {
                    ans.push_back(left);
                    // advance one word
                    tdict[S.substr(left, wl)]--;
                    count--;
                    left += wl;
                }
            }
            // not a valid word, reset all vars
            else {
                tdict.clear();
                count = 0;
                left = j + wl;
            }
        }
    }

    return ans;
}
```

written by shichaotan original link here

Answer 2

```
class Solution {
// The general idea:
// Construct a hash function f for L, f: vector<string> -> int,
// Then use the return value of f to check whether a substring is a concatenation

// of all words in L.
// f has two levels, the first level is a hash function f1 for every single word
in L.
// f1 : string -> double
// So with f1, L is converted into a vector of float numbers
// Then another hash function f2 is defined to convert a vector of doubles into a
single int.
// Finally f(L) := f2(f1(L))
// To obtain lower complexity, we require f1 and f2 can be computed through moving
window.
// The following corner case also needs to be considered:
// f2(f1(["ab", "cd"])) != f2(f1(["ac", "bd"]))
// There are many possible options for f2 and f1.
// The following code only shows one possibility (probably not the best),
// f2 is the function "hash" in the class,
// f1([a1, a2, ... , an]) := int( decimal_part(log(a1) + log(a2) + ... + log(an))
* 1000000000 )
public:
    // The complexity of this function is O(nW).
    double hash(double f, double code[], string &word) {
        double result = 0.;
        for (auto &c : word) result = result * f + code[c];
        return result;
    }
    vector<int> findSubstring(string S, vector<string> &L) {
        uniform_real_distribution<double> unif(0., 1.);
        default_random_engine seed;
        double code[128];
        for (auto &d : code) d = unif(seed);
        double f = unif(seed) / 5. + 0.8;
        double value = 0;

        // The complexity of the following for loop is O(L.size( ) * nW).
        for (auto &str : L) value += log(hash(f, code, str));

        int unit = 1e9;
        int key = (value-floor(value))*unit;
        int nS = S.size(), nL = L.size(), nW = L[0].size();
        double fn = pow(f, nW-1.);
        vector<int> result;
        if (nS < nW) return result;
        vector<double> values(nS-nW+1);
        string word(S.begin(), S.begin()+nW);
        values[0] = hash(f, code, word);

        // Use a moving window to hash every word with length nW in S to a float
number,
```

```
        // which is stored in vector values[]
        // The complexity of this step is O(nS).
        for (int i=1; i<=nS-nW; ++i) values[i] = (values[i-1] - code[S[i-1]]*fn)*
f + code[S[i+nW-1]];

        // This for loop will run nW times, each iteration has a complexity O(nS/
nW)
        // So the overall complexity is O(nW * (nS / nW)) = O(nS)
        for (int i=0; i<nW; ++i) {
            int j0=i, j1=i, k=0;
            double sum = 0.;

            // Use a moving window to hash every L.size() continuous words with l
ength nW in S.
            // This while loop will terminate within nS/nW iterations since the i
ncreasement of j1 is nW,
            // So the complexity of this while loop is O(nS / nW).
            while(j1<=nS-nW) {
                sum += log(values[j1]);
                ++k;
                j1 += nW;
                if (k==nL) {
                    int key1 = (sum-floor(sum)) * unit;
                    if (key1==key) result.push_back(j0);
                    sum -= log(values[j0]);
                    --k;
                    j0 += nW;
                }
            }
        }
        return result;
    }
};
```

Though theoretically it has a very small chance to fail.

written by jiajingfei original link here

Answer 3

The following python code is accepted by OJ. It is based on the following idea (assumption)

- We know that two multisets consist of same elements and size of the multisets are equal. if sum of hashes of all elements are the same for these multisets -> those multisets are identical

This is not true for same very very rare cases. Please describe such a case.

```
def findSubstring(self, S, L):
    n = len(L) #num words
    w = len(L[0])  #length of each word
    t = n*w    # total length

    hashsum = sum([hash(x) for x in L])
    h = [hash(S[i:i+w])*(S[i:i+w] in L) for i in xrange(len(S)-w+1)]
    return [i for i in xrange(len(S)-t+1) if sum(h[i:i+t:w])==hashsum]
```

written by hebele original link here

Next Permutation(31)

Answer 1

My idea is for an array:

1. Start from its last element, traverse backward to find the first one with index i that satisfy num[i-1] < num[i]. So, elements from num[i] to num[n-1] is reversely sorted.
2. To find the next permutation, we have to swap some numbers at different positions, to minimize the increased amount, we have to make the highest changed position as high as possible. Notice that index larger than or equal to i is not possible as num[i,n-1] is reversely sorted. So, we want to increase the number at index i-1, clearly, swap it with the smallest number between num[i,n-1] that is larger than num[i-1]. For example, original number is 121543321, we want to swap the '1' at position 2 with '2' at position 7.
3. The last step is to make the remaining higher position part as small as possible, we just have to reversely sort the num[i,n-1]

The following is my code:

```
public void nextPermutation(int[] num) {
    int n=num.length;
    if(n<2)
        return;
    int index=n-1;
    while(index>0){
        if(num[index-1]<num[index])
            break;
        index--;
    }
    if(index==0){
        reverseSort(num,0,n-1);
        return;
    }
    else{
        int val=num[index-1];
        int j=n-1;
        while(j>=index){
            if(num[j]>val)
                break;
            j--;
        }
        swap(num,j,index-1);
        reverseSort(num,index,n-1);
        return;
    }
}

public void swap(int[] num, int i, int j){
    int temp=0;
    temp=num[i];
    num[i]=num[j];
    num[j]=temp;
}

public void reverseSort(int[] num, int start, int end){
    if(start>end)
        return;
    for(int i=start;i<=(end+start)/2;i++)
        swap(num,i,start+end-i);
}
```

written by yuyibestman original link here

Answer 2

Well, in fact the problem of next permutation has been studied long ago. From the Wikipedia page, in the 14th century, a man named Narayana Pandita gives the following classic and yet quite simple algorithm (with minor modifications in notations to fit the problem statement):

1. Find the largest index `k` such that `nums[k] < nums[k + 1]`. If no such index exists, the permutation is sorted in descending order, just reverse it to ascending order and we are done. For example, the next permutation of `[3,`

`2, 1]` is `[1, 2, 3]`.
2. Find the largest index `l` greater than `k` such that `nums[k] < nums[l]`.
3. Swap the value of `nums[k]` with that of `nums[l]`.
4. Reverse the sequence from `nums[k + 1]` up to and including the final element `nums[nums.size() - 1]`.

Quite simple, yeah? Now comes the following code, which is barely a translation.

Well, a final note here, the above algorithm is indeed powerful ---**it can handle the cases of duplicates**! If you have tried the problems Permutations and Permutations II, then the following function is also useful. Both of Permutations and Permutations II can be solved easily using this function. Hints: sort `nums` in ascending order, add it to the result of all permutations and then repeatedly generate the next permutation and add it ... until we get back to the original sorted condition. If you want to learn more, please visit this solution and that solution.

```
class Solution {
    void nextPermutation(vector<int>& nums) {
        int k = -1;
        for (int i = nums.size() - 2; i >= 0; i--) {
            if (nums[i] < nums[i + 1]) {
                k = i;
                break;
            }
        }
        if (k == -1) {
            reverse(nums.begin(), nums.end());
            return;
        }
        int l = -1;
        for (int i = nums.size() - 1; i > k; i--) {
            if (nums[i] > nums[k]) {
                l = i;
                break;
            }
        }
        swap(nums[k], nums[l]);
        reverse(nums.begin() + k + 1, nums.end());
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```
class Solution {
public:
    void nextPermutation(vector<int> &num)
    {
        if (num.empty()) return;

        // in reverse order, find the first number which is in increasing trend ( we call it violated number here)
        int i;
        for (i = num.size()-2; i >= 0; --i)
        {
            if (num[i] < num[i+1]) break;
        }

        // reverse all the numbers after violated number
        reverse(begin(num)+i+1, end(num));
        // if violated number not found, because we have reversed the whole array , then we are done!
        if (i == -1) return;
        // else binary search find the first number larger than the violated numb er
        auto itr = upper_bound(begin(num)+i+1, end(num), num[i]);
        // swap them, done!
        swap(num[i], *itr);
    }
};
```

You might need to think for a while why this would work.

written by AsherBaby original link here

Longest Valid Parentheses(32)

Answer 1

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n = s.length(), longest = 0;
        stack<int> st;
        for (int i = 0; i < n; i++) {
            if (s[i] == '(') st.push(i);
            else {
                if (!st.empty()) {
                    if (s[st.top()] == '(') st.pop();
                    else st.push(i);
                }
                else st.push(i);
            }
        }
        if (st.empty()) longest = n;
        else {
            int a = n, b = 0;
            while (!st.empty()) {
                b = st.top(); st.pop();
                longest = max(longest, a-b-1);
                a = b;
            }
            longest = max(longest, a);
        }
        return longest;
    }
};
```

The workflow of the solution is as below.

1. Scan the string from beginning to end.
2. If current character is '(', push its index to the stack. If current character is ')' and the character at the index of the top of stack is '(', we just find a matching pair so pop from the stack. Otherwise, we push the index of ')' to the stack.
3. After the scan is done, the stack will only contain the indices of characters which cannot be matched. Then let's use the opposite side - substring between adjacent indices should be valid parentheses.
4. If the stack is empty, the whole input string is valid. Otherwise, we can scan the stack to get longest valid substring as described in step 3.

written by cjxxm original link here

Answer 2

My solution uses DP. The main idea is as follows: I construct a array**longest[]**, for any longest[i], it stores the longest length of valid parentheses which is end at i.
And the DP idea is :
If s[i] is '(', set longest[i] to 0,because any string end with '(' cannot be a valid one.

Else if s[i] is ')'
Â Â Â Â If s[i-1] is '(', longest[i] = longest[i-2] + 2
Â Â Â Â Else if s[i-1] is ')' **and s[i-longest[i-1]-1] == '('**, longest[i] = longest[i-1] + 2 + longest[i-longest[i-1]-2]
For example, input "()(())", at i = 5, longest array is [0,2,0,0,2,0], longest[5] = longest[4] + 2 + longest[1] = 6.

```
   int longestValidParentheses(string s) {
            if(s.length() <= 1) return 0;
            int curMax = 0;
            vector<int> longest(s.size(),0);
            for(int i=1; i < s.length(); i++){
                if(s[i] == ')'){
                    if(s[i-1] == '('){
                        longest[i] = (i-2) >= 0 ? (longest[i-2] + 2) : 2;
                        curMax = max(longest[i],curMax);
                    }
                    else{ // if s[i-1] == ')', combine the previous length.
                        if(i-longest[i-1]-1 >= 0 && s[i-longest[i-1]-1] == '('){
                            longest[i] = longest[i-1] + 2 + ((i-longest[i-1]-2 >=
0)?longest[i-longest[i-1]-2]:0);
                            curMax = max(longest[i],curMax);
                        }
                    }
                }
                //else if s[i] == '(', skip it, because longest[i] must be 0
            }
            return curMax;
        }
```

Updated: thanks to **Philip0116**, I have a more concise solution(though this is not as readable as the above one, but concise):

```
int longestValidParentheses(string s) {
        if(s.length() <= 1) return 0;
        int curMax = 0;
        vector<int> longest(s.size(),0);
        for(int i=1; i < s.length(); i++){
            if(s[i] == ')' && i-longest[i-1]-1 >= 0 && s[i-longest[i-1]-1] == '('
){
                    longest[i] = longest[i-1] + 2 + ((i-longest[i-1]-2 >= 0)?long
est[i-longest[i-1]-2]:0);
                    curMax = max(longest[i],curMax);
            }
        }
        return curMax;
    }
```

written by jerryrcwong original link here

Answer 3

```
public class Solution {
public int longestValidParentheses(String s) {
    Stack<Integer> stack = new Stack<Integer>();
    int max=0;
    int left = -1;
    for(int j=0;j<s.length();j++){
        if(s.charAt(j)=='(') stack.push(j);
        else {
            if (stack.isEmpty()) left=j;
            else{
                stack.pop();
                if(stack.isEmpty()) max=Math.max(max,j-left);
                else max=Math.max(max,j-stack.peek());
               }
            }
        }
    return max;
}
```

}

written by jmnjmnjmn original link here

Search in Rotated Sorted Array(33)

Answer 1

```
class Solution {
public:
    int search(int A[], int n, int target) {
        int lo=0,hi=n-1;
        // find the index of the smallest value using binary search.
        // Loop will terminate since mid < hi, and lo or hi will shrink by at leas
t 1.
        // Proof by contradiction that mid < hi: if mid==hi, then lo==hi and loop
would have been terminated.
        while(lo<hi){
            int mid=(lo+hi)/2;
            if(A[mid]>A[hi]) lo=mid+1;
            else hi=mid;
        }
        // lo==hi is the index of the smallest value and also the number of place
s rotated.
        int rot=lo;
        lo=0;hi=n-1;
        // The usual binary search and accounting for rotation.
        while(lo<=hi){
            int mid=(lo+hi)/2;
            int realmid=(mid+rot)%n;
            if(A[realmid]==target)return realmid;
            if(A[realmid]<target)lo=mid+1;
            else hi=mid-1;
        }
        return -1;
    }
};
```

written by lucastan original link here

Answer 2

```
public class Solution {
public int search(int[] A, int target) {
    int lo = 0;
    int hi = A.length - 1;
    while (lo < hi) {
        int mid = (lo + hi) / 2;
        if (A[mid] == target) return mid;

        if (A[lo] <= A[mid]) {
            if (target >= A[lo] && target < A[mid]) {
                hi = mid - 1;
            } else {
                lo = mid + 1;
            }
        } else {
            if (target > A[mid] && target <= A[hi]) {
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
    }
    return A[lo] == target ? lo : -1;
}
```

}

written by jerry13466 original link here

Answer 3

The idea is that when rotating the array, there must be one half of the array that is still in sorted order. For example, 6 7 1 2 3 4 5, the order is disrupted from the point between 7 and 1. So when doing binary search, we can make a judgement that which part is ordered and whether the target is in that range, if yes, continue the search in that half, if not continue in the other half.

```java
public class Solution {
    public int search(int[] nums, int target) {
        int start = 0;
        int end = nums.length - 1;
        while (start <= end){
            int mid = (start + end) / 2;
            if (nums[mid] == target)
                return mid;

            if (nums[start] <= nums[mid]){
                 if (target < nums[mid] && target >= nums[start])
                    end = mid - 1;
                 else
                    start = mid + 1;
            }

            if (nums[mid] <= nums[end]){
                if (target > nums[mid] && target <= nums[end])
                    start = mid + 1;
                 else
                    end = mid - 1;
            }
        }
        return -1;
    }
}
```

written by flyinghx61 original link here

Search for a Range(34)

Answer 1

The problem can be simply broken down as two binary searches for the begining and end of the range, respectively:

First let's find the left boundary of the range. We initialize the range to [i=0, j=n-1]. In each step, calculate the middle element [mid = (i+j)/2]. Now according to the relative value of A[mid] to target, there are three possibilities:

1. If A[mid] < target, then the range must begins on the***right*** of mid (hence i = mid+1 for the next iteration)
2. If A[mid] > target, it means the range must begins on the***left*** of mid (j = mid-1)
3. If A[mid] = target, then the range must begins***on the left of or at*** mid (j= mid)

Since we would move the search range to the same side for case 2 and 3, we might as well merge them as one single case so that less code is needed:

2*. If A[mid] >= target, j = mid;

Surprisingly, 1 and 2* are the only logic you need to put in loop while (i < j). When the while loop terminates, the value of i/j is where the start of the range is. Why?

No matter what the sequence originally is, as we narrow down the search range, eventually we will be at a situation where there are only two elements in the search range. Suppose our target is 5, then we have only 7 possible cases:

```
case 1: [5 7] (A[i] = target < A[j])
case 2: [5 3] (A[i] = target > A[j])
case 3: [5 5] (A[i] = target = A[j])
case 4: [3 5] (A[j] = target > A[i])
case 5: [3 7] (A[i] < target < A[j])
case 6: [3 4] (A[i] < A[j] < target)
case 7: [6 7] (target < A[i] < A[j])
```

For case 1, 2 and 3, if we follow the above rule, since mid = i => A[mid] = target in these cases, then we would set j = mid. Now the loop terminates and i and j both point to the first 5.

For case 4, since A[mid] < target, then set i = mid+1. The loop terminates and both i and j point to 5.

For all other cases, by the time the loop terminates, A[i] is not equal to 5. So we can easily know 5 is not in the sequence if the comparison fails.

In conclusion, when the loop terminates, if A[i]==target, then i is the left boundary of the range; otherwise, just return -1;

For the right of the range, we can use a similar idea. Again we can come up with several rules:

1. If A[mid] > target, then the range must begins on the ***left*** of mid (j = mid-1)
2. If A[mid] < target, then the range must begins on the ***right*** of mid (hence i = mid+1 for the next iteration)
3. If A[mid] = target, then the range must begins ***on the right of or at*** mid (i= mid)

Again, we can merge condition 2 and 3 into:

```
2* If A[mid] <= target, then i = mid;
```

However, the terminate condition on longer works this time. Consider the following case:

```
[5 7], target = 5
```

Now A[mid] = 5, then according to rule 2, we set i = mid. This practically does nothing because i is already equal to mid. As a result, the search range is not moved at all!

The solution is by using a small trick: instead of calculating mid as mid = (i+j)/2, we now do:

```
mid = (i+j)/2+1
```

Why does this trick work? When we use mid = (i+j)/2, the mid is rounded to the lowest integer. In other words, mid is always *biased* towards the left. This means we could have i == mid when j - i == mid, but we NEVER have j == mid. So in order to keep the search range moving, you must make sure the new i is set to something different than mid, otherwise we are at the risk that i gets stuck. But for the new j, it is okay if we set it to mid, since it was not equal to mid anyways. Our two rules in search of the left boundary happen to satisfy these requirements, so it works perfectly in that situation. Similarly, when we search for the right boundary, we must make sure i won't get stuck when we set the new i to i = mid. The easiest way to achieve this is by making mid *biased* to the right, i.e. mid = (i+j)/2+1.

All this reasoning boils down to the following simple code:

```
vector<int> searchRange(int A[], int n, int target) {
    int i = 0, j = n - 1;
    vector<int> ret(2, -1);
    // Search for the left one
    while (i < j)
    {
        int mid = (i + j) /2;
        if (A[mid] < target) i = mid + 1;
        else j = mid;
    }
    if (A[i]!=target) return ret;
    else ret[0] = i;

    // Search for the right one
    j = n-1;  // We don't have to set i to 0 the second time.
    while (i < j)
    {
        int mid = (i + j) /2 + 1;   // Make mid biased to the right
        if (A[mid] > target) j = mid - 1;
        else i = mid;               // So that this won't make the search range s
tuck.
    }
    ret[1] = j;
    return ret;
}
```

written by stellari original link here

Answer 2

```java
public class Solution {
    public int[] searchRange(int[] A, int target) {
        int start = Solution.firstGreaterEqual(A, target);
        if (start == A.length || A[start] != target) {
            return new int[]{-1, -1};
        }
        return new int[]{start, Solution.firstGreaterEqual(A, target + 1) - 1};
    }

    //find the first number that is greater than or equal to target.
    //could return A.length if target is greater than A[A.length-1].
    //actually this is the same as lower_bound in C++ STL.
    private static int firstGreaterEqual(int[] A, int target) {
        int low = 0, high = A.length;
        while (low < high) {
            int mid = low + ((high - low) >> 1);
            //low <= mid < high
            if (A[mid] < target) {
                low = mid + 1;
            } else {
                //should not be mid-1 when A[mid]==target.
                //could be mid even if A[mid]>target because mid<high.
                high = mid;
            }
        }
        return low;
    }
}
```

written by kxcf original link here

Answer 3

```java
public class Solution {
    public int[] searchRange(int[] A, int target) {
        int[] range = {A.length, -1};
        searchRange(A, target, 0, A.length - 1, range);
        if (range[0] > range[1]) range[0] = -1;
        return range;
    }

    public void searchRange(int[] A, int target, int left, int right, int[] range
) {
        if (left > right) return;
        int mid = left + (right - left) / 2;
        if (A[mid] == target) {
            if (mid < range[0]) {
                range[0] = mid;
                searchRange(A, target, left, mid - 1, range);
            }
            if (mid > range[1]) {
                range[1] = mid;
                searchRange(A, target, mid + 1, right, range);
            }
        } else if (A[mid] < target) {
            searchRange(A, target, mid + 1, right, range);
        } else {
            searchRange(A, target, left, mid - 1, range);
        }
    }
}
```

written by xuanaux original link here

Search Insert Position(35)

Answer 1

```
    public int searchInsert(int[] A, int target) {
        int low = 0, high = A.length-1;
        while(low<=high){
            int mid = (low+high)/2;
            if(A[mid] == target) return mid;
            else if(A[mid] > target) high = mid-1;
            else low = mid+1;
        }
        return low;
    }
```

written by AmmsA original link here

Answer 2

If there are duplicate elements equal to *target*, my code will always return the one with smallest index.

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int low = 0, high = nums.size()-1;

        // Invariant: the desired index is between [low, high+1]
        while (low <= high) {
            int mid = low + (high-low)/2;

            if (nums[mid] < target)
                low = mid+1;
            else
                high = mid-1;
        }

        // (1) At this point, low > high. That is, low >= high+1
        // (2) From the invariant, we know that the index is between [low, high+1
], so low <= high+1. Follwing from (1), now we know low == high+1.
        // (3) Following from (2), the index is between [low, high+1] = [low, low
], which means that low is the desired index
        //     Therefore, we return low as the answer. You can also return high+1
as the result, since low == high+1
        return low;
    }
};
```

written by a0806449540 original link here

Answer 3

I think the solution does not need a lot of if statement. Only two cases: 1 if found, just return current index 2 if not found, return next index where the search end

```
int search(int A[], int start, int end, int target) {
    if (start > end) return start;
    int mid = (start + end) / 2;
    if (A[mid] == target) return mid;
    else if (A[mid] > target) return search(A, start, mid - 1, target);
    else return search(A, mid + 1, end, target);
}
int searchInsert(int A[], int n, int target) {
    return search(A, 0, n - 1, target);
}
```

written by yuruofeifei original link here

Valid Sudoku(36)

Answer 1

Three flags are used to check whether a number appear.

used1: check each row

used2: check each column

used3: check each sub-boxes

```
class Solution
{
public:
    bool isValidSudoku(vector<vector<char> > &board)
    {
        int used1[9][9] = {0}, used2[9][9] = {0}, used3[9][9] = {0};

        for(int i = 0; i < board.size(); ++ i)
            for(int j = 0; j < board[i].size(); ++ j)
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0' - 1, k = i / 3 * 3 + j / 3;
                    if(used1[i][num] || used2[j][num] || used3[k][num])
                        return false;
                    used1[i][num] = used2[j][num] = used3[k][num] = 1;
                }

        return true;
    }
};
```

written by makuiyu original link here

Answer 2

```
public class Solution {
public boolean isValidSudoku(char[][] board) {
    for (int i=0; i<9; i++) {
        if (!isParticallyValid(board,i,0,i,8)) return false;
        if (!isParticallyValid(board,0,i,8,i)) return false;
    }
    for (int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if (!isParticallyValid(board,i*3,j*3,i*3+2,j*3+2)) return false;
        }
    }
    return true;
}
private boolean isParticallyValid(char[][] board, int x1, int y1,int x2,int y2){
    Set singleSet = new HashSet();
    for (int i= x1; i<=x2; i++){
        for (int j=y1;j<=y2; j++){
            if (board[i][j]!='.') if(!singleSet.add(board[i][j])) return false;
        }
    }
    return true;
}
```

}

Each time send the coordinates to check if the board is partially valid.

written by bigwolfandtiger original link here

Answer 3

```
   bool isValidSudoku(vector<vector<char>>& board) {
    vector<short> col(9, 0);
    vector<short> block(9, 0);
    vector<short> row(9, 0);
    for (int i = 0; i < 9; i++)
     for (int j = 0; j < 9; j++) {
         if (board[i][j] != '.') {
             int idx = 1 << (board[i][j] - '0');
             if (row[i] & idx || col[j] & idx || block[i/3 * 3 + j / 3] & idx)
                return false;
            row[i] |= idx;
            col[j] |= idx;
            block[i/3 * 3 + j/3] |= idx;
         }
     }
     return true;
  }
```

written by lchen77 original link here

Sudoku Solver(37)

Answer 1

Try 1 through 9 for each cell. Details see comments inside code. Let me know your suggestions.

```
public class Solution {
    public void solveSudoku(char[][] board) {
        if(board == null || board.length == 0)
            return;
        solve(board);
    }

    public boolean solve(char[][] board){
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(board[i][j] == '.'){
                    for(char c = '1'; c <= '9'; c++){//trial. Try 1 through 9 for
each cell
                        if(isValid(board, i, j, c)){
                            board[i][j] = c; //Put c for this cell

                            if(solve(board))
                                return true; //If it's the solution return true
                            else
                                board[i][j] = '.'; //Otherwise go back
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }

    public boolean isValid(char[][] board, int i, int j, char c){
        //Check colum
        for(int row = 0; row < 9; row++)
            if(board[row][j] == c)
                return false;

        //Check row
        for(int col = 0; col < 9; col++)
            if(board[i][col] == c)
                return false;

        //Check 3 x 3 block
        for(int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
            for(int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
                if(board[row][col] == c)
                    return false;
        return true;
    }
}
```

written by kun12 original link here

Answer 2

Update: there's a follow-up 0ms solution which is even more optimized

This is one of the fastest Sudoku solvers I've ever written. It is compact enough - just 150 lines of C++ code with comments. I thought it'd be interesting to share it, since it combines several techniques like reactive network update propagation and backtracking with very aggressive pruning.

The algorithm is online - it starts with an empty board and as you add numbers to it, it starts solving the Sudoku.

Unlike in other solutions where you have bitmasks of allowed/disallowed values per row/column/square, this solution track bitmask for every(!) cell, forming a set of constraints for the allowed values for each particular cell. Once a value is written into a cell, new constraints are immediately propagated to row, column and 3x3 square of the cell. If during this process a value of other cell can be unambiguously deduced - then the value is set, new constraints are propagated, so on.... You can think about this as an implicit reactive network of cells.

If we're lucky (and we'll be lucky for 19 of 20 of Sudokus published in magazines) then Sudoku is solved at the end (or even before!) processing of the input.

Otherwise, there will be empty cells which have to be resolved. Algorithm uses backtracking for this purpose. To optimize it, algorithm starts with the cell with the smallest ambiguity. This could be improved even further by using priority queue (but it's not implemented here). Backtracking is more or less standard, however, at each step we guess the number, the reactive update propagation comes back into play and it either quickly proves that the guess is unfeasible or significantly prunes the remaining search space.

It's interesting to note, that in this case taking and restoring snapshots of the compact representation of the state is faster than doing backtracking rollback by "undoing the moves".

```
class Solution {
    struct cell // encapsulates a single cell on a Sudoku board
    {
        uint8_t value; // cell value 1..9 or 0 if unset
        // number of possible (unconstrained) values for the cell
        uint8_t numPossibilities;
        // if bitset[v] is 1 then value can't be v
        bitset<10> constraints;
        cell() : value(0), numPossibilities(9),constraints() {};
    };
    array<array<cell,9>,9> cells;

    // sets the value of the cell to [v]
    // the function also propagates constraints to other cells and deduce new val
ues where possible
    bool set(int i, int j, int v)
```

```
    {
        // updating state of the cell
        cell& c = cells[i][j];
        if (c.value == v)
            return true;
        if (c.constraints[v])
            return false;
        c.constraints = bitset<10>(0x3FE); // all 1s
        c.constraints.reset(v);
        c.numPossibilities = 1;
        c.value = v;

        // propagating constraints
        for (int k = 0; k<9; k++) {
            // to the row:
            if (i != k && !updateConstraints(k, j, v))
                return false;
            // to the column:
            if (j != k && !updateConstraints(i, k, v))
                return false;
            // to the 3x3 square:
            int ix = (i / 3) * 3 + k / 3;
            int jx = (j / 3) * 3 + k % 3;
            if (ix != i && jx != j && !updateConstraints(ix, jx, v))
                return false;
        }
        return true;
    }
    // update constraints of the cell i,j by excluding possibility of 'excludedVa
lue'
    // once there's one possibility left the function recurses back into set()
    bool updateConstraints(int i, int j, int excludedValue)
    {
        cell& c = cells[i][j];
        if (c.constraints[excludedValue]) {
            return true;
        }
        if (c.value == excludedValue) {
            return false;
        }
        c.constraints.set(excludedValue);
        if (--c.numPossibilities > 1)
            return true;
        for (int v = 1; v <= 9; v++) {
            if (!c.constraints[v]) {
                return set(i, j, v);
            }
        }
        assert(false);
    }

    // backtracking state - list of empty cells
    vector<pair<int, int>> bt;

    // find values for empty cells
    bool findValuesForEmptyCells()
    {
```

```
        // collecting all empty cells
        bt.clear();
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (!cells[i][j].value)
                    bt.push_back(make_pair(i, j));
            }
        }
        // making backtracking efficient by pre-sorting empty cells by numPossibi
lities
        sort(bt.begin(), bt.end(), [this](const pair<int, int>&a, const pair<int,
int>&b) {
            return cells[a.first][a.second].numPossibilities < cells[b.first][b.s
econd].numPossibilities; });
        return backtrack(0);
    }

    // Finds value for all empty cells with index >=k
    bool backtrack(int k)
    {
        if (k >= bt.size())
            return true;
        int i = bt[k].first;
        int j = bt[k].second;
        // fast path - only 1 possibility
        if (cells[i][j].value)
            return backtrack(k + 1);
        auto constraints = cells[i][j].constraints;
        // slow path >1 possibility.
        // making snapshot of the state
        array<array<cell,9>,9> snapshot(cells);
        for (int v = 1; v <= 9; v++) {
            if (!constraints[v]) {
                if (set(i, j, v)) {
                    if (backtrack(k + 1))
                        return true;
                }
                // restoring from snapshot,
                // note: computationally this is cheaper
                // than alternative implementation with undoing the changes
                cells = snapshot;
            }
        }
        return false;
    }
public:
    void solveSudoku(vector<vector<char>> &board) {
        cells = array<array<cell,9>,9>(); // clear array
        // Decoding input board into the internal cell matrix.
        // As we do it - constraints are propagated and even additional values ar
e set as we go
        // (in the case if it is possible to unambiguously deduce them).
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.' && !set(i, j, board[i][j] - '0'))
                    return; // sudoku is either incorrect or unsolvable
```

```
            }
        }
        // if we're lucky we've already got a solution,
        // however, if we have empty cells we need to use backtracking to fill the
m
        if (!findValuesForEmptyCells())
            return; // sudoku is unsolvable

        // copying the solution back to the board
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++) {
                if (cells[i][j].value)
                    board[i][j] = cells[i][j].value + '0';
            }
        }
    }
};
```

written by 0xF4 original link here

Answer 3

Singapore's prime minister Lee Hsien Loong showcased his Sudoku Solver C code. You can read his original Facebook post here and another news reporting it here.

I have made some slight modification to adapt it so it can betested on LeetCode OJ. It passed all 6/6 test cases with a runtime of **1 ms**. Pretty impressive for a prime minister, huh?

```
// Original author: Hsien Loong Lee (http://bit.ly/1zfIGMc)
// Slight modification by @1337c0d3r to adapt to run on LeetCode OJ.
// https://leetcode.com/problems/sudoku-solver/
int InBlock[81], InRow[81], InCol[81];

const int BLANK = 0;
const int ONES = 0x3fe;     // Binary 1111111110

int Entry[81];  // Records entries 1-9 in the grid, as the corresponding bit set
to 1
int Block[9], Row[9], Col[9];   // Each int is a 9-bit array

int SeqPtr = 0;
int Sequence[81];



void SwapSeqEntries(int S1, int S2)
{
     int temp = Sequence[S2];
     Sequence[S2] = Sequence[S1];
     Sequence[S1] = temp;
}
```

```
void InitEntry(int i, int j, int val)
{
     int Square = 9 * i + j;
     int valbit = 1 << val;
     int SeqPtr2;

     // add suitable checks for data consistency

     Entry[Square] = valbit;
     Block[InBlock[Square]] &= ~valbit;
     Col[InCol[Square]] &= ~valbit; // Simpler Col[j] &= ~valbit;
     Row[InRow[Square]] &= ~valbit; // Simpler Row[i] &= ~valbit;

     SeqPtr2 = SeqPtr;
     while (SeqPtr2 < 81 && Sequence[SeqPtr2] != Square)
           SeqPtr2++ ;

     SwapSeqEntries(SeqPtr, SeqPtr2);
     SeqPtr++;
}


void PrintArray(char **board)
{
     int i, j, valbit, val, Square;
     char ch;

     Square = 0;

     for (i = 0; i < 9; i++) {
         for (j = 0; j < 9; j++) {
             valbit = Entry[Square++];
             if (valbit == 0) ch = '-';
             else {
                 for (val = 1; val <= 9; val++)
                     if (valbit == (1 << val)) {
                        ch = '0' + val;
                        break;
                     }
             }
             board[i][j] = ch;
         }
     }
}


int NextSeq(int S)
{
    int S2, Square, Possibles, BitCount;
    int T, MinBitCount = 100;

    for (T = S; T < 81; T++) {
        Square = Sequence[T];
        Possibles = Block[InBlock[Square]] & Row[InRow[Square]] & Col[InCol[Squar
e]];
        BitCount = 0;
```

```
        while (Possibles) {
           Possibles &= ~(Possibles & -Possibles);
           BitCount++;
        }

        if (BitCount < MinBitCount) {
           MinBitCount = BitCount;
           S2 = T;
        }
    }

    return S2;
}


void Place(int S, char** board)
{
    if (S >= 81) {
        PrintArray(board);
        return;
    }

    int S2 = NextSeq(S);
    SwapSeqEntries(S, S2);

    int Square = Sequence[S];

    int     BlockIndex = InBlock[Square],
            RowIndex = InRow[Square],
            ColIndex = InCol[Square];

    int     Possibles = Block[BlockIndex] & Row[RowIndex] & Col[ColIndex];
    while (Possibles) {
          int valbit = Possibles & (-Possibles); // Lowest 1 bit in Possibles
          Possibles &= ~valbit;
          Entry[Square] = valbit;
          Block[BlockIndex] &= ~valbit;
          Row[RowIndex] &= ~valbit;
          Col[ColIndex] &= ~valbit;

          Place(S + 1, board);

          Entry[Square] = BLANK; // Could be moved out of the loop
          Block[BlockIndex] |= valbit;
          Row[RowIndex] |= valbit;
          Col[ColIndex] |= valbit;
    }

    SwapSeqEntries(S, S2);
}

void solveSudoku(char **board, int m, int n) {
    SeqPtr = 0;
    int i, j, Square;

    for (i = 0; i < 9; i++)
        for (j = 0; j < 9; j++) {
```

```
            Square = 9 * i + j;
            InRow[Square] = i;
            InCol[Square] = j;
            InBlock[Square] = (i / 3) * 3 + ( j / 3);
        }


    for (Square = 0; Square < 81; Square++) {
        Sequence[Square] = Square;
        Entry[Square] = BLANK;
    }

    for (i = 0; i < 9; i++)
        Block[i] = Row[i] = Col[i] = ONES;

    for (int i = 0; i < 9; ++i)
       for (int j = 0; j < 9; ++j) {
           if ('.' != board[i][j])
                InitEntry(i, j, board[i][j] - '0');
       }

    Place(SeqPtr, board);
}
```

written by 1337c0d3r original link here

Count and Say(38)

Answer 1

It seems not only me misunderstood the question. Please modify the description, since it's frustrating if you are solving a "different" question. Thanks.

written by boa1150 original link here

Answer 2

At the beginning, I got confusions about what is the nth sequence. Well, my solution is accepted now, so I'm going to give some examples of nth sequence here. The following are sequence from n=1 to n=10:

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
6.     312211
7.     13112221
8.     1113213211
9.     31131211131221
10.   13211311123113112211
```

From the examples you can see, the (i+1)th sequence is the "count and say" of the ith sequence!

Hope this helps!

written by xin15 original link here

Answer 3

Because usually we start from the 0 th item, so add this description to avoid misunderstanding.

written by yuyibestman original link here

Combination Sum(39)

Answer 1

Accepted 16ms c++ solution use backtracking for Combination Sum:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum(std::vector<int> &candidates, i
nt target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum(std::vector<int> &candidates, int target, std::vector<std
::vector<int> > &res, std::vector<int> &combination, int begin) {
        if  (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >= candidates[i]; ++
i) {
            combination.push_back(candidates[i]);
            combinationSum(candidates, target - candidates[i], res, combination,
i);
            combination.pop_back();
        }
    }
};
```

Accepted 12ms c++ solution use backtracking for Combination Sum II:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum2(std::vector<int> &candidates,
int target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum2(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum2(std::vector<int> &candidates, int target, std::vector<st
d::vector<int> > &res, std::vector<int> &combination, int begin) {
        if  (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >= candidates[i]; ++
i)
            if (i == begin || candidates[i] != candidates[i - 1]) {
                combination.push_back(candidates[i]);
                combinationSum2(candidates, target - candidates[i], res, combinat
ion, i + 1);
                combination.pop_back();
            }
    }
};
```

Accepted oms c++ solution use backtracking for Combination Sum III:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum3(int k, int n) {
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum3(n, res, combination, 1, k);
        return res;
    }
private:
    void combinationSum3(int target, std::vector<std::vector<int> > &res, std::ve
ctor<int> &combination, int begin, int need) {
        if (!target) {
            res.push_back(combination);
            return;
        }
        else if (!need)
            return;
        for (int i = begin; i != 10 && target >= i * need + need * (need - 1) / 2
; ++i) {
            combination.push_back(i);
            combinationSum3(target - i, res, combination, i + 1, need - 1);
            combination.pop_back();
        }
    }
};
```

written by prime_tang original link here

Answer 2

Sort the candidates and we choose from small to large recursively, every time we add a candidate to our possible sub result, we subtract the target to a new smaller one.

```java
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> ret = new LinkedList<List<Integer>>();
    Arrays.sort(candidates); // sort the candidates
    // collect possible candidates from small to large to eliminate duplicates,
    recurse(new ArrayList<Integer>(), target, candidates, 0, ret);
    return ret;
}

// the index here means we are allowed to choose candidates from that index
private void recurse(List<Integer> list, int target, int[] candidates, int index,
List<List<Integer>> ret) {
    if (target == 0) {
        ret.add(list);
        return;
    }
    for (int i = index; i < candidates.length; i++) {
        int newTarget = target - candidates[i];
        if (newTarget >= 0) {
            List<Integer> copy = new ArrayList<Integer>(list);
            copy.add(candidates[i]);
            recurse(copy, newTarget, candidates, i, ret);
        } else {
            break;
        }
    }
}
```

written by dylan_yu original link here

Answer 3

```java
public class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        Arrays.sort(candidates);
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        getResult(result, new ArrayList<Integer>(), candidates, target, 0);

        return result;
    }

    private void getResult(List<List<Integer>> result, List<Integer> cur, int can
didates[], int target, int start){
        if(target > 0){
            for(int i = start; i < candidates.length && target >= candidates[i];
i++){
                cur.add(candidates[i]);
                getResult(result, cur, candidates, target - candidates[i], i);
                cur.remove(cur.size() - 1);
            }//for
        }//if
        else if(target == 0 ){
            result.add(new ArrayList<Integer>(cur));
        }//else if
    }
}
```

written by luckygxf original link here

Combination Sum II(40)

Answer 1

At the beginning, I stuck on this problem. After careful thought, I think this kind of backtracking contains a iterative component and a resursive component so I'd like to give more details to help beginners save time. The revursive component tries the elements after the current one and also tries duplicate elements. So we can get correct answer for cases like [1 1] 2. The iterative component checks duplicate combinations and skip it if it is. So we can get correct answer for cases like [1 1 1] 2.

```
class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int> &num, int target)
    {
        vector<vector<int>> res;
        sort(num.begin(),num.end());
        vector<int> local;
        findCombination(res, 0, target, local, num);
        return res;
    }
    void findCombination(vector<vector<int>>& res, const int order, const int tar
get, vector<int>& local, const vector<int>& num)
    {
        if(target==0)
        {
            res.push_back(local);
            return;
        }
        else
        {
            for(int i = order;i<num.size();i++) // iterative component
            {
                if(num[i]>target) return;
                if(i&&num[i]==num[i-1]&&i>order) continue; // check duplicate com
bination
                local.push_back(num[i]),
                findCombination(res,i+1,target-num[i],local,num); // recursive co
mponenet
                local.pop_back();
            }
        }
    }
};
```

written by luming.zhang.75 original link here

Answer 2

```
 public List<List<Integer>> combinationSum2(int[] cand, int target) {
    Arrays.sort(cand);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs_com(cand, 0, target, path, res);
    return res;
}
void dfs_com(int[] cand, int cur, int target, List<Integer> path, List<List<Intege
r>> res) {
    if (target == 0) {
        res.add(new ArrayList(path));
        return ;
    }
    if (target < 0) return;
    for (int i = cur; i < cand.length; i++){
        if (i > cur && cand[i] == cand[i-1]) continue;
        path.add(path.size(), cand[i]);
        dfs_com(cand, i+1, target - cand[i], path, res);
        path.remove(path.size()-1);
    }
}
```

written by lchen77 original link here

Answer 3

I also did it with recursion, turns out the DP solution is 3~4 times faster.

```
def combinationSum2(self, candidates, target):
    candidates.sort()
    table = [None] + [set() for i in range(target)]
    for i in candidates:
        if i > target:
            break
        for j in range(target - i, 0, -1):
            table[i + j] |= {elt + (i,) for elt in table[j]}
        table[i].add((i,))
    return map(list, table[target])
```

written by ChuntaoLu original link here

First Missing Positive(41)

Answer 1

Put each number in its right place.

For example:

When we find 5, then swap it with A[4].

At last, the first place where its number is not right, return the place + 1.

```
class Solution
{
public:
    int firstMissingPositive(int A[], int n)
    {
        for(int i = 0; i < n; ++ i)
            while(A[i] > 0 && A[i] <= n && A[A[i] - 1] != A[i])
                swap(A[i], A[A[i] - 1]);

        for(int i = 0; i < n; ++ i)
            if(A[i] != i + 1)
                return i + 1;

        return n + 1;
    }
};
```

written by makuiyu original link here

Answer 2

Share my O(n)/O(1) solution

The basic idea is ***for any k positive numbers (duplicates allowed), the first missing positive number must be within [1,k+1]***. The reason is like you put k balls into k+1 bins, there must be a bin empty, the empty bin can be viewed as the missing number.

---

1. Unfortunately, there are 0 and negative numbers in the array, so firstly I think of using partition technique (used in quick sort) to put all positive numbers together in one side. This can be finished in O(n) time, O(1) space.
2. After partition step, you get all the positive numbers lying within A[0,k-1]. Now, According to the basic idea, I infer the first missing number must be within [1,k+1]. I decide to use A[i] (0<=i<=k-1) to indicate whether the number (i+1) exists. But here I still have to main the original information A[i] holds. Fortunately, A[i] are all positive numbers, so I can set them to negative to indicate the existence of (i+1) and I can still use abs(A[i]) to get the original information A[i] holds.

3. After step 2, I can again scan all elements between A[0,k-1] to find the first positive element A[i], that means (i+1) doesn't exist, which is what I want.

```
 public int firstMissingPositive(int[] A) {
    int n=A.length;
    if(n==0)
        return 1;
    int k=partition(A)+1;
    int temp=0;
    int first_missing_Index=k;
    for(int i=0;i<k;i++){
        temp=Math.abs(A[i]);
        if(temp<=k)
            A[temp-1]=(A[temp-1]<0)?A[temp-1]:-A[temp-1];
    }
    for(int i=0;i<k;i++){
        if(A[i]>0){
            first_missing_Index=i;
            break;
        }
    }
    return first_missing_Index+1;
}

public int partition(int[] A){
    int n=A.length;
    int q=-1;
    for(int i=0;i<n;i++){
        if(A[i]>0){
            q++;
            swap(A,q,i);
        }
    }
    return q;
}

public void swap(int[] A, int i, int j){
    if(i!=j){
        A[i]^=A[j];
        A[j]^=A[i];
        A[i]^=A[j];
    }
}
```

written by yuyibestman original link here

Answer 3

time complexity is O(N) and space complexity is O(1).
Link: http://stackoverflow.com/questions/1586858/find-the-smallest-integer-not-in-a-list
Posted by Ants Aasma on Oct 20 '09.

The code is pasted here:

```
#Pass 1, move every value to the position of its value
for cursor in range(N):
    target = array[cursor]
    while target < N and target != array[target]:
        new_target = array[target]
        array[target] = target
        target = new_target

#Pass 2, find first location where the index doesn't match the value
for cursor in range(N):
    if array[cursor] != cursor:
        return cursor
return N
```

written by yzhao original link here

Trapping Rain Water(42)

Answer 1

Here is my idea: instead of calculating area by height*width, we can think it in a cumulative way. In other words, sum water amount of each bin(width=1). Search from left to right and maintain a max height of left and right separately, which is like a one-side wall of partial container. Fix the higher one and flow water from the lower part. For example, if current height of left is lower, we fill water in the left bin. Until left meets right, we filled the whole container.

```
class Solution {
public:
    int trap(int A[], int n) {
        int left=0; int right=n-1;
        int res=0;
        int maxleft=0, maxright=0;
        while(left<=right){
            if(A[left]<=A[right]){
                if(A[left]>=maxleft) maxleft=A[left];
                else res+=maxleft-A[left];
                left++;
            }
            else{
                if(A[right]>=maxright) maxright= A[right];
                else res+=maxright-A[right];
                right--;
            }
        }
        return res;
    }
};
```

written by mcrystal original link here

Answer 2

Keep track of the maximum height from both forward directions backward directions, call them leftmax and rightmax.

```
public int trap(int[] A){
    int a=0;
    int b=A.length-1;
    int max=0;
    int leftmax=0;
    int rightmax=0;
    while(a<=b){
        leftmax=Math.max(leftmax,A[a]);
        rightmax=Math.max(rightmax,A[b]);
        if(leftmax<rightmax){
            max+=(leftmax-A[a]);       // leftmax is smaller than rightmax, so th
e (leftmax-A[a]) water can be stored
            a++;
        }
        else{
            max+=(rightmax-A[b]);
            b--;
        }
    }
    return max;
}
```

written by yuyibestman original link here

Answer 3

Keep track of the already safe `level` and the total `water` so far. In each step, process and discard the `lower` one of the leftmost or rightmost elevation.

C

Changing the given parameters to discard the lower border. I'm quite fond of this one.

```
int trap(int* height, int n) {
    int level = 0, water = 0;
    while (n--) {
        int lower = *height < height[n] ? *height++ : height[n];
        if (lower > level) level = lower;
        water += level - lower;
    }
    return water;
}
```

Slight variation with two pointers (left and right).

```
int trap(int* height, int n) {
    int *L = height, *R = L+n-1, level = 0, water = 0;
    while (L < R) {
        int lower = *L < *R ? *L++ : *R--;
        if (lower > level) level = lower;
        water += level - lower;
    }
    return water;
}
```

C++

With left and right index.

```
int trap(vector<int>& height) {
    int l = 0, r = height.size()-1, level = 0, water = 0;
    while (l < r) {
        int lower = height[height[l] < height[r] ? l++ : r--];
        level = max(level, lower);
        water += level - lower;
    }
    return water;
}
```

With left and right iterator.

```
int trap(vector<int>& height) {
    auto l = height.begin(), r = height.end() - 1;
    int level = 0, water = 0;
    while (l != r + 1) {
        int lower = *l < *r ? *l++ : *r--;
        level = max(level, lower);
        water += level - lower;
    }
    return water;
}
```

written by StefanPochmann original link here

Multiply Strings(43)

Answer 1

This is the standard manual multiplication algorithm. We use two nested for loops, working backward from the end of each input number. We pre-allocate our result and accumulate our partial result in there. One special case to note is when our carry requires us to write to our sum string outside of our for loop.

At the end, we trim any leading zeros, or return 0 if we computed nothing but zeros.

```
string multiply(string num1, string num2) {
    string sum(num1.size() + num2.size(), '0');

    for (int i = num1.size() - 1; 0 <= i; --i) {
        int carry = 0;
        for (int j = num2.size() - 1; 0 <= j; --j) {
            int tmp = (sum[i + j + 1] - '0') + (num1[i] - '0') * (num2[j] - '0')
+ carry;
            sum[i + j + 1] = tmp % 10 + '0';
            carry = tmp / 10;
        }
        sum[i] += carry;
    }

    size_t startpos = sum.find_first_not_of("0");
    if (string::npos != startpos) {
        return sum.substr(startpos);
    }
    return "0";
}
```

written by ChiangKaiShrek original link here

Answer 2

```java
public class Solution {
    public String multiply(String num1, String num2) {
        int n1 = num1.length(), n2 = num2.length();
        int[] products = new int[n1 + n2];
        for (int i = n1 - 1; i >= 0; i--) {
            for (int j = n2 - 1; j >= 0; j--) {
                int d1 = num1.charAt(i) - '0';
                int d2 = num2.charAt(j) - '0';
                products[i + j + 1] += d1 * d2;
            }
        }
        int carry = 0;
        for (int i = products.length - 1; i >= 0; i--) {
            int tmp = (products[i] + carry) % 10;
            carry = (products[i] + carry) / 10;
            products[i] = tmp;
        }
        StringBuilder sb = new StringBuilder();
        for (int num : products) sb.append(num);
        while (sb.length() != 0 && sb.charAt(0) == '0') sb.deleteCharAt(0);
        return sb.length() == 0 ? "0" : sb.toString();
    }
}
```

If we break it into steps, it will have the following steps. 1. compute products from each pair of digits from num1 and num2. 2. carry each element over. 3. output the solution.

Things to note:

1. The product of two numbers cannot exceed the sum of the two lengths. (e.g. 99 * 99 cannot be five digit)

2.

```java
int d1 = num1.charAt(i) - '0';
int d2 = num2.charAt(j) - '0';
products[i + j + 1] += d1 * d2;
```

written by lx223 original link here

Answer 3

The key part is to use a vector to store all digits REVERSELY. after the calculation, find the rightmost NON-Zero digits and convert it to a string.

```cpp
class Solution {
public:
    string multiply(string num1, string num2) {

        unsigned int l1=num1.size(),l2=num2.size();
        if (l1==0||l2==0) return "0";

        vector<int> v(l1+l2,0);

        for (unsigned int i=0;i<l1;i++){
            int carry=0;
            int n1=(int)(num1[l1-i-1]-'0');//Calculate from rightmost to left
            for (unsigned int j=0;j<l2;j++){
                int n2=(num2[l2-j-1]-'0');//Calculate from rightmost to left

                int sum=n1*n2+v[i+j]+carry;
                carry=sum/10;
                v[i+j]=sum%10;
            }
            if (carry>0)
                v[i+l2]+=carry;

        }
        int start=l1+l2-1;
        while(v[start]==0) start--;
        if (start==-1) return "0";

        string s="";
        for (int i=start;i>=0;i--)
            s+=(char)(v[i]+'0');
        return s;
    }
};
```

written by reeclapple original link here

Wildcard Matching(44)

Answer 1

I found this solution from http://yucoding.blogspot.com/2013/02/leetcode-question-123-wildcard-matching.html

---

The basic idea is to have one pointer for the string and one pointer for the pattern. This algorithm iterates at most length(string) + length(pattern) times, for each iteration, at least one pointer advance one step.

Here is Yu's elegant solution in C++

```
 bool isMatch(const char *s, const char *p) {
        const char* star=NULL;
        const char* ss=s;
        while (*s){
            //advancing both pointers when (both characters match) or ('?' found
in pattern)
            //note that *p will not advance beyond its length
            if ((*p=='?')||(*p==*s)){s++;p++;continue;}

            // * found in pattern, track index of *, only advancing pattern point
er
            if (*p=='*'){star=p++; ss=s;continue;}

            //current characters didn't match, last pattern pointer was *, curren
t pattern pointer is not *
            //only advancing pattern pointer
            if (star){ p = star+1; s=++ss;continue;}

           //current pattern pointer is not star, last patter pointer was not *
           //characters do not match
            return false;
        }

       //check for remaining characters in pattern
        while (*p=='*'){p++;}

        return !*p;
    }
```

Here is my re-write in Java

```
ï»¿ï»¿boolean comparison(String str, String pattern) {
        int s = 0, p = 0, match = 0, starIdx = -1;
        while (s < str.length()){
            // advancing both pointers
            if (p < pattern.length()  && (pattern.charAt(p) == '?' || str.charAt(
s) == pattern.charAt(p))){
                s++;
                p++;
            }
            // * found, only advancing pattern pointer
            else if (p < pattern.length() && pattern.charAt(p) == '*'){
                starIdx = p;
                match = s;
                p++;
            }
           // last pattern pointer was *, advancing string pointer
            else if (starIdx != -1){
                p = starIdx + 1;
                match++;
                s = match;
            }
           //current pattern pointer is not star, last patter pointer was not *
          //characters do not match
            else return false;
        }

        //check for remaining characters in pattern
        while (p < pattern.length() && pattern.charAt(p) == '*')
            p++;

        return p == pattern.length();
}
```

written by pandora111 original link here

Answer 2

```
class Solution:
# @return a boolean
def isMatch(self, s, p):
    length = len(s)
    if len(p) - p.count('*') > length:
        return False
    dp = [True] + [False]*length
    for i in p:
        if i != '*':
            for n in reversed(range(length)):
                dp[n+1] = dp[n] and (i == s[n] or i == '?')
        else:
            for n in range(1, length+1):
                dp[n] = dp[n-1] or dp[n]
        dp[0] = dp[0] and i == '*'
    return dp[-1]
```

dp[n] means the substring s[:n] if match the pattern i

dp[0] means the empty string '' or s[:0] which only match the pattern '*'

use the reversed builtin because for every dp[n+1] we use the previous 'dp'

add Java O(m*n) version code

```
public boolean isMatch(String s, String p) {
    int count = 0;
    for (char c : p.toCharArray()) {
        if (c == '*')
            count++;
    }
    if (p.length() - count > s.length())
        return false;
    boolean[][] dp = new boolean[p.length() + 1][s.length() + 1];
    dp[0][0] = true;
    for (int j = 1; j <= p.length(); j++) {
        char pattern = p.charAt(j - 1);
        dp[j][0] = dp[j - 1][0] && pattern == '*';
        for (int i = 1; i <= s.length(); i++) {
            char letter = s.charAt(i - 1);
            if (pattern != '*') {
                dp[j][i] = dp[j - 1][i - 1] && (pattern == '?' || pattern == lett
er);
            } else
                dp[j][i] = dp[j][i - 1] || dp[j - 1][i];
        }
    }
    return dp[p.length()][s.length()];
}
```

written by tusizi original link here

Answer 3

Updated: Since the OJ has relaxed the time constraint, the following DP solution is now accepted without the trick :-)

Well, so many people has tried to solve this problem using DP. And almost all of them get TLE (if you see a **C++** DP solution that gets accepted, please let me know ^_^). Well, this post aims at providing an **accpted** DP solution which uses a **trick** to get around the largest test case, insteaed of a solution that is **fully correct**. So please do not give me down votes for that :-)

Let's briefly summarize the idea of DP. We define the state `P[i][j]` to be whether `s[0..i)` matches `p[0..j)`. The state equations are as follows:

1. `P[i][j] = P[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '?')`, if `p[j - 1] != '*'`;
2. `P[i][j] = P[i][j - 1] || P[i - 1][j]`, if `p[j - 1] == '*'`.

If you feel confused with the second equation, you may refer tothis link. There is an

explanation in the comments.

We optimize the DP code to `O(m)` space by recording `P[i - 1][j - 1]` using a single variable `pre`.

The trick to avoid TLE is to hard-code the result for the largest test case by

```
if (n > 30000) return false;
```

The complete code is as follows.

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length(), n = p.length();
        if (n > 30000) return false; // the trick
        vector<bool> cur(m + 1, false);
        cur[0] = true;
        for (int j = 1; j <= n; j++) {
            bool pre = cur[0]; // use the value before update
            cur[0] = cur[0] && p[j - 1] == '*';
            for (int i = 1; i <= m; i++) {
                bool temp = cur[i]; // record the value before update
                if (p[j - 1] != '*')
                    cur[i] = pre && (s[i - 1] == p[j - 1] || p[j - 1] == '?');
                else cur[i] = cur[i - 1] || cur[i];
                pre = temp;
            }
        }
        return cur[m];
    }
};
```

For those interested in a fully correct solution, this link has a nice Greedy solution. And I have rewritten the code below to fit the new C++ interface (changed from `char*` to `string`).

```cpp
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length(), n = p.length();
        int i = 0, j = 0, asterisk = -1, match;
        while (i < m) {
            if (j < n && p[j] == '*') {
                match = i;
                asterisk = j++;
            }
            else if (j < n && (s[i] == p[j] || p[j] == '?')) {
                i++;
                j++;
            }
            else if (asterisk >= 0) {
                i = ++match;
                j = asterisk + 1;
            }
            else return false;
        }
        while (j < n && p[j] == '*') j++;
        return j == n;
    }
};
```

written by jianchao.li.fighter original link here

Jump Game II(45)

Answer 1

I try to change this problem to a BFS problem, where nodes in level i are all the nodes that can be reached in i-1th jump. for example. 2 3 1 1 4 , is 2|| 3 1|| 1 4 ||

clearly, the minimum jump of 4 is 2 since 4 is in level 3. my ac code.

```
 int jump(int A[], int n) {
     if(n<2)return 0;
     int level=0,currentMax=0,i=0,nextMax=0;

     while(currentMax-i+1>0){       //nodes count of current level>0
         level++;
         for(;i<=currentMax;i++){   //traverse current level , and update the max reach of next level
             nextMax=max(nextMax,A[i]+i);
             if(nextMax>=n-1)return level;   // if last element is in level+1,  then the min jump=level
         }
         currentMax=nextMax;
     }
     return 0;
 }
```

written by enriquewang original link here

Answer 2

Hi All, below is my AC solution:

```
public int jump(int[] A) {
    int maxReach = A[0];
    int edge = 0;
    int minstep = 0;

    for(int i = 1; i < A.length; i++) {
        if (i > edge) {
            minstep += 1;
            edge = maxReach;
            if(edge > A.length - 1)
                return minstep;
        }
        maxReach = Math.max(maxReach, A[i] + i);
        if (maxReach == i):
            return -1;
    }

    return minstep;
}
```

When iterate the array, I set an edge for the Search phase, which means that if I exceeds the edge, the minstep must add one and the maxReach will be update. And when the last index is within the range of the edge, output the minstep.

[2, 3, 1, 1, 4]

First, the edge is 0; Second, after start iterate the array, it exceeds the edge 0 when reaching the A[0] and update the edge to 2; Third, after it reach the A[2], it exceeds the edge 2 and update the new edge to the maxReach 4. Finally, end of the array is inside the edge, output the minstep.

written by zhoutsby original link here

Answer 3

```
public int jump(int[] A) {
    int sc = 0;
    int e = 0;
    int max = 0;
    for(int i=0; i<A.length-1; i++) {
        max = Math.max(max, i+A[i]);
        if( i == e ) {
            sc++;
            e = max;
        }
    }
    return sc;
}
```

written by adam20 original link here

Permutations(46)

Answer 1

This recursive solution is the my first response for this problem. I was surprised when I found no similar solution posted here. It is much easier to understand than DFS-based ones, at least in my opinion. Please find more explanations here. All comments are welcome.

```
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int> > result;

        permuteRecursive(num, 0, result);
        return result;
    }

    // permute num[begin..end]
    // invariant: num[0..begin-1] have been fixed/permuted
    void permuteRecursive(vector<int> &num, int begin, vector<vector<int> > &resu
lt)    {
        if (begin >= num.size()) {
            // one permutation instance
            result.push_back(num);
            return;
        }

        for (int i = begin; i < num.size(); i++) {
            swap(num[begin], num[i]);
            permuteRecursive(num, begin + 1, result);
            // reset
            swap(num[begin], num[i]);
        }
    }
};
```

written by xiaohui7 original link here

Answer 2

the basic idea is, to permute n numbers, we can add the nth number into the resulting `List<List<Integer>>` from the n-1 numbers, in every possible position.

For example, if the input num[] is {1,2,3}: First, add 1 into the initial `List<List<Integer>>` (let's call it "answer").

Then, 2 can be added in front or after 1. So we have to copy the List in answer (it's just {1}), add 2 in position 0 of {1}, then copy the original {1} again, and add 2 in position 1. Now we have an answer of {{2,1},{1,2}}. There are 2 lists in the current answer.

Then we have to add 3. first copy {2,1} and {1,2}, add 3 in position 0; then copy {2,1} and {1,2}, and add 3 into position 1, then do the same thing for position 3. Finally we

have 2*3=6 lists in answer, which is what we want.

```
public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> ans = new ArrayList<List<Integer>>();
    if (num.length ==0) return ans;
    List<Integer> l0 = new ArrayList<Integer>();
    l0.add(num[0]);
    ans.add(l0);
    for (int i = 1; i< num.length; ++i){
        List<List<Integer>> new_ans = new ArrayList<List<Integer>>();
        for (int j = 0; j<=i; ++j){
           for (List<Integer> l : ans){
               List<Integer> new_l = new ArrayList<Integer>(l);
               new_l.add(j,num[i]);
               new_ans.add(new_l);
           }
        }
        ans = new_ans;
    }
    return ans;
}
```

---

python version is more concise:

```
def permute(self, nums):
    perms = [[]]
    for n in nums:
        new_perms = []
        for perm in perms:
            for i in xrange(len(perm)+1):
                new_perms.append(perm[:i] + [n] + perm[i:])   ###insert n
        perms = new_perms
    return perms
```

written by cbmbbz original link here

Answer 3

```java
public List<List<Integer>> permute(int[] num) {
    LinkedList<List<Integer>> res = new LinkedList<List<Integer>>();
    res.add(new ArrayList<Integer>());
    for (int n : num) {
        int size = res.size();
        for (; size > 0; size--) {
            List<Integer> r = res.pollFirst();
            for (int i = 0; i <= r.size(); i++) {
                List<Integer> t = new ArrayList<Integer>(r);
                t.add(i, n);
                res.add(t);
            }
        }
    }
    return res;
}
```

written by tusizi original link here

Permutations II(47)

Answer 1

```
class Solution {
public:
    void recursion(vector<int> num, int i, int j, vector<vector<int> > &res) {
        if (i == j-1) {
            res.push_back(num);
            return;
        }
        for (int k = i; k < j; k++) {
            if (i != k && num[i] == num[k]) continue;
            swap(num[i], num[k]);
            recursion(num, i+1, j, res);
        }
    }
    vector<vector<int> > permuteUnique(vector<int> &num) {
        sort(num.begin(), num.end());
        vector<vector<int> >res;
        recursion(num, 0, num.size(), res);
        return res;
    }
};
```

written by guoang original link here

Answer 2

```
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int> &S) {
        // res.clear();
        sort(S.begin(), S.end());
        res.push_back(S);
        int j;
        int i = S.size()-1;
        while (1){
            for (i=S.size()-1; i>0; i--){
                if (S[i-1]< S[i]){
                    break;
                }
            }
            if(i == 0){
                break;
            }

            for (j=S.size()-1; j>i-1; j--){
                if (S[j]>S[i-1]){
                    break;
                }
            }
            swap(S[i-1], S[j]);
            reverse(S, i, S.size()-1);
            res.push_back(S);
        }
        return res;
    }
    void reverse(vector<int> &S, int s, int e){
        while (s<e){
            swap(S[s++], S[e--]);
        }
    }

    vector<vector<int> > res;
};
```

Basically, assume we have "1234", the idea is to increase the number in ascending order, so next is "1243", next is "1324", and so on.

written by TransMatrix original link here

Answer 3

I see most solutions are using next permutation. That's great and only uses O(1) space.

Anyway I am sharing backtracking solution which uses O(n) space. This is actually a typical backtracking problem. We can use hash map to check whether the element was already taken. However, we could get TLE if we check vector num every time. So we iterate the hash map instead.

```
class Solution {
public:
vector<vector<int> > permuteUnique(vector<int> &num) {
    vector<vector<int>> v;
    vector<int> r;
    map<int, int> map;
    for (int i : num)
    {
        if (map.find(i) == map.end()) map[i] = 0;
        map[i]++;
    }
    permuteUnique(v, r, map, num.size());
    return v;
}

void permuteUnique(vector<vector<int>> &v, vector<int> &r, map<int, int> &map, in
t n)
{
    if (n <= 0)
    {
        v.push_back(r);
        return;
    }
    for (auto &p : map)
    {
        if (p.second <= 0) continue;
        p.second--;
        r.push_back(p.first);
        permuteUnique(v, r, map, n - 1);
        r.pop_back();
        p.second++;
    }
}
};
```

written by oujiafan original link here

Rotate Image(48)

Answer 1

here give a common method to solve the image rotation problems.

```
/*
 * clockwise rotate
 * first reverse up to down, then swap the symmetry
 * 1 2 3     7 8 9     7 4 1
 * 4 5 6  => 4 5 6  => 8 5 2
 * 7 8 9     1 2 3     9 6 3
*/
void rotate(vector<vector<int> > &matrix) {
    reverse(matrix.begin(), matrix.end());
    for (int i = 0; i < matrix.size(); ++i) {
        for (int j = i + 1; j < matrix[i].size(); ++j)
            swap(matrix[i][j], matrix[j][i]);
    }
}

/*
 * anticlockwise rotate
 * first reverse left to right, then swap the symmetry
 * 1 2 3     3 2 1     3 6 9
 * 4 5 6  => 6 5 4  => 2 5 8
 * 7 8 9     9 8 7     1 4 7
*/
void anti_rotate(vector<vector<int> > &matrix) {
    for (auto vi : matrix) reverse(vi.begin(), vi.end());
    for (int i = 0; i < matrix.size(); ++i) {
        for (int j = i + 1; j < matrix[i].size(); ++j)
            swap(matrix[i][j], matrix[j][i]);
    }
}
```

written by shichaotan original link here

Answer 2

While these solutions are Python, I think they're understandable/interesting for non-Python coders as well. But before I begin: No mathematician would call a matrix `matrix`, so I'll use the usual `A`. Also, btw, the 40 ms reached by two of the solutions is I think the fastest achieved by Python solutions so far.

Most Pythonic - `[::-1]` and `zip` - 44 ms

The most pythonic solution is a simple one-liner using `[::-1]` to flip the matrix upside down and then `zip` to transpose it. It assigns the result back into `A`, so it's "in-place" in a sense and the OJ accepts it as such, though some people might not.

```
class Solution:
    def rotate(self, A):
        A[:] = zip(*A[::-1])
```

Most Direct - 52 ms

A 100% in-place solution. It even reads and writes each matrix element only once and doesn't even use an extra temporary variable to hold them. It walks over the *"top-left quadrant"* of the matrix and directly rotates each element with the three corresponding elements in the other three quadrants. Note that I'm moving the four elements in parallel and that `[~i]` is way nicer than `[n-1-i]`.

```
class Solution:
    def rotate(self, A):
        n = len(A)
        for i in range(n/2):
            for j in range(n-n/2):
                A[i][j], A[~j][i], A[~i][~j], A[j][~i] = \
                         A[~j][i], A[~i][~j], A[j][~i], A[i][j]
```

Clean Most Pythonic - 56 ms

While the OJ accepts the above solution, the the result rows are actually tuples, not lists, so it's a bit dirty. To fix this, we can just apply `list` to every row:

```
class Solution:
    def rotate(self, A):
        A[:] = map(list, zip(*A[::-1]))
```

List Comprehension - 60 ms

If you don't like `zip`, you can use a nested list comprehension instead:

```
class Solution:
    def rotate(self, A):
        A[:] = [[row[i] for row in A[::-1]] for i in range(len(A))]
```

Almost as Direct - 40 ms

If you don't like the little repetitive code of the above "Most Direct" solution, we can instead do each four-cycle of elements by using three swaps of just two elements.

```
class Solution:
    def rotate(self, A):
        n = len(A)
        for i in range(n/2):
            for j in range(n-n/2):
                for _ in '123':
                    A[i][j], A[~j][i], i, j = A[~j][i], A[i][j], ~j, ~i
                i = ~j
```

Flip Flip - 40 ms

Basically the same as the first solution, but using `reverse` instead of `[::-1]` and transposing the matrix with loops instead of `zip`. It's 100% in-place, just instead of only moving elements around, it also moves the rows around.

```
class Solution:
    def rotate(self, A):
        A.reverse()
        for i in range(len(A)):
            for j in range(i):
                A[i][j], A[j][i] = A[j][i], A[i][j]
```

Flip Flip, all by myself - 48 ms

Similar again, but I first transpose and then flip left-right instead of upside-down, and do it all by myself in loops. This one is 100% in-place again in the sense of just moving the elements.

```
class Solution:
    def rotate(self, A):
        n = len(A)
        for i in range(n):
            for j in range(i):
                A[i][j], A[j][i] = A[j][i], A[i][j]
        for row in A:
            for j in range(n/2):
                row[j], row[~j] = row[~j], row[j]
```

written by StefanPochmann original link here

Answer 3

The idea was firstly transpose the matrix and then flip it symmetrically. For instance,

```
1  2  3
4  5  6
7  8  9
```

after transpose, it will be swap(matrix[i][j], matrix[j][i])

```
1  4  7
2  5  8
3  6  9
```

Then flip the matrix horizontally. (swap(matrix[i][j], matrix[i][matrix.length-1-j])

```
7  4  1
8  5  2
9  6  3
```

Hope this helps.

```
public class Solution {
    public void rotate(int[][] matrix) {
        for(int i = 0; i<matrix.length; i++){
            for(int j = i; j<matrix[0].length; j++){
                int temp = 0;
                temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
        for(int i =0 ; i<matrix.length; i++){
            for(int j = 0; j<matrix.length/2; j++){
                int temp = 0;
                temp = matrix[i][j];
                matrix[i][j] = matrix[i][matrix.length-1-j];
                matrix[i][matrix.length-1-j] = temp;
            }
        }
    }
}
```

written by LuckyIdiot original link here

Group Anagrams(49)

Answer 1

What does it mean "return all groups"? But the return result is vector? How can we return all groups? I mean, for example, we have such vector ["dog","cat","god","tac"]. What should I return?

written by htzfun original link here

Answer 2

Why the output is list and not a list of list. What If there are multiple groups of anagrams?

written by mahdy original link here

Answer 3

```
vector<string> anagrams(vector<string> &strs) {
    vector<string> result;
    vector<string> sortedStrs = strs;
    unordered_map<string, vector<int>> map;
    for(int i = 0; i < strs.size(); i++){
        sort(sortedStrs[i].begin(), sortedStrs[i].end());
        map[sortedStrs[i]].push_back(i);
    }
    for(auto it = map.begin(); it != map.end(); it++){
        if(it->second.size() > 1){
            for(int i = 0; i < it->second.size(); i++){
                result.push_back(strs[it->second[i]]);
            }
        }
    }
    return result;
}
```

Here is basic idea for this problem.

First, get a copy of "strs". Let's name this copy "sortedStrs".

Second, sort all strings in "sortedStrs".

And we have a hash map `unordered_map<string, vector<int>> map`.

Every string in "sortedStrs" will be recorded in this hash map with its position.

In the second loop, we traverse this hash map. And find each value of which size is larger than 1. Then find the original string in "strs".

Done.

written by zxyperfect original link here

Pow(x, n)(50)

Answer 1

```
public class Solution {
    public double pow(double x, int n) {
        if(n == 0)
            return 1;
        if(n<0){
            n = -n;
            x = 1/x;
        }
        return (n%2 == 0) ? pow(x*x, n/2) : x*pow(x*x, n/2);
    }
}
```

written by pei+heng original link here

Answer 2

/* This is a simple solution based on divide and conquer */

```
public class Solution {
       public double pow(double x, int m) {
           double temp=x;
           if(m==0)
           return 1;
           temp=pow(x,m/2);
           if(m%2==0)
           return temp*temp;
           else
           {
           if(m > 0)
               return x*temp*temp;
           else
               return (temp*temp)/x;
           }

    }
```

written by mohit4 original link here

Answer 3

```
class Solution {
public:
    double myPow(double x, int n) {
        double ans = 1;
        unsigned long long p;
        if (n < 0) {
            p = -n;
            x = 1 / x;
        } else {
            p = n;
        }
        while (p) {
            if (p & 1)
                ans *= x;
            x *= x;
            p >>= 1;
        }
        return ans;
    }
};
```

written by gongruya original link here

N-Queens(51)

Answer 1

In this problem, we can go row by row, and in each position, we need to check if the `column`, the `45Â° diagonal` and the `135Â° diagonal` had a queen before.

Solution A: Directly check the validity of each position, *12ms*:

```
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        solveNQueens(res, nQueens, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (isValid(nQueens, row, col, n)) {
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, row + 1, n);
                nQueens[row][col] = '.';
            }
    }
    bool isValid(std::vector<std::string> &nQueens, int row, int col, int &n) {
        //check if the column had a queen before.
        for (int i = 0; i != row; ++i)
            if (nQueens[i][col] == 'Q')
                return false;
        //check if the 45Â° diagonal had a queen before.
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; --i, --j)
            if (nQueens[i][j] == 'Q')
                return false;
        //check if the 135Â° diagonal had a queen before.
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; --i, ++j)
            if (nQueens[i][j] == 'Q')
                return false;
        return true;
    }
};
```

Solution B: Use flag vectors as bitmask, *4ms*:

The number of columns is `n`, the number of 45Â° diagonals is `2 * n - 1`, the number of 135Â° diagonals is also `2 * n - 1`. When reach `[row, col]`, the column No. is `col`, the 45Â° diagonal No. is `row + col` and the 135Â° diagonal No. is `n - 1 + col - row`. We can use three arrays to indicate if the column or the

diagonal had a queen before, if not, we can put a queen in this position and continue.

```
/**    | | |              / / /            \ \ \
  *    0 0 0             0 0 0             0 0 0
  *    | | |            / / / /           \ \ \ \
  *    0 0 0             0 0 0             0 0 0
  *    | | |            / / / /           \ \ \ \
  *    0 0 0             0 0 0             0 0 0
  *    | | |            / / /               \ \ \
  *   3 columns       5 45Â° diagonals     5 135Â° diagonals    (when n is 3)
  */
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        std::vector<int> flag_col(n, 1), flag_45(2 * n - 1, 1), flag_135(2 * n -
1, 1);
        solveNQueens(res, nQueens, flag_col, flag_45, flag_135, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, std::vector<int> &flag_col, std::vector<int> &flag_45, std::
vector<int> &flag_135, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (flag_col[col] && flag_45[row + col] && flag_135[n - 1 + col - row
]) {
                flag_col[col] = flag_45[row + col] = flag_135[n - 1 + col - row]
= 0;
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, flag_col, flag_45, flag_135, row + 1,
n);
                nQueens[row][col] = '.';
                flag_col[col] = flag_45[row + col] = flag_135[n - 1 + col - row]
= 1;
            }
    }
};
```

But we actually do not need to use three arrays, we just need one. Now, when reach `[row, col]`, the subscript of column is `col`, the subscript of 45Â° diagonal is `n + row + col` and the subscript of 135Â° diagonal is `n + 2 * n - 1 + n - 1 + col - row`.

```
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > res;
        std::vector<std::string> nQueens(n, std::string(n, '.'));
        /*
        flag[0] to flag[n - 1] to indicate if the column had a queen before.
        flag[n] to flag[3 * n - 2] to indicate if the 45Â° diagonal had a queen b
efore.
        flag[3 * n - 1] to flag[5 * n - 3] to indicate if the 135Â° diagonal had
a queen before.
        */
        std::vector<int> flag(5 * n - 2, 1);
        solveNQueens(res, nQueens, flag, 0, n);
        return res;
    }
private:
    void solveNQueens(std::vector<std::vector<std::string> > &res, std::vector<st
d::string> &nQueens, std::vector<int> &flag, int row, int &n) {
        if (row == n) {
            res.push_back(nQueens);
            return;
        }
        for (int col = 0; col != n; ++col)
            if (flag[col] && flag[n + row + col] && flag[4 * n - 2 + col - row])
{
                flag[col] = flag[n + row + col] = flag[4 * n - 2 + col - row] = 0
;
                nQueens[row][col] = 'Q';
                solveNQueens(res, nQueens, flag, row + 1, n);
                nQueens[row][col] = '.';
                flag[col] = flag[n + row + col] = flag[4 * n - 2 + col - row] = 1
;
            }
    }
};
```

written by prime_tang original link here

Answer 2

queens can attack other queen in the same row, same column, but i forget the diagonal.. = ã€,=

written by aqin original link here

Answer 3

Space complexity : Instead of using a 2D array to represent the chess board, i am using a 1D array , the index of which would represent the row number and the value of arr at row index will be the column number for the correct position of the queen.

i.e

```
       Instead of doing  arr[row][col]=1
       i am using arr[row]=col ;              where queen is positioned at (row,
col);
```

Logic : DFS for every column number ,ranging from 0 to n-1, for all the rows from 0 to n-1 and check the validity of queen position for every row,col combination(using isSafe function)

isSafe function : It checks whether the queen in current position(r,c) is being attacked by any of the r-1 queens positioned in row numbers 0 through r-1.

```
class Solution {
   public:
        vector < vector <string> > sol;
        int limit;

        vector<string> toChessString(vector<int> arr)
          {
            string s(arr.size(),'.');
            vector<string> ans(arr.size(),s);

            for(int i=0 ; i<arr.size() ; i++)
              ans[i][arr[i]]='Q';

              return ans;
          }


        bool isSafe(vector<int> arr, int r , int c )
         {
            int check;
            for(int row=r-1,ldia=c-1,rdia=c+1 ; row>=0 ; row--,ldia--,rdia++)
            {
                check=arr[row];

                if(check==c || check==ldia || check==rdia)
                 return false;
            }
            return true;
         }

        void solveNqueen(vector<int> arr , int r , int c)
        {
            if(r==limit)
             sol.push_back(toChessString(arr));

            else
             {
                 for(int col=c ; col<limit ; col++)
                 {
                    arr[r]=col;

                    if(isSafe(arr,r,col))
                      solveNqueen(arr,r+1,0);
                 }
```

```
            }
        }
    }


    vector<vector<string> > solveNQueens(int n) {
        vector<int> arr(n,0);
        limit=n;
        solveNqueen(arr,0,0);

        return sol;
    }
};
```

written by leet_nik original link here

N-Queens II(52)

Answer 1

```
/**
 * don't need to actually place the queen,
 * instead, for each row, try to place without violation on
 * col/ diagonal1/ diagnol2.
 * trick: to detect whether 2 positions sit on the same diagnol:
 * if delta(col, row) equals, same diagnol1;
 * if sum(col, row) equals, same diagnal2.
 */
private final Set<Integer> occupiedCols = new HashSet<Integer>();
private final Set<Integer> occupiedDiag1s = new HashSet<Integer>();
private final Set<Integer> occupiedDiag2s = new HashSet<Integer>();
public int totalNQueens(int n) {
    return totalNQueensHelper(0, 0, n);
}

private int totalNQueensHelper(int row, int count, int n) {
    for (int col = 0; col < n; col++) {
        if (occupiedCols.contains(col))
            continue;
        int diag1 = row - col;
        if (occupiedDiag1s.contains(diag1))
            continue;
        int diag2 = row + col;
        if (occupiedDiag2s.contains(diag2))
            continue;
        // we can now place a queen here
        if (row == n-1)
            count++;
        else {
            occupiedCols.add(col);
            occupiedDiag1s.add(diag1);
            occupiedDiag2s.add(diag2);
            count = totalNQueensHelper(row+1, count, n);
            // recover
            occupiedCols.remove(col);
            occupiedDiag1s.remove(diag1);
            occupiedDiag2s.remove(diag2);
        }
    }

    return count;
}
```

written by AlexTheGreat original link here

Answer 2

```
int totalNQueens(int n) {
    vector<bool> col(n, true);
    vector<bool> anti(2*n-1, true);
    vector<bool> main(2*n-1, true);
    vector<int> row(n, 0);
    int count = 0;
    dfs(0, row, col, main, anti, count);
    return count;
}
void dfs(int i, vector<int> &row, vector<bool> &col, vector<bool>& main, vector<b
ool> &anti, int &count) {
        if (i == row.size()) {
            count++;
            return;
        }
       for (int j = 0; j < col.size(); j++) {
         if (col[j] && main[i+j] && anti[i+col.size()-1-j]) {
             row[i] = j;
             col[j] = main[i+j] = anti[i+col.size()-1-j] = false;
             dfs(i+1, row, col, main, anti, count);
             col[j] = main[i+j] = anti[i+col.size()-1-j] = true;
      }
    }
}
```

written by lchen77 original link here

Answer 3

Following codes got AC. But you should never write some codes like this. This is post is just for joking.

```
/**
 * @param {number} n
 * @return {number}
 */
var totalNQueens = function(n) {
    var p = '', s = 0, l;
    for (var i = 0; i < n; i++) {
        l = '\nfor (var s# = 0; s# < ' + n + '; s#++)';
        for (var j = 0; j < i; j++)
            l += 'if (s# !== s@ && Math.abs(s# - s@) !== (# - @)) '.replace(/@/g,
j);
        p += l.replace(/#/g, i);
    }
    p += '\ns++;\ns';
    return eval(p);
};
```

written by ts original link here

Maximum Subarray(53)

Answer 1

this problem was discussed by Jon Bentley (Sep. 1984 Vol. 27 No. 9 Communications of the ACM P885)

the paragraph below was copied from his paper (with a little modifications)

algorithm that operates on arrays: it starts at the left end (element A[1]) and scans through to the right end (element A[n]), keeping track of the maximum sum subvector seen so far. The maximum is initially A[0]. Suppose we've solved the problem for A[1 .. i - 1]; how can we extend that to A[1 .. i]? The maximum sum in the first I elements is either the maximum sum in the first i - 1 elements (which we'll call MaxSoFar), or it is that of a subvector that ends in position i (which we'll call MaxEndingHere).

MaxEndingHere is either A[i] plus the previous MaxEndingHere, or just A[i], whichever is larger.

```
public static int maxSubArray(int[] A) {
    int maxSoFar=A[0], maxEndingHere=A[0];
    for (int i=1;i<A.length;++i){
        maxEndingHere= Math.max(maxEndingHere+A[i],A[i]);
        maxSoFar=Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```

written by cbmbbz original link here

Answer 2

Analysis of this problem: Apparently, this is a optimization problem, which can be usually solved by DP. So when it comes to DP, the first thing for us to figure out is the format of the sub problem(or the state of each sub problem). The format of the sub problem can be helpful when we are trying to come up with the recursive relation.

At first, I think the sub problem should look like: `maxSubArray(int A[], int i, int j)`, which means the maxSubArray for A[i: j]. In this way, our goal is to figure out what `maxSubArray(A, 0, A.length - 1)` is. However, if we define the format of the sub problem in this way, it's hard to find the connection from the sub problem to the original problem(at least for me). In other words, I can't find a way to divided the original problem into the sub problems and use the solutions of the sub problems to somehow create the solution of the original one.

So I change the format of the sub problem into something like: `maxSubArray(int A[], int i)`, which means the maxSubArray for A[0:i] which must has A[i] as the end element. Note that now the sub problem's format is less flexible and less powerful than the previous one because there's a limitation that A[i] should be

contained in that sequence and we have to keep track of each solution of the sub problem to update the global optimal value. However, now the connect between the sub problem & the original one becomes clearer:

```
maxSubArray(A, i) = maxSubArray(A, i - 1) > 0 ? maxSubArray(A, i - 1) : 0 + A[i];
```

And here's the code

```
public int maxSubArray(int[] A) {
        int n = A.length;
        int[] dp = new int[n];//dp[i] means the maximum subarray ending with A[i]
;
        dp[0] = A[0];
        int max = dp[0];

        for(int i = 1; i < n; i++){
            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
            max = Math.max(max, dp[i]);
        }

        return max;
}
```

written by FujiwaranoSai original link here

Answer 3

Idea is very simple. Basically, keep adding each integer to the sequence until the sum drops below 0. If sum is negative, then should reset the sequence.

```
class Solution {
public:
    int maxSubArray(int A[], int n) {
        int ans=A[0],i,j,sum=0;
        for(i=0;i<n;i++){
            sum+=A[i];
            ans=max(sum,ans);
            sum=max(sum,0);
        }
        return ans;
    }
};
```

written by lucastan original link here

Spiral Matrix(54)

Answer 1

This is a very simple and easy to understand solution. I traverse right and increment rowBegin, then traverse down and decrement colEnd, then I traverse left and decrement rowEnd, and finally I traverse up and increment colBegin.

The only tricky part is that when I traverse left or up I have to check whether the row or col still exists to prevent duplicates. If anyone can do the same thing without that check, please let me know!

Any comments greatly appreciated.

```java
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {

        List<Integer> res = new ArrayList<Integer>();

        if (matrix.length == 0) {
            return res;
        }

        int rowBegin = 0;
        int rowEnd = matrix.length-1;
        int colBegin = 0;
        int colEnd = matrix[0].length - 1;

        while (rowBegin <= rowEnd && colBegin <= colEnd) {
            // Traverse Right
            for (int j = colBegin; j <= colEnd; j ++) {
                res.add(matrix[rowBegin][j]);
            }
            rowBegin++;

            // Traverse Down
            for (int j = rowBegin; j <= rowEnd; j ++) {
                res.add(matrix[j][colEnd]);
            }
            colEnd--;

            if (rowBegin <= rowEnd) {
                // Traverse Left
                for (int j = colEnd; j >= colBegin; j --) {
                    res.add(matrix[rowEnd][j]);
                }
            }
            rowEnd--;

            if (colBegin <= colEnd) {
                // Traver Up
                for (int j = rowEnd; j >= rowBegin; j --) {
                    res.add(matrix[j][colBegin]);
                }
            }
            colBegin ++;
        }

        return res;
    }
}
```

written by qwl5004 original link here

Answer 2

When traversing the matrix in the spiral order, at any time we follow one out of the following four directions: RIGHT DOWN LEFT UP. Suppose we are working on a 5 x 3 matrix as such:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Imagine a cursor starts off at (0, -1), i.e. the position at '0', then we can achieve the spiral order by doing the following:

1. Go right 5 times
2. Go down 2 times
3. Go left 4 times
4. Go up 1 times.
5. Go right 3 times
6. Go down 0 times -> quit

Notice that the directions we choose always follow the order 'right->down->left->up', and for horizontal movements, the number of shifts follows:{5, 4, 3}, and vertical movements follows {2, 1, 0}.

Thus, we can make use of a direction matrix that records the offset for all directions, then an array of two elements that stores the number of shifts for horizontal and vertical movements, respectively. This way, we really just need one for loop instead of four.

Another good thing about this implementation is that: If later we decided to do spiral traversal on a different direction (e.g. Counterclockwise), then we only need to change the Direction matrix; the main loop does not need to be touched.

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
    vector<vector<int> > dirs{{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    vector<int> res;
    int nr = matrix.size();     if (nr == 0) return res;
    int nc = matrix[0].size();  if (nc == 0) return res;

    vector<int> nSteps{nc, nr-1};

    int iDir = 0;   // index of direction.
    int ir = 0, ic = -1;    // initial position
    while (nSteps[iDir%2]) {
        for (int i = 0; i < nSteps[iDir%2]; ++i) {
            ir += dirs[iDir][0]; ic += dirs[iDir][1];
            res.push_back(matrix[ir][ic]);
        }
        nSteps[iDir%2]--;
        iDir = (iDir + 1) % 4;
    }
    return res;
}
```

written by stellari original link here

Answer 3

```
    public List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> spiralList = new ArrayList<>();
    if(matrix == null || matrix.length == 0) return spiralList;

    // declare indices
    int top = 0;
    int bottom = matrix.length - 1;
    int left = 0;
    int right = matrix[0].length - 1;

    while(true){
        // 1. print top row
        for(int j=left; j <=right;j++){
            spiralList.add(matrix[top][j]);
        }
        top++;
        if(boundriesCrossed(left,right,bottom,top))
            break;

        // 2. print rightmost column
        for(int i=top; i <= bottom; i++){
            spiralList.add(matrix[i][right]);
        }
        right--;
        if(boundriesCrossed(left,right,bottom,top))
            break;

        // 3. print bottom row
        for(int j=right; j >=left; j--){
            spiralList.add(matrix[bottom][j]);
        }
        bottom--;
        if(boundriesCrossed(left,right,bottom,top))
            break;

        // 4. print leftmost column
        for(int i=bottom; i >= top; i--){
            spiralList.add(matrix[i][left]);
        }
        left++;
        if(boundriesCrossed(left,right,bottom,top))
            break;
    }// end while true

    return spiralList;
}

private boolean boundriesCrossed(int left,int right,int bottom,int top){
    if(left>right || bottom<top)
        return true;
    else
        return false;
}
```

written by ishanpande original link here

Jump Game(55)

Answer 1

I just iterate and update the maximal index that I can reach

```
bool canJump(int A[], int n) {
    int i = 0;
    for (int reach = 0; i < n && i <= reach; ++i)
        reach = max(i + A[i], reach);
    return i == n;
}
```

written by alexander7 original link here

Answer 2

Idea is to work backwards from the last index. Keep track of the smallest index that can "jump" to the last index. Check whether the current index can jump to this smallest index.

```
bool canJump(int A[], int n) {
    int last=n-1,i,j;
    for(i=n-2;i>=0;i--){
        if(i+A[i]>=last)last=i;
    }
    return last<=0;
}
```

written by lucastan original link here

Answer 3

```
public boolean canJump(int[] A) {
    int max = 0;
    for(int i=0;i<A.length;i++){
        if(i>max) {return false;}
        max = Math.max(A[i]+i,max);
    }
    return true;
}
```

written by xniu original link here

Merge Intervals(56)

Answer 1

The idea is to sort the intervals by their starting points. Then, we take the first interval and compare its end with the next intervals starts. As long as they overlap, we update the end to be the max end of the overlapping intervals. Once we find a non overlapping interval, we can add the previous "extended" interval and start over.

Sorting takes O(n log(n)) and merging the intervals takes O(n). So, the resulting algorithm takes O(n log(n)).

I used an anonymous comparator and a for-each loop to try to keep the code clean and simple.

```
public List<Interval> merge(List<Interval> intervals) {
    if (intervals.size() <= 1)
        return intervals;

    // Sort by ascending starting point using an anonymous Comparator
    Collections.sort(intervals, new Comparator<Interval>() {
        @Override
        public int compare(Interval i1, Interval i2) {
            return Integer.compare(i1.start, i2.start);
        }
    });

    List<Interval> result = new LinkedList<Interval>();
    int start = intervals.get(0).start;
    int end = intervals.get(0).end;

    for (Interval interval : intervals) {
        if (interval.start <= end) // Overlapping intervals, move the end if need
ed
            end = Math.max(end, interval.end);
        else {                     // Disjoint intervals, add the previous one an
d reset bounds
            result.add(new Interval(start, end));
            start = interval.start;
            end = interval.end;
        }
    }

    // Add the last interval
    result.add(new Interval(start, end));
    return result;
}
```

written by brubru777 original link here

Answer 2

```java
public class Solution {
    public List<Interval> merge(List<Interval> intervals) {
        Collections.sort(intervals, new Comparator<Interval>(){
            @Override
            public int compare(Interval obj0, Interval obj1) {
                return obj0.start - obj1.start;
            }
        });

        List<Interval> ret = new ArrayList<>();
        Interval prev = null;
        for (Interval inter : intervals) {
            if (  prev==null || inter.start>prev.end ) {
                ret.add(inter);
                prev = inter;
            } else if (inter.end>prev.end) {
                // Modify the element already in list
                prev.end = inter.end;
            }
        }
        return ret;
    }
}
```

written by danchou original link here

Answer 3

Just go through the intervals sorted by start coordinate and either combine the current interval with the previous one if they overlap, or add it to the output by itself if they don't.

```python
def merge(self, intervals):
    out = []
    for i in sorted(intervals, key=lambda i: i.start):
        if out and i.start <= out[-1].end:
            out[-1].end = max(out[-1].end, i.end)
        else:
            out += i,
    return out
```

written by StefanPochmann original link here

Insert Interval(57)

Answer 1

Hi guys!

Here's a pretty straight-forward and concise solution below.

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> result = new LinkedList<>();
    int i = 0;
    // add all the intervals ending before newInterval starts
    while (i < intervals.size() && intervals.get(i).end < newInterval.start)
        result.add(intervals.get(i++));
    // merge all overlapping intervals to one considering newInterval
    while (i < intervals.size() && intervals.get(i).start <= newInterval.end) {
        newInterval = new Interval( // we could mutate newInterval here also
                Math.min(newInterval.start, intervals.get(i).start),
                Math.max(newInterval.end, intervals.get(i).end));
        i++;
    }
    result.add(newInterval); // add the union of intervals we got
    // add all the rest
    while (i < intervals.size()) result.add(intervals.get(i++));
    return result;
}
```

Hope it helps.

written by shpolsky original link here

Answer 2

Solution 1: (7 lines, 88 ms)

Collect the intervals strictly left or right of the new interval, then merge the new one with the middle ones (if any) before inserting it between left and right ones.

```
def insert(self, intervals, newInterval):
    s, e = newInterval.start, newInterval.end
    left = [i for i in intervals if i.end < s]
    right = [i for i in intervals if i.start > e]
    if left + right != intervals:
        s = min(s, intervals[len(left)].start)
        e = max(e, intervals[~len(right)].end)
    return left + [Interval(s, e)] + right
```

Solution 2: (8 lines, 84 ms)

Same algorithm as solution 1, but different implementation with only one pass and explicitly collecting the to-be-merged intervals.

```
def insert(self, intervals, newInterval):
    s, e = newInterval.start, newInterval.end
    parts = merge, left, right = [], [], []
    for i in intervals:
        parts[(i.end < s) - (i.start > e)].append(i)
    if merge:
        s = min(s, merge[0].start)
        e = max(e, merge[-1].end)
    return left + [Interval(s, e)] + right
```

Solution 3: (11 lines, 80 ms)

Same again, but collect and merge while going over the intervals once.

```
def insert(self, intervals, newInterval):
    s, e = newInterval.start, newInterval.end
    left, right = [], []
    for i in intervals:
        if i.end < s:
            left += i,
        elif i.start > e:
            right += i,
        else:
            s = min(s, i.start)
            e = max(e, i.end)
    return left + [Interval(s, e)] + right
```

written by StefanPochmann original link here

Answer 3

```
vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
    vector<Interval> ret;
    auto it = intervals.begin();
    for(; it!=intervals.end(); ++it){
        if(newInterval.end < (*it).start) //all intervals after will not overlap
with the newInterval
            break;
        else if(newInterval.start > (*it).end) //*it will not overlap with the ne
wInterval
            ret.push_back(*it);
        else{ //update newInterval bacause *it overlap with the newInterval
            newInterval.start = min(newInterval.start, (*it).start);
            newInterval.end = max(newInterval.end, (*it).end);
        }
    }
    // don't forget the rest of the intervals and the newInterval
    ret.push_back(newInterval);
    for(; it!=intervals.end(); ++it)
        ret.push_back(*it);
    return ret;
}
```

My question is why this code need 500ms !?

written by Erudy original link here

Length of Last Word(58)

Answer 1

I've noticed that a lot of solutions use available library functions that return directly the positions of certain characters or do other operations like "split". I personally don't think that's a good idea. Firstly, these functions take some time and usually involve with iteration through the whole string. Secondly, questions like this one is intended to be a practice of detail implementation, not calling other functions. My solution like below uses only the most basic string operations and probably beats many other solutions which call other existing functions.

```
int lengthOfLastWord(const char* s) {
        int len = 0;
        while (*s) {
            if (*s++ != ' ')
                ++len;
            else if (*s && *s != ' ')
                len = 0;

        }
        return len;
    }
```

written by eaglesky1990 original link here

Answer 2

```
public int lengthOfLastWord(String s) {
    return s.trim().length()-s.trim().lastIndexOf(" ")-1;
}
```

written by lvlolitte original link here

Answer 3

Well, the basic idea is very simple. Start from the tail of `s` and move backwards to find the first non-space character. Then from this character, move backwards and count the number of non-space characters until we pass over the head of `s` or meet a space character. The count will then be the length of the last word.

```
class Solution {
public:
    int lengthOfLastWord(string s) {
        int len = 0, tail = s.length() - 1;
        while (tail >= 0 && s[tail] == ' ') tail--;
        while (tail >= 0 && s[tail] != ' ') {
            len++;
            tail--;
        }
        return len;
    }
};
```

written by jianchao.li.fighter original link here

Spiral Matrix II(59)

Answer 1

Solution 1: *Build it inside-out* - 44 ms, 5 lines

Start with the empty matrix, add the numbers in reverse order until we added the number 1. Always rotate the matrix clockwise and add a top row:

```
    ||  =>  |9|  =>  |8|      |6 7|      |4 5|      |1 2 3|
                    |9|  =>  |9 8|  =>  |9 6|  =>  |8 9 4|
                                        |8 7|      |7 6 5|
```

The code:

```
def generateMatrix(self, n):
    A, lo = [], n*n+1
    while lo > 1:
        lo, hi = lo - len(A), lo
        A = [range(lo, hi)] + zip(*A[::-1])
    return A
```

While this isn't O(n^2), it's actually quite fast, presumably due to me not doing much in Python but relying on `zip` and `range` and `+` being fast. I got it accepted in 44 ms, matching the fastest time for recent Python submissions (according to the submission detail page).

Solution 2: *Ugly inside-out* - 48 ms, 4 lines

Same as solution 1, but without helper variables. Saves a line, but makes it ugly. Also, because I access A[0][0], I had to handle the n=0 case differently.

```
def generateMatrix(self, n):
    A = [[n*n]]
    while A[0][0] > 1:
        A = [range(A[0][0] - len(A), A[0][0])] + zip(*A[::-1])
    return A * (n>0)
```

Solution 3: *Walk the spiral* - 52 ms, 9 lines

Initialize the matrix with zeros, then walk the spiral path and write the numbers 1 to n*n. Make a right turn when the cell ahead is already non-zero.

```
def generateMatrix(self, n):
    A = [[0] * n for _ in range(n)]
    i, j, di, dj = 0, 0, 0, 1
    for k in xrange(n*n):
        A[i][j] = k + 1
        if A[(i+di)%n][(j+dj)%n]:
            di, dj = dj, -di
        i += di
        j += dj
    return A
```

written by StefanPochmann original link here

Answer 2

```
class Solution {
    public:
        vector<vector<int> > generateMatrix(int n) {
            vector<vector<int> > ret( n, vector<int>(n) );
            int k = 1, i = 0;
            while( k <= n * n )
            {
                int j = i;
                    // four steps
                while( j < n - i )              // 1. horizonal, left to right
                    ret[i][j++] = k++;
                j = i + 1;
                while( j < n - i )              // 2. vertical, top to bottom
                    ret[j++][n-i-1] = k++;
                j = n - i - 2;
                while( j > i )                   // 3. horizonal, right to left
                    ret[n-i-1][j--] = k++;
                j = n - i - 1;
                while( j > i )                   // 4. vertical, bottom to  top
                    ret[j--][i] = k++;
                i++;      // next loop
            }
            return ret;
        }
    };
```

written by AllenYick original link here

Answer 3

This is my solution for Spiral Matrix I, https://oj.leetcode.com/discuss/12228/super-simple-and-easy-to-understand-solution. If you can understand that, this one is a no brainer :)

Guess what? I just made several lines of change (with comment "//change") from that and I have the following AC code:

```java
public class Solution {
    public int[][] generateMatrix(int n) {
        // Declaration
        int[][] matrix = new int[n][n];

        // Edge Case
        if (n == 0) {
            return matrix;
        }

        // Normal Case
        int rowStart = 0;
        int rowEnd = n-1;
        int colStart = 0;
        int colEnd = n-1;
        int num = 1; //change

        while (rowStart <= rowEnd && colStart <= colEnd) {
            for (int i = colStart; i <= colEnd; i ++) {
                matrix[rowStart][i] = num ++; //change
            }
            rowStart ++;

            for (int i = rowStart; i <= rowEnd; i ++) {
                matrix[i][colEnd] = num ++; //change
            }
            colEnd --;

            for (int i = colEnd; i >= colStart; i --) {
                if (rowStart <= rowEnd)
                    matrix[rowEnd][i] = num ++; //change
            }
            rowEnd --;

            for (int i = rowEnd; i >= rowStart; i --) {
                if (colStart <= colEnd)
                    matrix[i][colStart] = num ++; //change
            }
            colStart ++;
        }

        return matrix;
    }
}
```

Obviously, you could merge colStart and colEnd into rowStart and rowEnd because it is a square matrix. But this is easily extensible to matrices that are m*n.

Hope this helps :)

written by qwl5004 original link here

Permutation Sequence(60)

Answer 1

I'm sure somewhere can be simplified so it'd be nice if anyone can let me know. The pattern was that:

say n = 4, you have {1, 2, 3, 4}

If you were to list out all the permutations you have

1 + (permutations of 2, 3, 4)
2 + (permutations of 1, 3, 4)
3 + (permutations of 1, 2, 4)
4 + (permutations of 1, 2, 3)

We know how to calculate the number of permutations of n numbers... n! So each of those with permutations of 3 numbers means there are 6 possible permutations. Meaning there would be a total of 16 permutations in this particular one. So if you were to look for the (k = 14) 14th permutation, it would be in the

3 + (permutations of 1, 2, 4) subset.

To programmatically get that, you take k = 13 (subtract 1 because of things always starting at 0) and divide that by the 6 we got from the factorial, which would give you the index of the number you want. In the array {1, 2, 3, 4}, k/(n-1)! = 13/(4-1)! = 13/3! = 13/6 = 2. The array {1, 2, 3, 4} has a value of 3 at index 2. So the first number is a 3.

Then the problem repeats with less numbers.

The permutations of {1, 2, 4} would be:

1 + (permutations of 2, 4)
2 + (permutations of 1, 4)
4 + (permutations of 1, 2)

But our k is no longer the 14th, because in the previous step, we've already eliminated the 12 4-number permutations starting with 1 and 2. So you subtract 12 from k.. which gives you 1. Programmatically that would be...

k = k - (index from previous) * (n-1)! = k - 2*(n-1)! = 13* - 2(3)! = 1

In this second step, permutations of 2 numbers has only 2 possibilities, meaning each of the three permutations listed above a has two possibilities, giving a total of 6. We're looking for the first one, so that would be in the 1 + (permutations of 2, 4) subset.

Meaning: index to get number from is k / (n - 2)! = 1 / (4-2)! = 1 / 2! = 0.. from {1, 2, 4}, index 0 is 1

so the numbers we have so far is 3, 1... and then repeating without explanations.

{2, 4}
k = k - (index from pervious) * (n-2)! = k - 0 * (n - 2)! = 1 - 0 = 1;
third number's index = k / (n - 3)! = 1 / (4-3)! = 1/ 1! = 1... from {2, 4}, index 1 has 4
Third number is 4

{2}
k = k - (index from pervious) * (n - 3)! = k - 1 * (4 - 3)! = 1 - 1 = 0;
third number's index = k / (n - 4)! = 0 / (4-4)! = 0/ 1 = 0... from {2}, index 0 has 2
Fourth number is 2

Giving us 3142. If you manually list out the permutations using DFS method, it would be 3142. Done! It really was all about pattern finding.

```
public class Solution {
public String getPermutation(int n, int k) {
    int pos = 0;
    List<Integer> numbers = new ArrayList<>();
    int[] factorial = new int[n+1];
    StringBuilder sb = new StringBuilder();

    // create an array of factorial lookup
    int sum = 1;
    factorial[0] = 1;
    for(int i=1; i<=n; i++){
        sum *= i;
        factorial[i] = sum;
    }
    // factorial[] = {1, 1, 2, 6, 24, ... n!}

    // create a list of numbers to get indices
    for(int i=1; i<=n; i++){
        numbers.add(i);
    }
    // numbers = {1, 2, 3, 4}

    k--;

    for(int i = 1; i <= n; i++){
        int index = k/factorial[n-i];
        sb.append(String.valueOf(numbers.get(index)));
        numbers.remove(index);
        k-=index*factorial[n-i];
    }

    return String.valueOf(sb);
}
```

}

written by tso original link here

Answer 2

```
string getPermutation(int n, int k) {
    int i,j,f=1;
    // left part of s is partially formed permutation, right part is the leftover
chars.
    string s(n,'0');
    for(i=1;i<=n;i++){
        f*=i;
        s[i-1]+=i; // make s become 1234...n
    }
    for(i=0,k--;i<n;i++){
        f/=n-i;
        j=i+k/f; // calculate index of char to put at s[i]
        char c=s[j];
        // remove c by shifting to cover up (adjust the right part).
        for(;j>i;j--)
            s[j]=s[j-1];
        k%=f;
        s[i]=c;
    }
    return s;
}
```

written by lucastan original link here

Answer 3

Recursion will use more memory, while this problem can be solved by iteration. I solved this problem before, but I didn't realize that using k = k-1 would avoid dealing with case k%(n-1)!==0. Rewrote this code, should be pretty concise now.

Only thing is that I have to use a list to store the remaining numbers, neither linkedlist nor arraylist are very efficient, anyone has a better idea?

The logic is as follows: for n numbers the permutations can be divided to (n-1)! groups, thus k/(n-1)! indicates the index of current number, and k%(n-1)! denotes remaining sequence (to the right). We keep doing this until n reaches 0, then we get n numbers permutations that is kth.

```java
public String getPermutation(int n, int k) {
        List<Integer> num = new LinkedList<Integer>();
        for (int i = 1; i <= n; i++) num.add(i);
        int[] fact = new int[n];  // factorial
        fact[0] = 1;
        for (int i = 1; i < n; i++) fact[i] = i*fact[i-1];
        k = k-1;
        StringBuilder sb = new StringBuilder();
        for (int i = n; i > 0; i--){
            int ind = k/fact[i-1];
            k = k%fact[i-1];
            sb.append(num.get(ind));
            num.remove(ind);
        }
        return sb.toString();
    }
```

written by Adeath original link here

Rotate List(61)

Answer 1

There is no trick for this problem. Some people used slow/fast pointers to find the tail node but I don't see the benefit (in the sense that it doesn't reduce the pointer move op) to do so. So I just used one loop to find the length first.

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(!head) return head;

        int len=1; // number of nodes
        ListNode *newH, *tail;
        newH=tail=head;

        while(tail->next)  // get the number of nodes in the list
        {
            tail = tail->next;
            len++;
        }
        tail->next = head; // circle the link

        if(k %= len)
        {
            for(auto i=0; i<len-k; i++) tail = tail->next; // the tail node is th
e (len-k)-th node (1st node is head)
        }
        newH = tail->next;
        tail->next = NULL;
        return newH;
    }
};
```

written by dong.wang.1694 original link here

Answer 2

Since n may be a large number compared to the length of list. So we need to know the length of linked list.After that, move the list after the (l-n%l)th node to the front to finish the rotation.

Ex: {1,2,3} k=2 Move the list after the 1st node to the front

Ex: {1,2,3} k=5, In this case Move the list after (3-5%3=1)st node to the front.

So the code has three parts.

1) Get the length

2) Move to the (l-n%l)th node

3)Do the rotation

```
public ListNode rotateRight(ListNode head, int n) {
    if (head==null||head.next==null) return head;
    ListNode dummy=new ListNode(0);
    dummy.next=head;
    ListNode fast=dummy,slow=dummy;

    int i;
    for (i=0;fast.next!=null;i++)//Get the total length
        fast=fast.next;

    for (int j=i-n%i;j>0;j--) //Get the i-n%i th node
        slow=slow.next;

    fast.next=dummy.next; //Do the rotation
    dummy.next=slow.next;
    slow.next=null;

    return dummy.next;
}
```

written by reeclapple original link here

Answer 3

i am not getting that what i should do when K is greater than size of the list.

written by rforritz original link here

Unique Paths(62)

Answer 1

Binomial coefficient:

```
class Solution {
    public:
        int uniquePaths(int m, int n) {
            int N = n + m - 2;// how much steps we need to do
            int k = m - 1; // number of steps that need to go down
            double res = 1;
            // here we calculate the total possible path number
            // Combination(N, k) = n! / (k!(n - k)!)
            // reduce the numerator and denominator and get
            // C = ( (n - k + 1) * (n - k + 2) * ... * n ) / k!
            for (int i = 1; i <= k; i++)
                res = res * (N - k + i) / i;
            return (int)res;
        }
    };
```

First of all you should understand that we need to do n + m - 2 movements : m - 1 down, n - 1 right, because we start from cell (1, 1).

Secondly, the path it is the sequence of movements(go down / go right), therefore we can say that two paths are different when there is i-th (1 .. m + n - 2) movement in path1 differ i-th movement in path2.

So, how we can build paths. Let's choose (n - 1) movements(number of steps to the right) from (m + n - 2), and rest (m - 1) is (number of steps down).

I think now it is obvious that count of different paths are all combinations (n - 1) movements from (m + n-2).

written by d40a original link here

Answer 2

This is a fundamental DP problem. First of all, let's make some observations.

Since the robot can only move right and down, when it arrives at a point, there are only two possibilities:

1. It arrives at that point from above (moving down to that point);
2. It arrives at that point from left (moving right to that point).

Thus, we have the following state equations: suppose the number of paths to arrive at a point `(i, j)` is denoted as `P[i][j]`, it is easily concluded that `P[i][j] = P[i - 1][j] + P[i][j - 1]`.

The boundary conditions of the above equation occur at the leftmost column (`P[i][j - 1]` does not exist) and the uppermost row (`P[i - 1][j]` does not exist). These conditions can be handled by initialization (pre-processing) --- initialize `P[0]`

`[j] = 1, P[i][0] = 1` for all valid `i, j`. Note the initial value is `1` instead of `0`!

Now we can write down the following (unoptimized) code.

```
class Solution {
    int uniquePaths(int m, int n) {
        vector<vector<int> > path(m, vector<int> (n, 1));
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                path[i][j] = path[i - 1][j] + path[i][j - 1];
        return path[m - 1][n - 1];
    }
};
```

As can be seen, the above solution runs in `O(n^2)` time and costs `O(m*n)` space. However, you may have observed that each time when we update `path[i][j]`, we only need `path[i - 1][j]` (at the same column) and `path[i][j - 1]` (at the left column). So it is enough to maintain two columns (the current column and the left column) instead of maintaining the full `m*n` matrix. Now the code can be optimized to have `O(min(m, n))` space complexity.

```
class Solution {
    int uniquePaths(int m, int n) {
        if (m > n) return uniquePaths(n, m);
        vector<int> pre(m, 1);
        vector<int> cur(m, 1);
        for (int j = 1; j < n; j++) {
            for (int i = 1; i < m; i++)
                cur[i] = cur[i - 1] + pre[i];
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, we find that keeping two columns is used to recover `pre[i]`, which is just `cur[i]` before its update. So there is even no need to use two vectors and one is just enough. Now the space is further saved and the code also gets much shorter.

```
class Solution {
    int uniquePaths(int m, int n) {
        if (m > n) return uniquePaths(n, m);
        vector<int> cur(m, 1);
        for (int j = 1; j < n; j++)
            for (int i = 1; i < m; i++)
                cur[i] += cur[i - 1];
        return cur[m - 1];
    }
};
```

Well, till now, I guess you may even want to optimize it to `O(1)` space complexity since the above code seems to rely on only `cur[i]` and `cur[i - 1]`. You may think that 2 variables is enough? Well, it is not. Since the whole `cur` needs to be updated for `n - 1` times, it means that all of its values need to be saved for next update and so two variables is not enough.

written by jianchao.li.fighter original link here

Answer 3

This is a combinatorial problem and can be solved without DP. For mxn grid, robot has to move exactly m-1 steps down and n-1 steps right and these can be done in any order.

For the eg., given in question, 3x7 matrix, robot needs to take 2+6 = 8 steps with 2 down and 6 right in any order. That is nothing but a permutation problem. Denote down as 'D' and right as 'R', following is one of the path :-

D R R R D R R R

We have to tell the total number of permutations of the above given word. So, decrease both m & n by 1 and apply following formula:-

Total permutations = (m+n)! / (m! * n!)

Following is my code doing the same :-

```
public class Solution {
    public int uniquePaths(int m, int n) {
        if(m == 1 || n == 1)
            return 1;
        m--;
        n--;
        if(m < n) {              // Swap, so that m is the bigger number
            m = m + n;
            n = m - n;
            m = m - n;
        }
        long res = 1;
        int j = 1;
        for(int i = m+1; i <= m+n; i++, j++){       // Instead of taking factoria
l, keep on multiply & divide
            res *= i;
            res /= j;
        }

        return (int)res;
    }
}
```

written by whitehat original link here

Unique Paths II(63)

Answer 1

just use dp to find the answer , if there is a obstacle at (i,j), then dp[i][j] = 0. time is O(n*m) , space is O(n*m) . here is my code:

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        int m = obstacleGrid.size() , n = obstacleGrid[0].size();
        vector<vector<int>> dp(m+1,vector<int>(n+1,0));
        dp[0][1] = 1;
        for(int i = 1 ; i <= m ; ++i)
            for(int j = 1 ; j <= n ; ++j)
                if(!obstacleGrid[i-1][j-1])
                    dp[i][j] = dp[i-1][j]+dp[i][j-1];
        return dp[m][n];
    }
};
```

written by kingmacrobo original link here

Answer 2

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int width = obstacleGrid[0].length;
    int[] dp = new int[width];
    dp[0] = 1;
    for (int[] row : obstacleGrid) {
        for (int j = 0; j < width; j++) {
            if (row[j] == 1)
                dp[j] = 0;
            else if (j > 0)
                dp[j] += dp[j - 1];
        }
    }
    return dp[width - 1];
}
```

written by tusizi original link here

Answer 3

```java
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        //Empty case
        if(obstacleGrid.length == 0) return 0;

        int rows = obstacleGrid.length;
        int cols = obstacleGrid[0].length;

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(obstacleGrid[i][j] == 1)
                    obstacleGrid[i][j] = 0;
                else if(i == 0 && j == 0)
                    obstacleGrid[i][j] = 1;
                else if(i == 0)
                    obstacleGrid[i][j] = obstacleGrid[i][j - 1] * 1;// For row 0,
if there are no paths to left cell, then its 0,else 1
                else if(j == 0)
                    obstacleGrid[i][j] = obstacleGrid[i - 1][j] * 1;// For col 0,
if there are no paths to upper cell, then its 0,else 1
                else
                    obstacleGrid[i][j] = obstacleGrid[i - 1][j] + obstacleGrid[i]
[j - 1];
            }
        }

        return obstacleGrid[rows - 1][cols - 1];

    }
}
```

written by vvelrath%40buffalo.edu original link here

Minimum Path Sum(64)

Answer 1

This is a typical DP problem. Suppose the minimum path sum of arriving at point `(i, j)` is `S[i][j]`, then the state equation is `S[i][j] = min(S[i - 1][j], S[i][j - 1]) + grid[i][j]`.

Well, some boundary conditions need to be handled. The boundary conditions happen on the topmost row (`S[i - 1][j]` does not exist) and the leftmost column (`S[i][j - 1]` does not exist). Suppose `grid` is like `[1, 1, 1, 1]`, then the minimum sum to arrive at each point is simply an accumulation of previous points and the result is `[1, 2, 3, 4]`.

Now we can write down the following (unoptimized) code.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<vector<int> > sum(m, vector<int>(n, grid[0][0]));
        for (int i = 1; i < m; i++)
            sum[i][0] = sum[i - 1][0] + grid[i][0];
        for (int j = 1; j < n; j++)
            sum[0][j] = sum[0][j - 1] + grid[0][j];
        for (int i = 1; i < m; i++)
            for (int j = 1; j < n; j++)
                sum[i][j]  = min(sum[i - 1][j], sum[i][j - 1]) + grid[i][j];
        return sum[m - 1][n - 1];
    }
};
```

As can be seen, each time when we update `sum[i][j]`, we only need `sum[i - 1][j]` (at the current column) and `sum[i][j - 1]` (at the left column). So we need not maintain the full `m*n` matrix. Maintaining two columns is enough and now we have the following code.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<int> pre(m, grid[0][0]);
        vector<int> cur(m, 0);
        for (int i = 1; i < m; i++)
            pre[i] = pre[i - 1] + grid[i][0];
        for (int j = 1; j < n; j++) {
            cur[0] = pre[0] + grid[0][j];
            for (int i = 1; i < m; i++)
                cur[i] = min(cur[i - 1], pre[i]) + grid[i][j];
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, it can be seen that maintaining `pre` is for recovering `pre[i]`, which is simply `cur[i]` before its update. So it is enough to use only one vector. Now the space is further optimized and the code also gets shorter.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        vector<int> cur(m, grid[0][0]);
        for (int i = 1; i < m; i++)
            cur[i] = cur[i - 1] + grid[i][0];
        for (int j = 1; j < n; j++) {
            cur[0] += grid[0][j];
            for (int i = 1; i < m; i++)
                cur[i] = min(cur[i - 1], cur[i]) + grid[i][j];
        }
        return cur[m - 1];
    }
};
```

written by jianchao.li.fighter original link here

Answer 2

```java
public int minPathSum(int[][] grid) {
    int m = grid.length;// row
    int n = grid[0].length; // column
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 && j != 0) {
                grid[i][j] = grid[i][j] + grid[i][j - 1];
            } else if (i != 0 && j == 0) {
                grid[i][j] = grid[i][j] + grid[i - 1][j];
            } else if (i == 0 && j == 0) {
                grid[i][j] = grid[i][j];
            } else {
                grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j])
                        + grid[i][j];
            }
        }
    }

    return grid[m - 1][n - 1];
}
```

written by wdjoxda original link here

Answer 3

You can only reach a cell by going from its left or top neighbor.

```cpp
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if(!grid.size())return 0;
        const int rows=grid.size(),cols=grid[0].size();
        // r[i] == min path sum to previous row's column i.
        vector<int> r(cols,0);
        int i,j;
        r[0]=grid[0][0];
        for(j=1;j<cols;j++){
            r[j]=grid[0][j]+r[j-1];
        }
        for(i=1;i<rows;i++){
            r[0]+=grid[i][0];
            for(j=1;j<cols;j++){
                r[j]=min(r[j-1],r[j])+grid[i][j];
            }
        }
        return r[cols-1];
    }
};
```

written by lucastan original link here

Valid Number(65)

Answer 1

The description do not give a clear explantion of the definition of a valid Number, we just use more and more trick to get the right solution. It's too bad, it's waste of my time

written by aqin original link here

Answer 2

The idea is pretty straightforward. A valid number is composed of the significand and the exponent (which is optional). As we go through the string, do the following things one by one:

1. skip the leading whitespaces;
2. check if the significand is valid. To do so, simply skip the leading sign and count the number of digits and the number of points. A valid significand has no more than one point and at least one digit.
3. check if the exponent part is valid. We do this if the significand is followed by 'e'. Simply skip the leading sign and count the number of digits. A valid exponent contain at least one digit.
4. skip the trailing whitespaces. We must reach the ending 0 if the string is a valid number.

==========

```
bool isNumber(const char *s)
{
    int i = 0;

    // skip the whilespaces
    for(; s[i] == ' '; i++) {}

    // check the significand
    if(s[i] == '+' || s[i] == '-') i++; // skip the sign if exist

    int n_nm, n_pt;
    for(n_nm=0, n_pt=0; (s[i]<='9' && s[i]>='0') || s[i]=='.'; i++)
        s[i] == '.' ? n_pt++:n_nm++;
    if(n_pt>1 || n_nm<1) // no more than one point, at least one digit
        return false;

    // check the exponent if exist
    if(s[i] == 'e') {
        i++;
        if(s[i] == '+' || s[i] == '-') i++; // skip the sign

        int n_nm = 0;
        for(; s[i]>='0' && s[i]<='9'; i++, n_nm++) {}
        if(n_nm<1)
            return false;
    }

    // skip the trailing whitespaces
    for(; s[i] == ' '; i++) {}

    return s[i]==0;  // must reach the ending 0 of the string
}
```

written by GuaGua original link here

Answer 3

All we need is to have a couple of flags so we can process the string in linear time:

```java
public boolean isNumber(String s) {
    s = s.trim();

    boolean pointSeen = false;
    boolean eSeen = false;
    boolean numberSeen = false;
    boolean numberAfterE = true;
    for(int i=0; i<s.length(); i++) {
        if('0' <= s.charAt(i) && s.charAt(i) <= '9') {
            numberSeen = true;
            numberAfterE = true;
        } else if(s.charAt(i) == '.') {
            if(eSeen || pointSeen) {
                return false;
            }
            pointSeen = true;
        } else if(s.charAt(i) == 'e') {
            if(eSeen || !numberSeen) {
                return false;
            }
            numberAfterE = false;
            eSeen = true;
        } else if(s.charAt(i) == '-' || s.charAt(i) == '+') {
            if(i != 0 && s.charAt(i-1) != 'e') {
                return false;
            }
        } else {
            return false;
        }
    }

    return numberSeen && numberAfterE;
}
```

We start with trimming.

- If we see `[0-9]` we reset the number flags.
- We can only see `.` if we didn't see `e` or `.`.
- We can only see `e` if we didn't see `e` but we did see a number. We reset numberAfterE flag.
- We can only see `+` and `-` in the beginning and after an `e`
- any other character break the validation.

At the and it is only valid if there was at least 1 number and if we did see an `e` then a number after it as well.

So basically the number should match this regular expression:

`[-+]?[0-9]*(.[0-9]+)?(e[-+]?[0-9]+)?`

written by balint original link here

Plus One(66)

Answer 1

```
void plusone(vector<int> &digits)
{
    int n = digits.size();
    for (int i = n - 1; i >= 0; --i)
    {
        if (digits[i] == 9)
        {
            digits[i] = 0;
        }
        else
        {
            digits[i]++;
            return;
        }
    }
        digits[0] =1;
        digits.push_back(0);

}
```

written by zezedi original link here

Answer 2

```
public int[] plusOne(int[] digits) {

    int n = digits.length;
    for(int i=n-1; i>=0; i--) {
        if(digits[i] < 9) {
            digits[i]++;
            return digits;
        }

        digits[i] = 0;
    }

    int[] newNumber = new int [n+1];
    newNumber[0] = 1;

    return newNumber;
}
```

written by diaa original link here

Answer 3

```java
public int[] plusOne(int[] digits) {
    for (int i = digits.length - 1; i >=0; i--) {
        if (digits[i] != 9) {
            digits[i]++;
            break;
        } else {
            digits[i] = 0;
        }
    }
    if (digits[0] == 0) {
        int[] res = new int[digits.length+1];
        res[0] = 1;
        return res;
    }
    return digits;
}
```

written by hello_today_ original link here

Add Binary(67)

Answer 1

```
class Solution
{
public:
    string addBinary(string a, string b)
    {
        string s = "";

        int c = 0, i = a.size() - 1, j = b.size() - 1;
        while(i >= 0 || j >= 0 || c == 1)
        {
            c += i >= 0 ? a[i --] - '0' : 0;
            c += j >= 0 ? b[j --] - '0' : 0;
            s = char(c % 2 + '0') + s;
            c /= 2;
        }

        return s;
    }
};
```

written by makuiyu original link here

Answer 2

```java
public class Solution {
    public String addBinary(String a, String b) {
        if(a == null || a.isEmpty()) {
            return b;
        }
        if(b == null || b.isEmpty()) {
            return a;
        }
        char[] aArray = a.toCharArray();
        char[] bArray = b.toCharArray();
        StringBuilder stb = new StringBuilder();

        int i = aArray.length - 1;
        int j = bArray.length - 1;
        int aByte;
        int bByte;
        int carry = 0;
        int result;

        while(i > -1 || j > -1 || carry == 1) {
            aByte = (i > -1) ? Character.getNumericValue(aArray[i--]) : 0;
            bByte = (j > -1) ? Character.getNumericValue(bArray[j--]) : 0;
            result = aByte ^ bByte ^ carry;
            carry = ((aByte + bByte + carry) >= 2) ? 1 : 0;
            stb.append(result);
        }
        return stb.reverse().toString();
    }
}
```

Addition bits are calculated by xor. Carry bit is calculated as simple integer addition.

written by markivr original link here

Answer 3

```java
public class Solution {
    public String addBinary(String a, String b) {
        StringBuilder sb = new StringBuilder();
        int i = a.length() - 1, j = b.length() -1, carry = 0;
        while (i >= 0 || j >= 0) {
            int sum = carry;
            if (j >= 0) sum += b.charAt(j--) - '0';
            if (i >= 0) sum += a.charAt(i--) - '0';
            sb.append(sum % 2);
            carry = sum / 2;
        }
        if (carry != 0) sb.append(carry);
        return sb.reverse().toString();
    }
}
```

Computation from string usually can be simplified by using a carry as such.

written by lx223 original link here

Text Justification(68)

Answer 1

```
vector<string> fullJustify(vector<string> &words, int L) {
    vector<string> res;
    for(int i = 0, k, l; i < words.size(); i += k) {
        for(k = l = 0; i + k < words.size() and l + words[i+k].size() <= L - k; k
++) {
            l += words[i+k].size();
        }
        string tmp = words[i];
        for(int j = 0; j < k - 1; j++) {
            if(i + k >= words.size()) tmp += " ";
            else tmp += string((L - l) / (k - 1) + (j < (L - l) % (k - 1)), ' ');
            tmp += words[i+j+1];
        }
        tmp += string(L - tmp.size(), ' ');
        res.push_back(tmp);
    }
    return res;
}
```

For each line, I first figure out which words can fit in. According to the code, these words are words[i] through words[i+k-1]. Then spaces are added between the words. The trick here is to use mod operation to manage the spaces that can't be evenly distrubuted: the first (L-l) % (k-1) gaps acquire an additional space.

written by qddpx original link here

Answer 2

In some of the texts that I have been able to find I see that this problem admits a dynamic programming solution that is superior to greedy solutions. (MSWord vs LATEX). I think, that to solve this question specifically (meaning something that OJ accepts) requires a greedy solution.

As far as I understand the "idea" of text justification is not to distribute spaces as evenly as possible within all the words of an individual line; But instead lower the overall cost of the way you justify text, which means that even though you may have some lines that have uneven spaces between words than others, but this lowers the overall cost of a justification in other lines.

In this question's description the correct answer is described as a very specific way to do text justification that seems to be not what the superior solution is.

Do you think its right to actually post this question as an exercise at all? What does this question aim to teach as far as good text justification algorithms are concerned?

written by graham2181 original link here

Answer 3

Input: [""], 2

Output: [""]
Expected: [" "]
As described ,the last line of text should be left justified and no extra space is inserted between words.Can anyone explain this?

written by chaman original link here

Sqrt(x)(69)

Answer 1

Instead of using fancy Newton's method, this plain binary search approach also works.

```
public int sqrt(int x) {
    if (x == 0)
        return 0;
    int left = 1, right = Integer.MAX_VALUE;
    while (true) {
        int mid = left + (right - left)/2;
        if (mid > x/mid) {
            right = mid - 1;
        } else {
            if (mid + 1 > x/(mid + 1))
                return mid;
            left = mid + 1;
        }
    }
}
```

written by AlexTheGreat original link here

Answer 2

Basic Idea:

Since sqrt(x) is composed of binary bits, I calculate sqrt(x) by deciding every bit from the most significant to least significant. **Since an integer n can have O(log n) bits with each bit decided within constant time, this algorithm has time limit O(log n), actually, because an Integer can have at most 32 bits, I can also say this algorithm takes O(32)=O(1) time.**

```
 public int sqrt(int x) {
    if(x==0)
        return 0;
    int h=0;
    while((long)(1<<h)*(long)(1<<h)<=x) // firstly, find the most significant bit
        h++;
    h--;
    int b=h-1;
    int res=(1<<h);
    while(b>=0){  // find the remaining bits
        if((long)(res | (1<<b))*(long)(res |(1<<b))<=x)
            res|=(1<<b);
        b--;
    }
    return res;
}
```

written by yuyibestman original link here

Answer 3

Quite a few people used Newton already, but I didn't see someone make it this short. Same solution in every language. Explanation under the solutions.

C++ and C

```
    long r = x;
    while (r*r > x)
        r = (r + x/r) / 2;
    return r;
```

Python

```
    r = x
    while r*r > x:
        r = (r + x/r) / 2
    return r
```

Ruby

```
    r = x
    r = (r + x/r) / 2 while r*r > x
    r
```

Java and C#

```
    long r = x;
    while (r*r > x)
        r = (r + x/r) / 2;
    return (int) r;
```

JavaScript

```
    r = x;
    while (r*r > x)
        r = ((r + x/r) / 2) | 0;
    return r;
```

Explanation

Apparently, using only integer division for the Newton method works. And I guessed that if I start at x, the root candidate will decrease monotonically and never get too small.

The above solutions all got accepted, and in C++ I also verified it locally on my PC for all possible inputs (0 to 2147483647):

```
#include <iostream>
#include <climits>
using namespace std;

int mySqrt(int x) {
    long long r = x;
    while (r*r > x)
        r = (r + x/r) / 2;
    return r;
}

int main() {
    for (long long x=0; x<=INT_MAX; ++x) {
        long long r = mySqrt(x);
        if (r<0 || r*r > x || (r+1)*(r+1) <= x)
            cout << "false: " << x << " " << r << endl;
        if (x % 10000000 == 0)
            cout << x << endl;
    }
    cout << "all checked" << endl;
}
```

written by StefanPochmann original link here

Climbing Stairs(70)

Answer 1

The problem seems to be a *dynamic programming* one. **Hint**: the tag also suggests that! Here are the steps to get the solution incrementally.

- Base cases:
  if n <= 0, then the number of ways should be zero. if n == 1, then there is only way to climb the stair. if n == 2, then there are two ways to climb the stairs. One solution is one step by another; the other one is two steps at one time.
- The key intuition to solve the problem is that given a number of stairs n, if we know the number ways to get to the points `[n-1]` and `[n-2]` respectively, denoted as `n1` and `n2` , then the total ways to get to the point `[n]` is `n1 + n2` . Because from the `[n-1]` point, we can take one single step to reach `[n]` . And from the `[n-2]` point, we could take two steps to get there. There is NO overlapping between these two solution sets, because we differ in the final step.

Now given the above intuition, one can construct an array where each node stores the solution for each number n. Or if we look at it closer, it is clear that this is basically a fibonacci number, with the starting numbers as 1 and 2, instead of 1 and 1.

The implementation in Java as follows:

```
public int climbStairs(int n) {
    // base cases
    if(n <= 0) return 0;
    if(n == 1) return 1;
    if(n == 2) return 2;

    int one_step_before = 2;
    int two_steps_before = 1;
    int all_ways = 0;

    for(int i=2; i<n; i++){
        all_ways = one_step_before + two_steps_before;
        one_step_before = two_steps_before;
        two_steps_before = all_ways;
    }
    return all_ways;
}
```

written by liaison original link here

Answer 2

Hi guys, I come up with this arithmetic way. Find the inner logic relations and get the answer.

```
public class Solution {

public int climbStairs(int n) {
    if(n == 0 || n == 1 || n == 2){return n;}
    int[] mem = new int[n];
    mem[0] = 1;
    mem[1] = 2;
    for(int i = 2; i < n; i++){
        mem[i] = mem[i-1] + mem[i-2];
    }
    return mem[n-1];
}
```

}

written by tangxukai original link here

Answer 3

Same simple algorithm written in every offered language. Variable `a` tells you the number of ways to reach the current step, and `b` tells you the number of ways to reach the next step. So for the situation one step further up, the old `b` becomes the new `a`, and the new `b` is the old `a+b`, since that new step can be reached by climbing 1 step from what `b` represented or 2 steps from what `a` represented.

Ruby wins, and *"the C languages"* all look the same.

Ruby (60 ms)

```
def climb_stairs(n)
    a = b = 1
    n.times { a, b = b, a+b }
    a
end
```

C++ (0 ms)

```
int climbStairs(int n) {
    int a = 1, b = 1;
    while (n--)
        a = (b += a) - a;
    return a;
}
```

Java (208 ms)

```java
public int climbStairs(int n) {
    int a = 1, b = 1;
    while (n-- > 0)
        a = (b += a) - a;
    return a;
}
```

Python (52 ms)

```python
def climbStairs(self, n):
    a = b = 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

C (0 ms)

```c
int climbStairs(int n) {
    int a = 1, b = 1;
    while (n--)
        a = (b += a) - a;
    return a;
}
```

C# (48 ms)

```csharp
public int ClimbStairs(int n) {
    int a = 1, b = 1;
    while (n-- > 0)
        a = (b += a) - a;
    return a;
}
```

Javascript (116 ms)

```javascript
var climbStairs = function(n) {
    a = b = 1
    while (n--)
        a = (b += a) - a
    return a
};
```

written by StefanPochmann original link here

Simplify Path(71)

Answer 1

C++ also have *getline* which acts like Java's *split*. I guess the code can comment itself.

```
string simplifyPath(string path) {
    string res, tmp;
    vector<string> stk;
    stringstream ss(path);
    while(getline(ss,tmp,'/')) {
        if (tmp == "" or tmp == ".") continue;
        if (tmp == ".." and !stk.empty()) stk.pop_back();
        else if (tmp != "..") stk.push_back(tmp);
    }
    for(auto str : stk) res += "/"+str;
    return res.empty() ? "/" : res;
}
```

written by monaziyi original link here

Answer 2

Hi guys!

The main idea is to push to the stack every valid file name (not in {"",".",".."}), popping only if there's smth to pop and we met "..". I don't feel like the code below needs any additional comments.

```
public String simplifyPath(String path) {
    Deque<String> stack = new LinkedList<>();
    Set<String> skip = new HashSet<>(Arrays.asList("..",".",""));
    for (String dir : path.split("/")) {
        if (dir.equals("..") && !stack.isEmpty()) stack.pop();
        else if (!skip.contains(dir)) stack.push(dir);
    }
    String res = "";
    for (String dir : stack) res = "/" + dir + res;
    return res.isEmpty() ? "/" : res;
}
```

Hope it helps!

written by shpolsky original link here

Answer 3

1. traverse the string to record each folder name.
2. two special cases:

a.double dot:pop one.

b.single dot: do nothing (don`t push it).

```
string simplifyPath(string path) {
    vector<string>   nameVect;
    string name;

    path.push_back('/');
    for(int i=0;i<path.size();i++){
        if(path[i]=='/'){
            if(name.size()==0)continue;
            if(name==".."){     //special case 1ï¼šdouble dotï¼pop dir
                 if(nameVect.size()>0)nameVect.pop_back();
            }else if(name=="."){//special case 2:singel dotï¼don`t push
            }else{
                nameVect.push_back(name);
            }
            name.clear();
        }else{
            name.push_back(path[i]);//record the name
        }
    }

    string result;
    if(nameVect.empty())return "/";
    for(int i=0;i<nameVect.size();i++){
        result.append("/"+nameVect[i]);
    }
    return result;
}
```

written by enriquewang original link here

Edit Distance(72)

Answer 1

This is a classic problem of Dynamic Programming. We define the state `dp[i][j]` to be the minimum number of operations to convert `word1[0..i - 1]` to `word2[0..j - 1]`. The state equations have two cases: the boundary case and the general case. Note that in the above notations, both `i` and `j` take values starting from `1`.

For the boundary case, that is, to convert a string to an empty string, it is easy to see that the mininum number of operations to convert `word1[0..i - 1]` to `""` requires at least `i` operations (deletions). In fact, the boundary case is simply:

1. `dp[i][0] = i`;
2. `dp[0][j] = j`.

Now let's move on to the general case, that is, convert a non-empty `word1[0..i - 1]` to another non-empty `word2[0..j - 1]`. Well, let's try to break this problem down into smaller problems (sub-problems). Suppose we have already known how to convert `word1[0..i - 2]` to `word2[0..j - 2]`, which is `dp[i - 1][j - 1]`. Now let's consider `word[i - 1]` and `word2[j - 1]`. If they are euqal, then no more operation is needed and `dp[i][j] = dp[i - 1][j - 1]`. Well, what if they are not equal?

If they are not equal, we need to consider three cases:

1. Replace `word1[i - 1]` by `word2[j - 1]` (`dp[i][j] = dp[i - 1][j - 1] + 1 (for replacement)`);
2. Delete `word1[i - 1]` and `word1[0..i - 2] = word2[0..j - 1]` (`dp[i][j] = dp[i - 1][j] + 1 (for deletion)`);
3. Insert `word2[j - 1]` to `word1[0..i - 1]` and `word1[0..i - 1] + word2[j - 1] = word2[0..j - 1]` (`dp[i][j] = dp[i][j - 1] + 1 (for insertion)`).

Make sure you understand the subtle differences between the equations for deletion and insertion. For deletion, we are actually converting `word1[0..i - 2]` to `word2[0..j - 1]`, which costs `dp[i - 1][j]`, and then deleting the `word1[i - 1]`, which costs `1`. The case is similar for insertion.

Putting these together, we now have:

1. `dp[i][0] = i`;
2. `dp[0][j] = j`;
3. `dp[i][j] = dp[i - 1][j - 1]`, if `word1[i - 1] = word2[j - 1]`;
4. `dp[i][j] = min(dp[i - 1][j - 1] + 1, dp[i - 1][j] + 1, dp[i][j - 1] + 1)`, otherwise.

The above state equations can be turned into the following code directly.

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.length(), n = word2.length();
        vector<vector<int> > dp(m + 1, vector<int> (n + 1, 0));
        for (int i = 1; i <= m; i++)
            dp[i][0] = i;
        for (int j = 1; j <= n; j++)
            dp[0][j] = j;
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1[i - 1] == word2[j - 1])
                    dp[i][j] = dp[i - 1][j - 1];
                else dp[i][j] = min(dp[i - 1][j - 1] + 1, min(dp[i][j - 1] + 1,
dp[i - 1][j] + 1));
            }
        }
        return dp[m][n];
    }
};
```

Well, you may have noticed that each time when we update `dp[i][j]`, we only need `dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]`. In fact, we need not maintain the full `m*n` matrix. Instead, maintaing one column is enough. The code can be optimized to `O(m)` or `O(n)` space, depending on whether you maintain a row or a column of the original matrix.

The optimized code is as follows.

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.length(), n = word2.length();
        vector<int> cur(m + 1, 0);
        for (int i = 1; i <= m; i++)
            cur[i] = i;
        for (int j = 1; j <= n; j++) {
            int pre = cur[0];
            cur[0] = j;
            for (int i = 1; i <= m; i++) {
                int temp = cur[i];
                if (word1[i - 1] == word2[j - 1])
                    cur[i] = pre;
                else cur[i] = min(pre + 1, min(cur[i] + 1, cur[i - 1] + 1));
                pre = temp;
            }
        }
        return cur[m];
    }
};
```

Well, if you find the above code hard to understand, you may first try to write a two-column version that explicitly maintains two columns (the previous column and the

current column) and then simplify the two-column version into the one-column version like the above code :-)

written by jianchao.li.fighter original link here

Answer 2

Use f[i][j] to represent the shortest edit distance between word1[0,i) and word2[0, j). Then compare the last character of word1[0,i) and word2[0,j), which are c and d respectively (c == word1[i-1], d == word2[j-1]):

if c == d, then : f[i][j] = f[i-1][j-1]

Otherwise we can use three operations to convert word1 to word2:

(a) if we replaced c with d: f[i][j] = f[i-1][j-1] + 1;

(b) if we added d after c: f[i][j] = f[i][j-1] + 1;

(c) if we deleted c: f[i][j] = f[i-1][j] + 1;

Note that f[i][j] only depends on f[i-1][j-1], f[i-1][j] and f[i][j-1], therefore we can reduce the space to O(n) by using only the (i-1)th array and previous updated element(f[i][j-1]).

```
int minDistance(string word1, string word2) {

        int l1 = word1.size();
        int l2 = word2.size();

        vector<int> f(l2+1, 0);
        for (int j = 1; j <= l2; ++j)
            f[j] = j;

        for (int i = 1; i <= l1; ++i)
        {
            int prev = i;
            for (int j = 1; j <= l2; ++j)
            {
                int cur;
                if (word1[i-1] == word2[j-1]) {
                    cur = f[j-1];
                } else {
                    cur = min(min(f[j-1], prev), f[j]) + 1;
                }

                f[j-1] = prev;
                prev = cur;
            }
            f[l2] = prev;
        }
        return f[l2];

    }
```

Actually at first glance I thought this question was similar to Word Ladder and I

tried to solve it using BFS(pretty stupid huh?). But in fact, the main difference is that there's a strict restriction on the intermediate words in Word Ladder problem, while there's no restriction in this problem. If we added some restriction on intermediate words for this question, I don't think this DP solution would still work.

written by eaglesky1990 original link here

Answer 3

http://www.stanford.edu/class/cs124/lec/med.pdf

written by vikram.kuruguntla original link here

Set Matrix Zeroes(73)

Answer 1

My idea is simple: store states of each row in the first of that row, and store states of each column in the first of that column. Because the state of row0 and the state of column0 would occupy the same cell, I let it be the state of row0, and use another variable "col0" for column0. In the first phase, use matrix elements to set states in a top-down way. In the second phase, use states to set matrix elements in a bottom-up way.

```
void setZeroes(vector<vector<int> > &matrix) {
    int col0 = 1, rows = matrix.size(), cols = matrix[0].size();

    for (int i = 0; i < rows; i++) {
        if (matrix[i][0] == 0) col0 = 0;
        for (int j = 1; j < cols; j++)
            if (matrix[i][j] == 0)
                matrix[i][0] = matrix[0][j] = 0;
    }

    for (int i = rows - 1; i >= 0; i--) {
        for (int j = cols - 1; j >= 1; j--)
            if (matrix[i][0] == 0 || matrix[0][j] == 0)
                matrix[i][j] = 0;
        if (col0 == 0) matrix[i][0] = 0;
    }
}
```

written by mzchen original link here

Answer 2

```
public class Solution {
public void setZeroes(int[][] matrix) {
    boolean fr = false,fc = false;
    for(int i = 0; i < matrix.length; i++) {
        for(int j = 0; j < matrix[0].length; j++) {
            if(matrix[i][j] == 0) {
                if(i == 0) fr = true;
                if(j == 0) fc = true;
                matrix[0][j] = 0;
                matrix[i][0] = 0;
            }
        }
    }
    for(int i = 1; i < matrix.length; i++) {
        for(int j = 1; j < matrix[0].length; j++) {
            if(matrix[i][0] == 0 || matrix[0][j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }
    if(fr) {
        for(int j = 0; j < matrix[0].length; j++) {
            matrix[0][j] = 0;
        }
    }
    if(fc) {
        for(int i = 0; i < matrix.length; i++) {
            matrix[i][0] = 0;
        }
    }

}
```

}

written by lz2343 original link here

Answer 3

I find the last row which has 0, and use it to store the 0-collumns. Then go row by row set them to 0. Then go column by column set them to 0. Finally set the last row which has 0. It's long but hey it's O(1)

```
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {

        int H = matrix.size();
        int W = matrix[0].size();

        // find the last 0 row
        int last_0_row = -1;
        for (int y = H - 1; y >= 0 && last_0_row == -1; y--)
            for (int x = 0; x < W; x++)
                if (matrix[y][x] == 0)
                {
                    last_0_row = y;
                    break;
                }
        if (last_0_row == -1)
            return;

        // go row by row
        for (int y = 0; y < last_0_row; y++)
        {
            bool this_is_a_0_row = false;

            for (int x = 0; x < W; x++)
            {
                if (matrix[y][x] == 0)
                {
                    this_is_a_0_row = true;
                    matrix[last_0_row][x] = 0;
                }
            }

            if (this_is_a_0_row)
            for (int x = 0; x < W; x++)
            {
                matrix[y][x] = 0;
            }
        }

        // set collums to 0
        for (int y = 0; y < H; y++)
        for (int x = 0; x < W; x++)
        {
            if (matrix[last_0_row][x] == 0)
                matrix[y][x] = 0;
        }

        // set the last 0 row
        for (int x = 0; x < W; x++)
        {
            matrix[last_0_row][x] = 0;
        }
    }
};
```

written by lugiavn original link here

Search a 2D Matrix(74)

Answer 1

Use binary search.

n * m matrix convert to an array => matrix[x][y] => a[x * m + y]

an array convert to n * m matrix => a[x] =>matrix[x / m][x % m];

```
class Solution {
public:
    bool searchMatrix(vector<vector<int> > &matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int l = 0, r = m * n - 1;
        while (l != r){
            int mid = (l + r - 1) >> 1;
            if (matrix[mid / m][mid % m] < target)
                l = mid + 1;
            else
                r = mid;
        }
        return matrix[r / m][r % m] == target;
    }
};
```

written by vaputa original link here

Answer 2

```java
/**
 *  Do binary search in this "ordered" matrix
 */
public boolean searchMatrix(int[][] matrix, int target) {

    int row_num = matrix.length;
    int col_num = matrix[0].length;

    int begin = 0, end = row_num * col_num - 1;

    while(begin <= end){
        int mid = (begin + end) / 2;
        int mid_value = matrix[mid/col_num][mid%col_num];

        if( mid_value == target){
            return true;

        }else if(mid_value < target){
            //Should move a bit further, otherwise dead loop.
            begin = mid+1;
        }else{
            end = mid-1;
        }
    }

    return false;
}
```

written by liaison original link here

Answer 3

```java
  Solution {  public:
    bool searchMatrix(vector<vector<int> > &matrix, int target) {
        int n = (int)matrix.size();
        int m = (int)matrix[0].size();
        --n; --m;
        while(n > 0 && matrix[n - 1][m] >= target) --n;
        while(m > 0 && matrix[n][m - 1] >= target) --m;
        return (matrix[n][m] == target);
    }
};
```

I just used that fact, that number in the matrix increases

written by Oleksiy original link here

Sort Colors(75)

Answer 1

The idea is to sweep all 0s to the left and all 2s to the right, then all 1s are left in the middle.

```
class Solution {
public:
    void sortColors(int A[], int n) {
        int second=n-1, zero=0;
        for (int i=0; i<=second; i++) {
            while (A[i]==2 && i<second) swap(A[i], A[second--]);
            while (A[i]==0 && i>zero) swap(A[i], A[zero++]);
        }
    }
};
```

written by yidawang original link here

Answer 2

```
// two pass O(m+n) space
void sortColors(int A[], int n) {
    int num0 = 0, num1 = 0, num2 = 0;

    for(int i = 0; i < n; i++) {
        if (A[i] == 0) ++num0;
        else if (A[i] == 1) ++num1;
        else if (A[i] == 2) ++num2;
    }

    for(int i = 0; i < num0; ++i) A[i] = 0;
    for(int i = 0; i < num1; ++i) A[num0+i] = 1;
    for(int i = 0; i < num2; ++i) A[num0+num1+i] = 2;
}

// one pass in place solution
void sortColors(int A[], int n) {
    int n0 = -1, n1 = -1, n2 = -1;
    for (int i = 0; i < n; ++i) {
        if (A[i] == 0)
        {
            A[++n2] = 2; A[++n1] = 1; A[++n0] = 0;
        }
        else if (A[i] == 1)
        {
            A[++n2] = 2; A[++n1] = 1;
        }
        else if (A[i] == 2)
        {
            A[++n2] = 2;
        }
    }
}

// one pass in place solution
void sortColors(int A[], int n) {
    int j = 0, k = n - 1;
    for (int i = 0; i <= k; ++i){
        if (A[i] == 0 && i != j)
            swap(A[i--], A[j++]);
        else if (A[i] == 2 && i != k)
            swap(A[i--], A[k--]);
    }
}

// one pass in place solution
void sortColors(int A[], int n) {
    int j = 0, k = n-1;
    for (int i=0; i <= k; i++) {
        if (A[i] == 0)
            swap(A[i], A[j++]);
        else if (A[i] == 2)
            swap(A[i--], A[k--]);
    }
}
```

written by shichaotan original link here

Answer 3

```
class Solution {
public:
//use counting sort
void sortColors(int A[], int n) {
int red = -1, white = -1, blue = -1;

for(int i = 0; i < n; i++){
    if(A[i] == 0){
        A[++blue] = 2;
        A[++white] = 1;
        A[++red] = 0;
    }
    else if(A[i] == 1){
        A[++blue] = 2;
        A[++white] = 1;
    }
    else if(A[i] == 2)
        A[++blue] = 2;
}
}
};
```

the clever thing is that use three variable to store the three colors' index position. When you face A[i] == 0, all the variables add 1 because 0 is former. Do the same thing to other 2 situation.

Ex: If you just face 2, just need to assign 2 to the A[++blue], and "++blue" will increase "blue" with 1. Next if you face 0, you will increase 3 variable and assign the number to A!

It will make sure you always get the right sorted array when you run the for loop.

written by autekroy original link here

Minimum Window Substring(76)

Answer 1

```
class Solution {
public:
    string minWindow(string S, string T) {
        if (S.empty() || T.empty())
        {
            return "";
        }
        int count = T.size();
        int require[128] = {0};
        bool chSet[128] = {false};
        for (int i = 0; i < count; ++i)
        {
            require[T[i]]++;
            chSet[T[i]] = true;
        }
        int i = -1;
        int j = 0;
        int minLen = INT_MAX;
        int minIdx = 0;
        while (i < (int)S.size() && j < (int)S.size())
        {
            if (count)
            {
                i++;
                require[S[i]]--;
                if (chSet[S[i]] && require[S[i]] >= 0)
                {
                    count--;
                }
            }
            else
            {
                if (minLen > i - j + 1)
                {
                    minLen = i - j + 1;
                    minIdx = j;
                }
                require[S[j]]++;
                if (chSet[S[j]] && require[S[j]] > 0)
                {
                    count++;
                }
                j++;
            }
        }
        if (minLen == INT_MAX)
        {
            return "";
        }
        return S.substr(minIdx, minLen);
    }
};
```

Implementation of mike3's idea

running time : 56ms.

written by heleifz original link here

Answer 2

```
string minWindow(string S, string T) {
    string result;
    if(S.empty() || T.empty()){
        return result;
    }
    unordered_map<char, int> map;
    unordered_map<char, int> window;
    for(int i = 0; i < T.length(); i++){
        map[T[i]]++;
    }
    int minLength = INT_MAX;
    int letterCounter = 0;
    for(int slow = 0, fast = 0; fast < S.length(); fast++){
        char c = S[fast];
        if(map.find(c) != map.end()){
            window[c]++;
            if(window[c] <= map[c]){
                letterCounter++;
            }
        }
        if(letterCounter >= T.length()){
            while(map.find(S[slow]) == map.end() || window[S[slow]] > map[S[slow]
]){
                window[S[slow]]--;
                slow++;
            }
            if(fast - slow + 1 < minLength){
                minLength = fast - slow + 1;
                result = S.substr(slow, minLength);
            }
        }
    }
    return result;
}
```

There are three key variables in my solution:

```
unordered_map <char, int> map; unordered_map<char, int> window; int letterCounter
;
```

variable "map" is used to indicate what characters and how many characters are in T.

variable "window" is to indicate what characters and how many characters are between pointer "slow" and pointer "fast".

Now let's start.

The first For loop is used to construct variable "map".

The second For loop is used to find the minimum window.

The first thing we should do in the second For loop is to find a window which can cover T. I use "letterCounter" to be a monitor. If "letterCounter" is equal to T.length(), then we find this window. Before that, only the first If clause can be executed. However, after we find this window, the second If clause can also be executed.

In the second If clause, we move "slow" forward in order to shrink the window size. Every time finding a smaller window, I update the result.

At the end of program, I return result, which is the minimum window.

written by zxyperfect original link here

Answer 3

Can the String T have repeating characters - for instance "AA"? In that case should the minimum window contain two A's or does it suffice for it have a single A.

written by hhk.1989 original link here

Combinations(77)

Answer 1

Basically, this solution follows the idea of the mathematical formula C(n,k)=C(n-1,k-1)+C(n-1,k).

Here C(n,k) is divided into two situations. Situation one, number n is selected, so we only need to select k-1 from n-1 next. Situation two, number n is not selected, and the rest job is selecting k from n-1.

```
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        if (k == n || k == 0) {
            List<Integer> row = new LinkedList<>();
            for (int i = 1; i <= k; ++i) {
                row.add(i);
            }
            return new LinkedList<>(Arrays.asList(row));
        }
        List<List<Integer>> result = this.combine(n - 1, k - 1);
        result.forEach(e -> e.add(n));
        result.addAll(this.combine(n - 1, k));
        return result;
    }
}
```

written by kxcf original link here

Answer 2

my idea is using backtracking ,every time I push a number into vector,then I push a bigger one into it; then i pop the latest one,and push a another bigger one... and if I has push k number into vector,I push this into result;

this solution take 24 ms.

```
class Solution {
public:
    vector<vector<int> > combine(int n, int k) {
        vector<vector<int> >res;
        if(n<k)return res;
        vector<int> temp(0,k);
        combine(res,temp,0,0,n,k);
        return res;
    }

    void combine(vector<vector<int> > &res,vector<int> &temp,int start,int num,in
t n ,int k){
        if(num==k){
            res.push_back(temp);
            return;
        }
        for(int i = start;i<n;i++){
            temp.push_back(i+1);
            combine(res,temp,i+1,num+1,n,k);
            temp.pop_back();
            }
        }
};
```

written by nangao original link here

Answer 3

```
    public static List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> combs = new ArrayList<List<Integer>>();
        combine(combs, new ArrayList<Integer>(), 1, n, k);
        return combs;
    }
    public static void combine(List<List<Integer>> combs, List<Integer> comb, int
start, int n, int k) {
        if(k==0) {
            combs.add(new ArrayList<Integer>(comb));
            return;
        }
        for(int i=start;i<=n;i++) {
            comb.add(i);
            combine(combs, comb, i+1, n, k-1);
            comb.remove(comb.size()-1);
        }
    }
```

written by fabrizio3 original link here

Subsets(78)

Answer 1

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort (S.begin(), S.end());
        int elem_num = S.size();
        int subset_num = pow (2, elem_num);
        vector<vector<int> > subset_set (subset_num, vector<int>());
        for (int i = 0; i < elem_num; i++)
            for (int j = 0; j < subset_num; j++)
                if ((j >> i) & 1)
                    subset_set[j].push_back (S[i]);
        return subset_set;
    }
};
```

written by thumike original link here

Answer 2

Recursive (Backtracking)

This is a typical problem that can be tackled by backtracking. Since backtracking has a more-or-less similar template, so I do not give explanations for this method.

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> subs;
        vector<int> sub;
        genSubsets(nums, 0, sub, subs);
        return subs;
    }
    void genSubsets(vector<int>& nums, int start, vector<int>& sub, vector<vector
<int>>& subs) {
        subs.push_back(sub);
        for (int i = start; i < nums.size(); i++) {
            sub.push_back(nums[i]);
            genSubsets(nums, i + 1, sub, subs);
            sub.pop_back();
        }
    }
};
```

Iterative

This problem can also be solved iteratively. Take `[1, 2, 3]` in the problem

statement as an example. The process of generating all the subsets is like:

1. Initially: `[[]]`
2. Adding the first number to all the existed subsets: `[[], [1]]`;
3. Adding the second number to all the existed subsets: `[[], [1], [2], [1, 2]]`;
4. Adding the third number to all the existed subsets: `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`.

Have you got the idea :-)

The code is as follows.

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> subs(1, vector<int>());
        for (int i = 0; i < nums.size(); i++) {
            int n = subs.size();
            for (int j = 0; j < n; j++) {
                subs.push_back(subs[j]);
                subs.back().push_back(nums[i]);
            }
        }
        return subs;
    }
};
```

Bit Manipulation

This is the most clever solution that I have seen. The idea is that to give all the possible subsets, we just need to exhaust all the possible combinations of the numbers. And each number has only two possibilities: either in or not in a subset. And this can be represented using a bit.

There is also another a way to visualize this idea. That is, if we use the above example, `1` appears once in every two consecutive subsets, `2` appears twice in every four consecutive subsets, and `3` appears four times in every eight subsets, shown in the following (initially the `8` subsets are all empty):

`[], [], [], [], [], [], [], []`

`[], [1], [], [1], [], [1], [], [1]`

`[], [1], [2], [1, 2], [], [1], [2], [1, 2]`

`[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]`

The code is as follows.

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        int num_subset = pow(2, nums.size());
        vector<vector<int> > res(num_subset, vector<int>());
        for (int i = 0; i < nums.size(); i++)
            for (int j = 0; j < num_subset; j++)
                if ((j >> i) & 1)
                    res[j].push_back(nums[i]);
        return res;
    }
};
```

Well, just a final remark. For Python programmers, this may be an easy task in practice since the `itertools` package has a function `combinations` for it :-)

written by jianchao.li.fighter original link here

Answer 3

```
class Solution {
public:
vector<vector<int> > subsets(vector<int> &S) {
    vector<vector<int> > res(1, vector<int>());
    sort(S.begin(), S.end());

    for (int i = 0; i < S.size(); i++) {
        int n = res.size();
        for (int j = 0; j < n; j++) {
            res.push_back(res[j]);
            res.back().push_back(S[i]);
        }
    }

    return res;
  }
};
```

written by jaewoo original link here

Word Search(79)

Answer 1

Here accepted solution based on recursion. To save memory I decuded to apply bit mask for every visited cell. Please check board[y][x] ^= 256;

```
public boolean exist(char[][] board, String word) {
    char[] w = word.toCharArray();
    for (int y=0; y<board.length; y++) {
        for (int x=0; x<board[y].length; x++) {
            if (exist(board, y, x, w, 0)) return true;
        }
    }
    return false;
}

private boolean exist(char[][] board, int y, int x, char[] word, int i) {
    if (i == word.length) return true;
    if (y<0 || x<0 || y == board.length || x == board[y].length) return false;
    if (board[y][x] != word[i]) return false;
    board[y][x] ^= 256;
    boolean exist = exist(board, y, x+1, word, i+1)
        || exist(board, y, x-1, word, i+1)
        || exist(board, y+1, x, word, i+1)
        || exist(board, y-1, x, word, i+1);
    board[y][x] ^= 256;
    return exist;
}
```

written by pavel-shlyk original link here

Answer 2

```
   class Solution {
    public:
         bool exist(vector<vector<char> > &board, string word) {
             m=board.size();
             n=board[0].size();
            for(int x=0;x<m;x++)
                for(int y=0;y<n;y++)
                {
                    if(isFound(board,word.c_str(),x,y))
                        return true;
                }
            return false;
        }
    private:
        int m;
        int n;
        bool isFound(vector<vector<char> > &board, const char* w, int x, int y)
        {
            if(x<0||y<0||x>=m||y>=n||board[x][y]=='\0'||*w!=board[x][y])
                return false;
            if(*(w+1)=='\0')
                return true;
            char t=board[x][y];
            board[x][y]='\0';
            if(isFound(board,w+1,x-1,y)||isFound(board,w+1,x+1,y)||isFound(board,
w+1,x,y-1)||isFound(board,w+1,x,y+1))
                return true;
            board[x][y]=t;
            return false;
        }
    };
```

written by pwh1 original link here

Answer 3

Typical dfs+backtracking question. It compare board[row][col] with word[start], if they match, change board[row][col] to '*' to mark it as visited. Then move to the next one (i.e. word[start+1]) and compare it to the current neighbors (doing it by recursion)

```
class Solution {
private:
    bool dfs(vector<vector<char>>& board, int row, int col, const string &word, i
nt start, int M, int N, int sLen)
    {
        char curC;
        bool res = false;
        if( (curC = board[row][col]) != word[start]) return false;
        if(start==sLen-1) return true;
        board[row][col] = '*';
        if(row>0) res = dfs(board, row-1, col, word, start+1, M, N, sLen);
        if(!res && row < M-1) res = dfs(board, row+1, col, word, start+1, M, N, s
Len);
        if(!res && col > 0)   res = dfs(board, row, col-1, word, start+1, M, N, s
Len);
        if(!res && col < N-1) res = dfs(board,  row, col+1, word, start+1, M, N,
sLen);
        board[row][col] = curC;
        return res;
    }

public:
    bool exist(vector<vector<char>>& board, string word) {
        int M,N,i,j,sLen = word.size();
        if( (M=board.size()) && (N=board[0].size()) && sLen)
        {
            for(i=0; i<M; ++i)
                for(j=0; j<N; ++j)
                    if(dfs(board, i, j, word, 0, M, N, sLen)) return true;
        }
        return false;
    }
};
```

written by dong.wang.1694 original link here

Remove Duplicates from Sorted Array II(80)

Answer 1

Same simple solution written in several languages. Just go through the numbers and include those in the result that haven't been included twice already.

C++

```
int removeDuplicates(vector<int>& nums) {
    int i = 0;
    for (int n : nums)
        if (i < 2 || n > nums[i-2])
            nums[i++] = n;
    return i;
}
```

Java

```
public int removeDuplicates(int[] nums) {
    int i = 0;
    for (int n : nums)
        if (i < 2 || n > nums[i-2])
            nums[i++] = n;
    return i;
}
```

Python

```
def removeDuplicates(self, nums):
    i = 0
    for n in nums:
        if i < 2 or n > nums[i-2]:
            nums[i] = n
            i += 1
    return i
```

Ruby

```
def remove_duplicates(nums)
    i = 0
    nums.each { |n| nums[(i+=1)-1] = n if i < 2 || n > nums[i-2] }
    i
end
```

written by StefanPochmann original link here

Answer 2

I think both Remove Duplicates from Sorted Array I and II could be solved in a consistent and more general way by allowing the duplicates to appear k times (k = 1 for problem I and k = 2 for problem II). Here is my way: we need a count variable to

keep how many times the duplicated element appears, if we encounter a different element, just set counter to 1, if we encounter a duplicated one, we need to check this count, if it is already k, then we need to skip it, otherwise, we can keep this element. The following is the implementation and can pass both OJ:

```
int removeDuplicates(int A[], int n, int k) {

            if (n <= k) return n;

            int i = 1, j = 1;
            int cnt = 1;
            while (j < n) {
                if (A[j] != A[j-1]) {
                    cnt = 1;
                    A[i++] = A[j];
                }
                else {
                    if (cnt < k) {
                        A[i++] = A[j];
                        cnt++;
                    }
                }
                ++j;
            }
            return i;
}
```

For more details, you can also see this post: LeetCode Remove Duplicates from Sorted Array I and II: O(N) Time and O(1) Space

written by tech-wonderland.net original link here

Answer 3

```
int removeDuplicates(vector<int>& nums) {
    int n = nums.size(), count = 0;
    for (int i = 2; i < n; i++)
        if (nums[i] == nums[i - 2 - count]) count++;
        else nums[i - count] = nums[i];
    return n - count;
}
```

written by aserdega original link here

Search in Rotated Sorted Array II(81)

Answer 1

```
class Solution {
public:
  bool search(int A[], int n, int target) {
    int lo =0, hi = n-1;
    int mid = 0;
    while(lo<hi){
          mid=(lo+hi)/2;
          if(A[mid]==target) return true;
          if(A[mid]>A[hi]){
              if(A[mid]>target && A[lo] <= target) hi = mid;
              else lo = mid + 1;
          }else if(A[mid] < A[hi]){
              if(A[mid]<target && A[hi] >= target) lo = mid + 1;
              else hi = mid;
          }else{
              hi--;
          }

    }
    return A[lo] == target ? true : false;
  }
};
```

written by ggyc1993 original link here

Answer 2

Since we will have some duplicate elements in this problem, it is a little tricky because sometimes we cannot decide whether to go to the left side or right side. So for this condition, I have to probe both left and right side simultaneously to decide which side we need to find the number. Only in this condition, the time complexity may be *O(n)*. The rest conditions are always *O(log n)*.

For example:

input: `113111111111`, Looking for *target* `3`.

Is my solution correct? My code is as followed:

```
public class Solution {
    public boolean search(int[] A, int target) {
        // IMPORTANT: Please reset any member data you declared, as
        // the same Solution instance will be reused for each test case.
        int i = 0;
        int j = A.length - 1;
        while(i <= j){
            int mid = (i + j) / 2;
            if(A[mid] == target)
                return true;
            else if(A[mid] < A[i]){
                if(target > A[j])
                    j = mid - 1;
                else if(target < A[mid])
                    j = mid - 1;
                else
                    i = mid + 1;
            }else if(A[mid] > A[i]){
                if(target < A[mid] && target >= A[i])
                    j = mid - 1;
                else
                    i = mid + 1;
            }else{ // A[mid] == A[i]
                if(A[mid] != A[j])
                    i = mid + 1;
                else{
                    boolean flag = true;
                    for(int k = 1; mid - k >= i && mid + k <= j; k++){
                        if(A[mid] != A[mid - k]){
                            j = mid - k;
                            flag = false;
                            break;
                        }else if(A[mid] != A[mid + k]){
                            i = mid + k;
                            flag = false;
                            break;
                        }
                    }
                    if(flag)
                        return false;
                }
            }
        }
        return false;
    }
}
```

written by baojialiang original link here

Answer 3

The idea is the same as the previous one without duplicates

```
1) everytime check if targe == nums[mid], if so, we find it.
2) otherwise, we check if the first half is in order (i.e. nums[left]<=nums[mid])

  and if so, go to step 3), otherwise, the second half is in order,   go to step
4)
3) check if target in the range of [left, mid-1] (i.e. nums[left]<=target < nums[
mid]), if so, do search in the first half, i.e. right = mid-1; otherwise, search
in the second half left = mid+1;
4)  check if target in the range of [mid+1, right] (i.e. nums[mid]<target <= nums
[right]), if so, do search in the second half, i.e. left = mid+1; otherwise searc
h in the first half right = mid-1;
```

The only difference is that due to the existence of duplicates, we can have nums[left] == nums[mid] and in that case, the first half could be out of order (i.e. NOT in the ascending order, e.g. [3 1 2 3 3 3 3]) and we have to deal this case separately. In that case, it is guaranteed that nums[right] also equals to nums[mid], so what we can do is to check if nums[mid]== nums[left] == nums[right] before the original logic, and if so, we can move left and right both towards the middle by 1. and repeat.

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int left = 0, right =  nums.size()-1, mid;

        while(left<=right)
        {
            mid = (left + right) >> 1;
            if(nums[mid] == target) return true;

            // the only difference from the first one, trickly case, just updat l
eft and right
            if( (nums[left] == nums[mid]) && (nums[right] == nums[mid]) ) {++left
; --right;}

            else if(nums[left] <= nums[mid])
            {
                if( (nums[left]<=target) && (nums[mid] > target) ) right = mid-1;
                else left = mid + 1;
            }
            else
            {
                if((nums[mid] < target) &&  (nums[right] >= target) ) left = mid+
1;
                else right = mid-1;
            }
        }
        return false;
    }
};
```

written by dong.wang.1694 original link here

Remove Duplicates from Sorted List II(82)

Answer 1

```
public ListNode deleteDuplicates(ListNode head) {
        if(head==null) return null;
        ListNode FakeHead=new ListNode(0);
        FakeHead.next=head;
        ListNode pre=FakeHead;
        ListNode cur=head;
        while(cur!=null){
            while(cur.next!=null&&cur.val==cur.next.val){
                cur=cur.next;
            }
            if(pre.next==cur){
                pre=pre.next;
            }
            else{
                pre.next=cur.next;
            }
            cur=cur.next;
        }
        return FakeHead.next;
    }
```

written by snowfish original link here

Answer 2

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return null;

    if (head.next != null && head.val == head.next.val) {
        while (head.next != null && head.val == head.next.val) {
            head = head.next;
        }
        return deleteDuplicates(head.next);
    } else {
        head.next = deleteDuplicates(head.next);
    }
    return head;
}
```

if current node is not unique, return deleteDuplicates with head.next. If current node is unique, link it to the result of next list made by recursive call. Any improvement?

written by totalheap original link here

Answer 3

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        ListNode **runner = &head;

        if(!head || !head->next)return head;

        while(*runner)
        {
            if((*runner)->next && (*runner)->next->val == (*runner)->val)
            {
                ListNode *temp = *runner;
                while(temp && (*runner)->val == temp->val)
                    temp = temp->next;

                *runner = temp;
            }
            else
                runner = &((*runner)->next);
        }

        return head;
    }
};
```

written by tommy1122337 original link here

Remove Duplicates from Sorted List(83)

Answer 1

This solution is inspired by renzid https://leetcode.com/discuss/33043/3-line-recursive-solution

```
public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null)return head;
        head.next = deleteDuplicates(head.next);
        return head.val == head.next.val ? head.next : head;
}
```

Enjoy!

written by wen587sort original link here

Answer 2

```
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode list = head;

         while(list != null) {
            if (list.next == null) {
                break;
            }
            if (list.val == list.next.val) {
                list.next = list.next.next;
            } else {
                list = list.next;
            }
        }

         return head;
    }
}
```

written by aleksey.danilevsky.5 original link here

Answer 3

I noticed that the solutions posted here are too long and complicated. They use unnecessary variables and/or checks etc. The solution can be much more concise. Here is my solution:

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        ListNode* cur = head;
        while (cur) {
            while (cur->next && cur->val == cur->next->val)
                cur->next = cur->next->next;
            cur = cur->next;
        }
        return head;
    }
};
```

Note about freeing memory. We need to free memory when we delete a node. But don't use `delete node;` construct on an interview without discussing it with the interviewer. A list node can be allocated in many different ways and we can use `delete node;` only if we are sure that the nodes were allocated with `new TreeNode(...);`.

written by alexander4 original link here

Largest Rectangle in Histogram(84)

Answer 1

I push a sentinel node back into the end of height to make the code logic more concise.

```
class Solution {
  public:
      int largestRectangleArea(vector<int> &height) {

          int ret = 0;
          height.push_back(0);
          vector<int> index;

          for(int i = 0; i < height.size(); i++)
          {
              while(index.size() > 0 && height[index.back()] >= height[i])
              {
                  int h = height[index.back()];
                  index.pop_back();

                  int sidx = index.size() > 0 ? index.back() : -1;
                  if(h * (i-sidx-1) > ret)
                      ret = h * (i-sidx-1);
              }
              index.push_back(i);
          }

          return ret;
      }
  };
```

written by sipiprotoss5 original link here

Answer 2

For explanation, please see http://www.geeksforgeeks.org/largest-rectangle-under-histogram/

```
public class Solution {
    public int largestRectangleArea(int[] height) {
        int len = height.length;
        Stack<Integer> s = new Stack<Integer>();
        int maxArea = 0;
        for(int i = 0; i <= len; i++){
            int h = (i == len ? 0 : height[i]);
            if(s.isEmpty() || h >= height[s.peek()]){
                s.push(i);
            }else{
                int tp = s.pop();
                maxArea = Math.max(maxArea, height[tp] * (s.isEmpty() ? i : i - 1
- s.peek()));
                i--;
            }
        }
        return maxArea;
    }
}
```

written by wz366 original link here

Answer 3

I was stuck and took an eye on Geeks4Geeks. I got the idea and tried to figure it out by myself... It takes me a lot of time to make it through....

EDITED: Now it is pretty concise....

```
public class Solution {
public int largestRectangleArea(int[] height) {
    if (height==null) return 0;//Should throw exception
    if (height.length==0) return 0;

    Stack<Integer> index= new Stack<Integer>();
    index.push(-1);
    int max=0;

    for  (int i=0;i<height.length;i++){
            //Start calculate the max value
        while (index.peek()>-1)
            if (height[index.peek()]>height[i]){
                int top=index.pop();
                max=Math.max(max,height[top]*(i-1-index.peek()));
            }else break;

        index.push(i);
    }
    while(index.peek()!=-1){
        int top=index.pop();
        max=Math.max(max,height[top]*(height.length-1-index.peek()));
    }
    return max;
}
```

}

written by reeclapple original link here

Maximal Rectangle(85)

Answer 1

The DP solution proceeds row by row, starting from the first row. Let the maximal rectangle area at row i and column j be computed by [right(i,j) - left(i,j)]*height(i,j).

All the 3 variables left, right, and height can be determined by the information from previous row, and also information from the current row. So it can be regarded as a DP solution. The transition equations are:

> left(i,j) = max(left(i-1,j), cur*left), cur*left can be determined from the current row
>
> right(i,j) = min(right(i-1,j), cur*right), cur*right can be determined from the current row
>
> height(i,j) = height(i-1,j) + 1, if matrix[i][j]=='1';
>
> height(i,j) = 0, if matrix[i][j]=='0'

The code is as below. The loops can be combined for speed but I separate them for more clarity of the algorithm.

```
class Solution {public:
int maximalRectangle(vector<vector<char> > &matrix) {
    if(matrix.empty()) return 0;
    const int m = matrix.size();
    const int n = matrix[0].size();
    int left[n], right[n], height[n];
    fill_n(left,n,0); fill_n(right,n,n); fill_n(height,n,0);
    int maxA = 0;
    for(int i=0; i<m; i++) {
        int cur_left=0, cur_right=n;
        for(int j=0; j<n; j++) { // compute height (can do this from either side)
            if(matrix[i][j]=='1') height[j]++;
            else height[j]=0;
        }
        for(int j=0; j<n; j++) { // compute left (from left to right)
            if(matrix[i][j]=='1') left[j]=max(left[j],cur_left);
            else {left[j]=0; cur_left=j+1;}
        }
        // compute right (from right to left)
        for(int j=n-1; j>=0; j--) {
            if(matrix[i][j]=='1') right[j]=min(right[j],cur_right);
            else {right[j]=n; cur_right=j;}
        }
        // compute the area of rectangle (can do this from either side)
        for(int j=0; j<n; j++)
            maxA = max(maxA,(right[j]-left[j])*height[j]);
    }
    return maxA;
}
```

};

If you think this algorithm is not easy to understand, you can try this example:

```
0 0 0 1 0 0 0
0 0 1 1 1 0 0
0 1 1 1 1 1 0
```

The vector "left" and "right" from row 0 to row 2 are as follows

row 0:

```
l: 0 0 0 3 0 0 0
r: 7 7 7 4 7 7 7
```

row 1:

```
l: 0 0 2 3 2 0 0
r: 7 7 5 4 5 7 7
```

row 2:

```
l: 0 1 2 3 2 1 0
r: 7 6 5 4 5 6 7
```

The vector "left" is computing the left boundary. Take (i,j)=(1,3) for example. On current row 1, the left boundary is at j=2. However, because matrix[1][3] is 1, you need to consider the left boundary on previous row as well, which is 3. So the real left boundary at (1,3) is 3.

I hope this additional explanation makes things clearer.

written by morrischen2008 original link here

Answer 2

This question is similar as [Largest Rectangle in Histogram]:

You can maintain a row length of Integer array H recorded its height of '1's, and scan and update row by row to find out the largest rectangle of each row.

For each row, if matrix[row][i] == '1'. H[i] +=1, or reset the H[i] to zero. and accroding the algorithm of [Largest Rectangle in Histogram], to update the maximum area.

```java
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix==null||matrix.length==0||matrix[0].length==0)
            return 0;
        int cLen = matrix[0].length;    // column length
        int rLen = matrix.length;       // row length
        // height array
        int[] h = new int[cLen+1];
        h[cLen]=0;
        int max = 0;


        for (int row=0;row<rLen;row++) {
            Stack<Integer> s = new Stack<Integer>();
            for (int i=0;i<cLen+1;i++) {
                if (i<cLen)
                    if(matrix[row][i]=='1')
                        h[i]+=1;
                    else h[i]=0;

                if (s.isEmpty()||h[s.peek()]<=h[i])
                    s.push(i);
                else {
                    while(!s.isEmpty()&&h[i]<h[s.peek()]){
                        int top = s.pop();
                        int area = h[top]*(s.isEmpty()?i:(i-s.peek()-1));
                        if (area>max)
                            max = area;
                    }
                    s.push(i);
                }
            }
        }
        return max;
    }
}
```

written by wangyushawn original link here

Answer 3

We can apply the maximum in histogram in each row of the 2D matrix. What we need is to maintain an int array for each row, which represent for the height of the histogram.

Please refer to https://leetcode.com/problems/largest-rectangle-in-histogram/ first.

Suppose there is a 2D matrix like

1 1 0 1 0 1

0 1 0 0 1 1

1 1 1 1 0 1

1 1 1 1 0 1

First initiate the height array as 1 1 0 1 0 1, which is just a copy of the first row. Then we can easily calculate the max area is 2.

Then update the array. We scan the second row, when the matrix[1][i] is 0, set the height[i] to 0; else height[i] += 1, which means the height has increased by 1. So the height array again becomes 0 2 0 0 1 2. The max area now is also 2.

Apply the same method until we scan the whole matrix. the last height arrays is 2 4 2 2 0 2, so the max area has been found as 2 * 4 = 8.

Then reason we scan the whole matrix is that the maximum value may appear in any row of the height.

Code as follows:

```java
public class Solution {
public int maximalRectangle(char[][] matrix) {
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;

    int[] height = new int[matrix[0].length];
    for(int i = 0; i < matrix[0].length; i ++){
        if(matrix[0][i] == '1') height[i] = 1;
    }
    int result = largestInLine(height);
    for(int i = 1; i < matrix.length; i ++){
        resetHeight(matrix, height, i);
        result = Math.max(result, largestInLine(height));
    }

    return result;
}

private void resetHeight(char[][] matrix, int[] height, int idx){
    for(int i = 0; i < matrix[0].length; i ++){
        if(matrix[idx][i] == '1') height[i] += 1;
        else height[i] = 0;
    }
}

public int largestInLine(int[] height) {
    if(height == null || height.length == 0) return 0;
    int len = height.length;
    Stack<Integer> s = new Stack<Integer>();
    int maxArea = 0;
    for(int i = 0; i <= len; i++){
        int h = (i == len ? 0 : height[i]);
        if(s.isEmpty() || h >= height[s.peek()]){
            s.push(i);
        }else{
            int tp = s.pop();
            maxArea = Math.max(maxArea, height[tp] * (s.isEmpty() ? i : i - 1 - s
.peek()));
            i--;
        }
    }
    return maxArea;
}
```

}

written by bu.will.9 original link here

Partition List(86)

Answer 1

```
ListNode *partition(ListNode *head, int x) {
    ListNode node1(0), node2(0);
    ListNode *p1 = &node1, *p2 = &node2;
    while (head) {
        if (head->val < x)
            p1 = p1->next = head;
        else
            p2 = p2->next = head;
        head = head->next;
    }
    p2->next = NULL;
    p1->next = node2.next;
    return node1.next;
}
```

written by shichaotan original link here

Answer 2

the basic idea is to maintain two queues, the first one stores all nodes with val less than x , and the second queue stores all the rest nodes. Then concat these two queues. Remember to set the tail of second queue a null next, or u will get TLE.

```
public ListNode partition(ListNode head, int x) {
    ListNode dummy1 = new ListNode(0), dummy2 = new ListNode(0);  //dummy heads o
f the 1st and 2nd queues
    ListNode curr1 = dummy1, curr2 = dummy2;      //current tails of the two queu
es;
    while (head!=null){
        if (head.val<x) {
            curr1.next = head;
            curr1 = head;
        }else {
            curr2.next = head;
            curr2 = head;
        }
        head = head.next;
    }
    curr2.next = null;          //important! avoid cycle in linked list. otherwis
e u will get TLE.
    curr1.next = dummy2.next;
    return dummy1.next;
}
```

written by cbmbbz original link here

Answer 3

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode left(0), right(0);
        ListNode *l = &left, *r = &right;

        while(head){
            ListNode* & ref = head->val < x ? l : r;
            ref->next = head;
            ref = ref->next;

            head = head->next;
        }
        l->next = right.next;
        r->next = NULL;
        return left.next;
    }
};
```

written by yeze322 original link here

Scramble String(87)

Answer 1

Assume the strings are all lower case letters

```
class Solution {
public:
    bool isScramble(string s1, string s2) {
        if(s1==s2)
            return true;

        int len = s1.length();
        int count[26] = {0};
        for(int i=0; i<len; i++)
        {
            count[s1[i]-'a']++;
            count[s2[i]-'a']--;
        }

        for(int i=0; i<26; i++)
        {
            if(count[i]!=0)
                return false;
        }

        for(int i=1; i<=len-1; i++)
        {
            if( isScramble(s1.substr(0,i), s2.substr(0,i)) && isScramble(s1.subst
r(i), s2.substr(i)))
                return true;
            if( isScramble(s1.substr(0,i), s2.substr(len-i)) && isScramble(s1.sub
str(i), s2.substr(0,len-i)))
                return true;
        }
        return false;
    }
};
```

written by raychan original link here

Answer 2

The example shows the case where left child ALWAYS has equal or one-less characters than right child. But since "abb" is a scramble of "bab", as suggested by a test case, strings are not always partitioned in the way as the example implies.

However, if the answer is Yes, I think scrambles just become permutations. Isn't it?

So I am so confused what is expected...

Thanks!

written by diaz900 original link here

Answer 3

The basic idea is to divide s1(s2) into two substrings with length k and len-k and check if the two substrings s1[0..k-1] and s1[k, len-1] are the scrambles of s2[0..k-1] and s2[k,len-1] or s2[len-k, len-1] and s2[0..len-k-1] via recursion. The straigtforward recursion will be very slow due to many repeated recursive function calls. To speed up the recursion, we can use an unordered_map isScramblePair to save intermediate results. The key used here is s1+s2, but other keys are also possible (e.g. using indices)

```
    class Solution {
        bool DP_helper(unordered_map<string, bool> &isScramblePair, string s1, st
ring s2)
        {
            int i,len = s1.size();
            bool res = false;
            if(0==len) return true;
            else if(1==len) return s1 == s2;
            else
            {
                if(isScramblePair.count(s1+s2)) return isScramblePair[s1+s2]; //
checked before, return intermediate result directly
                if(s1==s2) res=true;
                else{
                    for(i=1; i<len && !res; ++i)
                    {
//check s1[0..i-1] with s2[0..i-1] and s1[i..len-1] and s2[i..len-1]
                        res = res || (DP_helper(isScramblePair, s1.substr(0,i), s
2.substr(0,i)) && DP_helper(isScramblePair, s1.substr(i,len-i), s2.substr(i,len-i)
));
 //if no match, then check s1[0..i-1] with s2[len-k.. len-1] and s1[i..len-1] and
s2[0..len-i]
                       res = res || (DP_helper(isScramblePair, s1.substr(0,i), s2
.substr(len-i,i)) && DP_helper(isScramblePair, s1.substr(i,len-i), s2.substr(0,len
-i)));
                    }
                }
                return isScramblePair[s1+s2]= res; //save the intermediate result
s

            }
        }
    public:
        bool isScramble(string s1, string s2) {
           unordered_map<string, bool> isScramblePair;
           return DP_helper(isScramblePair, s1, s2);
        }
    };
```

The recursive version has exponential complexity. To further improve the performance, we can use bottom-up DP, which is O(N^4) complexity. Here we build a table isS[len][i][j], which indicates whether s1[i..i+len-1] is a scramble of s2[j..j+len-1].

```
class Solution {
public:
    bool isScramble(string s1, string s2) {
        int sSize = s1.size(), len, i, j, k;
        if(0==sSize) return true;
        if(1==sSize) return s1==s2;
        bool isS[sSize+1][sSize][sSize];

        for(i=0; i<sSize; ++i)
            for(j=0; j<sSize; ++j)
                isS[1][i][j] = s1[i] == s2[j];

        for(len=2; len <=sSize; ++len)
            for(i=0; i<=sSize-len; ++i)
                for(j=0; j<=sSize-len; ++j)
                {
                    isS[len][i][j] = false;
                        for(k=1; k<len && !isS[len][i][j]; ++k)
                        {
                            isS[len][i][j] = isS[len][i][j] || (isS[k][i][j] && i
sS[len-k][i+k][j+k]);
                            isS[len][i][j] = isS[len][i][j] || (isS[k][i+len-k][j
] && isS[len-k][i][j+k]);
                        }
                }
        return isS[sSize][0][0];

    }
};
```

Furhtermore, in many cases, we found we can terminate our recursion early by pruning: i.e. by first checking if s1 and s2 have the same character set before we do recursion: if not, just terminate without recursion. This observation leads us to the following Recursion+cache+pruning version. Here the key of the cache changes to idx1*sSize* +*idx2* + *len*sSize*sSize;

```
class Solution {
private:
    bool DP_helper(string &s1, string &s2, int idx1, int idx2, int len, char isS[
])
    {
        int sSize = s1.size(),i, j, k, hist[26] , zero_count =0;
        if(isS[(len*sSize+idx1)*sSize+idx2]) return isS[(len*sSize+idx1)*sSize+id
x2]==1;
        bool res = false;

        fill_n(hist, 26, 0);
        for(k=0; k<len;++k)
        { // check if s1[idx1:idx1+len-1] and s2[idx2:idx2+len-1] have same chara
cters
            zero_count +=  (0==hist[s1[idx1+k]-'a']) - (0== ++hist[s1[idx1+k]-'a'
]);
            zero_count +=  (0==hist[s2[idx2+k]-'a']) - (0== --hist[s2[idx2+k]-'a'
]);
        }
        if(zero_count) {isS[(len*sSize+idx1)*sSize+idx2] = 2; return false;} //if
not, return directly
        if(len==1)     {isS[(len*sSize+idx1)*sSize+idx2] = 1; return true;}
        for(k=1;k<len && !res;++k) //otherwise, recursion with cache
        {
            res = res || (DP_helper(s1, s2, idx1, idx2, k, isS) && DP_helper(s1,
s2, idx1+k, idx2+k, len-k, isS) );
            res = res || (DP_helper(s1, s2, idx1+len-k, idx2, k, isS) && DP_helpe
r(s1, s2, idx1, idx2+k, len-k, isS) );
        }
        isS[(len*sSize+idx1)*sSize+idx2] = res?1:2;
        return res;
    }
public:
    bool isScramble(string s1, string s2) {
        const int sSize = s1.size();
        if(0==sSize) return true;
        char isS[(sSize+1)*sSize*sSize];
        fill_n(isS, (sSize+1)*sSize*sSize, 0);
        return DP_helper(s1, s2, 0, 0, sSize, isS);
    }
};
```

written by dong.wang.1694 original link here

Merge Sorted Array(88)

Answer 1

```
class Solution {
public:
    void merge(int A[], int m, int B[], int n) {
        int i=m-1;
        int j=n-1;
        int k = m+n-1;
        while(i >=0 && j>=0)
        {
            if(A[i] > B[j])
                A[k--] = A[i--];
            else
                A[k--] = B[j--];
        }
        while(j>=0)
            A[k--] = B[j--];
    }
};
```

written by leetchunhui original link here

Answer 2

This code relies on the simple observation that once all of the numbers from `nums2` have been merged into `nums1`, the rest of the numbers in `nums1` that were not moved are already in the correct place.

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i = m - 1, j = n - 1, tar = m + n - 1;
        while (j >= 0) {
            nums1[tar--] = i >= 0 && nums1[i] > nums2[j] ? nums1[i--] : nums2[j--
];
        }
    }
};
```

written by deck original link here

Answer 3

```
public void merge(int A[], int m, int B[], int n) {
    int i=m-1, j=n-1, k=m+n-1;
    while (i>-1 && j>-1) A[k--]= (A[i]>B[j]) ? A[i--] : B[j--];
    while (j>-1)         A[k--]=B[j--];
}
```

written by annafan original link here

Gray Code(89)

Answer 1

I have a solution here which takes O(1) on space and no recursion used. Is this the best possible solution? (I combined the base cases in the loop as mike3 does. Thanks mike3!)

```
vector<int> grayCode(int n)
{
    vector<int> result(1, 0);
    for (int i = 0; i < n; i++) {
        int curCount = result.size();
        // push back all element in result in reverse order
        while (curCount) {
            curCount--;
            int curNum = result[curCount];
            curNum += (1<<i);
            result.push_back(curNum);
        }
    }
    return result;
}
```

written by scorpionsky original link here

Answer 2

```
public List<Integer> grayCode(int n) {
    List<Integer> result = new LinkedList<>();
    for (int i = 0; i < 1<<n; i++) result.add(i ^ i>>1);
    return result;
}
```

The idea is simple. G(i) = i^ (i/2).

written by jinrf original link here

Answer 3

```
class Solution {
public:
    vector<int> grayCode(int n) {
        int N(1 << n), tmp;
        vector<int> result;
        for(int i(0); i < N; i++)
        {
            tmp = i << 1;
            result.push_back((tmp^i) >> 1);
        }
        return result;
    }
};
```

As we known:

```
Gi = Bi+1 xor Bi
```

For example, trans binay '001' to gray code:

```
tmp = 001 << 1
```

then,

```
bin 0 0 0 1
tmp 0 0 1 0
-xor-----------
    0 0 1 1
```

and the gray code is:

```
0 0 1 1 >> 1 (ignore last bit) => 0 0 1
```

written by Blueve original link here

Subsets II(90)

Answer 1

To solve this problem, it is helpful to first think how many subsets are there. If there is no duplicate element, the answer is simply 2^n, where n is the number of elements. This is because you have two choices for each element, either putting it into the subset or not. So all subsets for this no-duplicate set can be easily constructed: num of subset

- (1 to 2^0) empty set is the first subset
- (2^0+1 to 2^1) add the first element into subset from (1)
- (2^1+1 to 2^2) add the second element into subset (1 to 2^1)
- (2^2+1 to 2^3) add the third element into subset (1 to 2^2)
- ....
- (2^(n-1)+1 to 2^n) add the nth element into subset(1 to 2^(n-1))

Then how many subsets are there if there are duplicate elements? We can treat duplicate element as a spacial element. For example, if we have duplicate elements (5, 5), instead of treating them as two elements that are duplicate, we can treat it as one special element 5, but this element has more than two choices: you can either NOT put it into the subset, or put ONE 5 into the subset, or put TWO 5s into the subset. Therefore, we are given an array (a1, a2, a3, ..., an) with each of them appearing (k1, k2, k3, ..., kn) times, the number of subset is (k1+1)*(k2+1)*...(kn+1). We can easily see how to write down all the subsets similar to the approach above.

```
    class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > totalset = {{}};
        sort(S.begin(),S.end());
        for(int i=0; i<S.size();){
            int count = 0; // num of elements are the same
            while(count + i<S.size() && S[count+i]==S[i])  count++;
            int previousN = totalset.size();
            for(int k=0; k<previousN; k++){
                vector<int> instance = totalset[k];
                for(int j=0; j<count; j++){
                    instance.push_back(S[i]);
                    totalset.push_back(instance);
                }
            }
            i += count;
        }
        return totalset;
        }
};
```

written by mathsam original link here

Answer 2

If we want to insert an element which is a dup, we can only insert it after the newly inserted elements from last step.

```
vector<vector<int> > subsetsWithDup(vector<int> &S) {
    sort(S.begin(), S.end());
    vector<vector<int>> ret = {{}};
    int size = 0, startIndex = 0;
    for (int i = 0; i < S.size(); i++) {
        startIndex = i >= 1 && S[i] == S[i - 1] ? size : 0;
        size = ret.size();
        for (int j = startIndex; j < size; j++) {
            vector<int> temp = ret[j];
            temp.push_back(S[i]);
            ret.push_back(temp);
        }
    }
    return ret;
}
```

written by yuruofeifei original link here

Answer 3

The characteristics of C++ reference is an outstanding tool for backtracking algorithm!

let us use [1,2,3,4] as an example to explain my solution:

```
subsets([1,2,3,4]) = []
                     // push(1)
                     [1, subsets([2,3,4])] // if push N times in subsets([2,3,4])
, the pop times is also N, so vec is also [1] after backtrack.
                     // pop(), push(2)
                     [2, subsets([3,4])]
                     // pop(), push(3)
                     [3, subsets([4])]
                     // pop(), push(4)
                     [4, subsets([])]
                     // pop()
```

Accepted 10ms c++ solution use backtracking for Subsets

```
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int> &nums) {
        std::sort(nums.begin(), nums.end());
        std::vector<std::vector<int> > res;
        std::vector<int> vec;
        subsets(res, nums, vec, 0);
        return res;
    }
private:
    void subsets(std::vector<std::vector<int> > &res, std::vector<int> &nums, std
::vector<int> &vec, int begin) {
        res.push_back(vec);
        for (int i = begin; i != nums.size(); ++i) {
            vec.push_back(nums[i]);
            subsets(res, nums, vec, i + 1);
            vec.pop_back();
        }
    }
};
```

Accepted 10ms c++ solution use backtracking for Subsets II

```
class Solution {
public:
    std::vector<std::vector<int> > subsetsWithDup(std::vector<int> &nums) {
        std::sort(nums.begin(), nums.end());
        std::vector<std::vector<int> > res;
        std::vector<int> vec;
        subsetsWithDup(res, nums, vec, 0);
        return res;
    }
private:
    void subsetsWithDup(std::vector<std::vector<int> > &res, std::vector<int> &nu
ms, std::vector<int> &vec, int begin) {
        res.push_back(vec);
        for (int i = begin; i != nums.size(); ++i)
            if (i == begin || nums[i] != nums[i - 1]) {
                vec.push_back(nums[i]);
                subsetsWithDup(res, nums, vec, i + 1);
                vec.pop_back();
            }
    }
};
```

written by prime_tang original link here

Decode Ways(91)

Answer 1

```
public class Solution {
    public int numDecodings(String s) {
        int n = s.length();
        if (n == 0) return 0;

        int[] memo = new int[n+1];
        memo[n]  = 1;
        memo[n-1] = s.charAt(n-1) != '0' ? 1 : 0;

        for (int i = n - 2; i >= 0; i--)
            if (s.charAt(i) == '0') continue;
            else memo[i] = (Integer.parseInt(s.substring(i,i+2))<=26) ? memo[i+1]
+memo[i+2] : memo[i+1];

        return memo[0];
    }
}
```

written by manky original link here

Answer 2

```
int numDecodings(string s) {
    if (!s.size() || s.front() == '0') return 0;
    // r2: decode ways of s[i-2] , r1: decode ways of s[i-1]
    int r1 = 1, r2 = 1;

    for (int i = 1; i < s.size(); i++) {
        // zero voids ways of the last because zero cannot be used separately
        if (s[i] == '0') r1 = 0;

        // possible two-digit letter, so new r1 is sum of both while new r2 is the
old r1
        if (s[i - 1] == '1' || s[i - 1] == '2' && s[i] <= '6') {
            r1 = r2 + r1;
            r2 = r1 - r2;
        }

        // one-digit letter, no new way added
        else {
            r2 = r1;
        }
    }

    return r1;
}
```

written by shichaotan original link here

Answer 3

```
 int n = s.size();
    if(n == 0 || s[0] == '0') return 0;
    if(n == 1) return 1;
    int res = 0,fn_1 = 1,fn_2 = 1;
    for(int i = 1;i < n;i++){
        int temp = fn_1;
        if(isValid(s[i])&&isValid(s[i-1],s[i]))  res+=fn_1+fn_2;
        if(!isValid(s[i])&&isValid(s[i-1],s[i])) res+=fn_2;
        if(isValid(s[i])&&!isValid(s[i-1],s[i])) res+=fn_1;
        if(!isValid(s[i])&&!isValid(s[i-1],s[i]))  return 0;
        fn_1 = res;
        fn_2 = temp;
        res = 0;
    }
    return fn_1;
}
bool isValid(char a,char b){
    return a == '1'||(a == '2' && b <='6');
}
bool isValid(char a){
    return a != '0';
}
```

written by wang.shuai.750 original link here

Reverse Linked List II(92)

Answer 1

Simply just reverse the list along the way using 4 pointers: dummy, pre, start, then

```
public ListNode reverseBetween(ListNode head, int m, int n) {
    if(head == null) return null;
    ListNode dummy = new ListNode(0); // create a dummy node to mark the head of this list
    dummy.next = head;
    ListNode pre = dummy; // make a pointer pre as a marker for the node before reversing
    for(int i = 0; i<m-1; i++) pre = pre.next;

    ListNode start = pre.next; // a pointer to the beginning of a sub-list that will be reversed
    ListNode then = start.next; // a pointer to a node that will be reversed

    // 1 - 2 -3 - 4 - 5 ; m=2; n =4 ---> pre = 1, start = 2, then = 3
    // dummy-> 1 -> 2 -> 3 -> 4 -> 5

    for(int i=0; i<n-m; i++)
    {
        start.next = then.next;
        then.next = pre.next;
        pre.next = then;
        then = start.next;
    }

    // first reversing : dummy->1 - 3 - 2 - 4 - 5; pre = 1, start = 2, then = 4
    // second reversing: dummy->1 - 4 - 3 - 2 - 5; pre = 1, start = 2, then = 5 (finish)

    return dummy.next;

}
```

written by ardyadipta original link here

Answer 2

```
 ListNode *reverseBetween(ListNode *head, int m, int n) {
    if(m==n)return head;
    n-=m;
    ListNode prehead(0);
    prehead.next=head;
    ListNode* pre=&prehead;
    while(--m)pre=pre->next;
    ListNode* pstart=pre->next;
    while(n--)
    {
        ListNode *p=pstart->next;
        pstart->next=p->next;
        p->next=pre->next;
        pre->next=p;
    }
    return prehead.next;
}
```

written by harpe1999 original link here

Answer 3

The basic idea is as follows:

(1) Create a `new_head` that points to `head` and use it to locate the immediate node before the `m`-th (notice that it is `1`-indexed) node `pre`;

(2) Set `cur` to be the immediate node after `pre` and at each time move the immediate node after `cur` (named `move`) to be the immediate node after `pre`. Repeat it for `n - m` times.

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        ListNode* new_head = new ListNode(0);
        new_head -> next = head;
        ListNode* pre = new_head;
        for (int i = 0; i < m - 1; i++)
            pre = pre -> next;
        ListNode* cur = pre -> next;
        for (int i = 0; i < n - m; i++) {
            ListNode* move = cur -> next;
            cur -> next = move -> next;
            move -> next = pre -> next;
            pre -> next = move;
        }
        return new_head -> next;
    }
};
```

written by jianchao.li.fighter original link here

Restore IP Addresses(93)

Answer 1

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<String>();
        int len = s.length();
        for(int i = 1; i<4 && i<len-2; i++){
            for(int j = i+1; j<i+4 && j<len-1; j++){
                for(int k = j+1; k<j+4 && k<len; k++){
                    String s1 = s.substring(0,i), s2 = s.substring(i,j), s3 = s.s
ubstring(j,k), s4 = s.substring(k,len);
                    if(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4)){
                        res.add(s1+"."+s2+"."+s3+"."+s4);
                    }
                }
            }
        }
        return res;
    }
    public boolean isValid(String s){
        if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' && s.length()>1) ||
Integer.parseInt(s)>255)
            return false;
        return true;
    }
}
```

3-loop divides the string s into 4 substring: s1, s2, s3, s4. Check if each substring is valid. In isValid, strings whose length greater than 3 or equals to 0 is not valid; or if the string's length is longer than 1 and the first letter is '0' then it's invalid; or the string whose integer representation greater than 255 is invalid.

written by fiona_mao original link here

Answer 2

```
public List<String> restoreIpAddresses(String s) {
    List<String> solutions = new ArrayList<String>();
    restoreIp(s, solutions, 0, "", 0);
    return solutions;
}

private void restoreIp(String ip, List<String> solutions, int idx, String restore
d, int count) {
    if (count > 4) return;
    if (count == 4 && idx == ip.length()) solutions.add(restored);

    for (int i=1; i<4; i++) {
        if (idx+i > ip.length()) break;
        String s = ip.substring(idx,idx+i);
        if ((s.startsWith("0") && s.length()>1) || (i==3 && Integer.parseInt(s) >
= 256)) continue;
        restoreIp(ip, solutions, idx+i, restored+s+(count==3?"" : "."), count+1);
    }
}
```

written by greatgrahambini original link here

Answer 3

the basic idea is to make three cuts into the string, separating it into four parts, each part contains 1~3 digits and it must be <255.

```
static List<String> restoreIpAddresses(String s) {
    List<String> ans = new ArrayList<String>();
    int len = s.length();
    for (int i = 1; i <=3; ++i){  // first cut
        if (len-i > 9) continue;
        for (int j = i+1; j<=i+3; ++j){  //second cut
            if (len-j > 6) continue;
            for (int k = j+1; k<=j+3 && k<len; ++k){  // third cut
                int a,b,c,d;                // the four int's seperated by "."
                a = Integer.parseInt(s.substring(0,i));
                b = Integer.parseInt(s.substring(i,j)); // notice that "01" can b
e parsed into 1. Need to deal with that later.
                c = Integer.parseInt(s.substring(j,k));
                d = Integer.parseInt(s.substring(k));
                if (a>255 || b>255 || c>255 || d>255) continue;
                String ip = a+"."+b+"."+c+"."+d;
                if (ip.length()<len+3) continue;  // this is to reject those int'
s parsed from "01" or "00"-like substrings
                ans.add(ip);
            }
        }
    }
    return ans;
}
```

written by cbmbbz original link here

Binary Tree Inorder Traversal(94)

Answer 1

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> list = new ArrayList<Integer>();

    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode cur = root;

    while(cur!=null || !stack.empty()){
        while(cur!=null){
            stack.add(cur);
            cur = cur.left;
        }
        cur = stack.pop();
        list.add(cur.val);
        cur = cur.right;
    }

    return list;
}
```

written by lvlolitte original link here

Answer 2

Method 1: Using one stack and the binary tree node will be changed. Easy ,not Practical

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        if(!root)
        return vector;
        stack<TreeNode *> stack;
        stack.push(root);
        while(!stack.empty())
        {
            TreeNode *pNode = stack.top();
            if(pNode->left)
            {
                stack.push(pNode->left);
                pNode->left = NULL;
            }
            else
            {
                vector.push_back(pNode->val);
                stack.pop();
                if(pNode->right)
                stack.push(pNode->right);
            }
        }
        return vector;
    }
};
```

Method 2: Using one stack and one unordered_map, this will not changed the node. Better

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        if(!root)
        return vector;
        unordered_map<TreeNode *, bool> map;//left child has been visited:true.
        stack<TreeNode *> stack;
        stack.push(root);
        while(!stack.empty())
        {
            TreeNode *pNode = stack.top();
            if(pNode->left && !map[pNode])
            {
                stack.push(pNode->left);
                map[pNode] = true;
            }
            else
            {
                vector.push_back(pNode->val);
                stack.pop();
                if(pNode->right)
                stack.push(pNode->right);
            }
        }
        return vector;
    }
};
```

Method 3: Using one stack and will not changed the node. Best(at least in this three solutions)

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vector;
        stack<TreeNode *> stack;
        TreeNode *pCurrent = root;

        while(!stack.empty() || pCurrent)
        {
            if(pCurrent)
            {
                stack.push(pCurrent);
                pCurrent = pCurrent->left;
            }
            else
            {
                TreeNode *pNode = stack.top();
                vector.push_back(pNode->val);
                stack.pop();
                pCurrent = pNode->right;
            }
        }
        return vector;
    }
};
```

written by KaiChen original link here

Answer 3

Hi, this is a fundamental and yet classic problem. I share my three solutions here:

1. Iterative solution using stack --- `O(n)` time and `O(n)` space;
2. Recursive solution --- `O(n)` time and `O(n)` space (considering the spaces of function call stack);
3. **Morris traversal --- `O(n)` time and `O(1)` space!!!**

Iterative solution using stack:

```
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> nodes;
    stack<TreeNode*> toVisit;
    TreeNode* curNode = root;
    while (curNode || !toVisit.empty()) {
        if (curNode) {
            toVisit.push(curNode);
            curNode = curNode -> left;
        }
        else {
            curNode = toVisit.top();
            toVisit.pop();
            nodes.push_back(curNode -> val);
            curNode = curNode -> right;
        }
    }
    return nodes;
}
```

Recursive solution:

```
void inorder(TreeNode* root, vector<int>& nodes) {
    if (!root) return;
    inorder(root -> left, nodes);
    nodes.push_back(root -> val);
    inorder(root -> right, nodes);
}
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> nodes;
    inorder(root, nodes);
    return nodes;
}
```

Morris traversal:

```
vector<int> inorderTraversal(TreeNode* root) {
    TreeNode* curNode = root;
    vector<int> nodes;
    while (curNode) {
        if (curNode -> left) {
            TreeNode* predecessor = curNode -> left;
            while (predecessor -> right && predecessor -> right != curNode)
                predecessor = predecessor -> right;
            if (!(predecessor -> right)) {
                predecessor -> right = curNode;
                curNode = curNode -> left;
            }
            else {
                predecessor -> right = NULL;
                nodes.push_back(curNode -> val);
                curNode = curNode -> right;
            }
        }
        else {
            nodes.push_back(curNode -> val);
            curNode = curNode -> right;
        }
    }
    return nodes;
}
```

written by jianchao.li.fighter original link here

Unique Binary Search Trees II(95)

Answer 1

I start by noting that 1..n is the in-order traversal for any BST with nodes 1 to n. So if I pick i-th node as my root, the left subtree will contain elements 1 to (i-1), and the right subtree will contain elements (i+1) to n. I use recursive calls to get back all possible trees for left and right subtrees and combine them in all possible ways with the root.

```
public class Solution {
    public List<TreeNode> generateTrees(int n) {

        return genTrees(1,n);
    }

    public List<TreeNode> genTrees (int start, int end)
    {

        List<TreeNode> list = new ArrayList<TreeNode>();

        if(start>end)
        {
            list.add(null);
            return list;
        }

        if(start == end){
            list.add(new TreeNode(start));
            return list;
        }

        List<TreeNode> left,right;
        for(int i=start;i<=end;i++)
        {

            left = genTrees(start, i-1);
            right = genTrees(i+1,end);

            for(TreeNode lnode: left)
            {
                for(TreeNode rnode: right)
                {
                    TreeNode root = new TreeNode(i);
                    root.left = lnode;
                    root.right = rnode;
                    list.add(root);
                }
            }

        }

        return list;
    }
}
```

written by Jayanta original link here

Answer 2

Here is my java solution with DP:

```
public class Solution {
    public static List<TreeNode> generateTrees(int n) {
        List<TreeNode>[] result = new List[n+1];
        result[0] = new ArrayList<TreeNode>();
        result[0].add(null);

        for(int len = 1; len <= n; len++){
            result[len] = new ArrayList<TreeNode>();
            for(int j=0; j<len; j++){
                for(TreeNode nodeL : result[j]){
                    for(TreeNode nodeR : result[len-j-1]){
                        TreeNode node = new TreeNode(j+1);
                        node.left = nodeL;
                        node.right = clone(nodeR, j+1);
                        result[len].add(node);
                    }
                }
            }
        }
        return result[n];
    }

    private static TreeNode clone(TreeNode n, int offset){
        if(n == null)
            return null;
        TreeNode node = new TreeNode(n.val + offset);
        node.left = clone(n.left, offset);
        node.right = clone(n.right, offset);
        return node;
    }
}
```

result[i] stores the result until length **i**. For the result for length i+1, select the root node j from 0 to i, combine the result from left side and right side. Note for the right side we have to clone the nodes as the value will be offsetted by **j**.

written by jianwu original link here

Answer 3

This problem is a variant of the problem of Unique Binary Search Trees.

I provided a solution along with explanation for the above problem, in the question "DP solution in 6 lines with explanation"

It is intuitive to solve this problem by following the same algorithm. Here is the code in a divide-and-conquer style.

```java
public List<TreeNode> generateTrees(int n) {
    return generateSubtrees(1, n);
}

private List<TreeNode> generateSubtrees(int s, int e) {
    List<TreeNode> res = new LinkedList<TreeNode>();
    if (s > e) {
        res.add(null); // empty tree
        return res;
    }

    for (int i = s; i <= e; ++i) {
        List<TreeNode> leftSubtrees = generateSubtrees(s, i - 1);
        List<TreeNode> rightSubtrees = generateSubtrees(i + 1, e);

        for (TreeNode left : leftSubtrees) {
            for (TreeNode right : rightSubtrees) {
                TreeNode root = new TreeNode(i);
                root.left = left;
                root.right = right;
                res.add(root);
            }
        }
    }
    return res;
}
```

written by liaison original link here

Unique Binary Search Trees(96)

Answer 1

The problem can be solved in a dynamic programming way. Iâ€™ll explain the intuition and formulas in the following.

Given a sequence 1â€¦n, to construct a Binary Search Tree (BST) out of the sequence, we could enumerate each number i in the sequence, and use the number as the root, naturally, the subsequence 1â€¦(i-1) on its left side would lay on the left branch of the root, and similarly the right subsequence (i+1)â€¦n lay on the right branch of the root. We then can construct the subtree from the subsequence recursively. Through the above approach, we could ensure that the BST that we construct are all unique, since they have unique roots.

The problem is to calculate the number of unique BST. To do so, we need to define two functions:

`G(n)` : the number of unique BST for a sequence of length n.

`F(i, n), 1 <= i <= n` : the number of unique BST, where the number i is the root of BST, and the sequence ranges from 1 to n.

As one can see, `G(n)` is the actual function we need to calculate in order to solve the problem. And `G(n)` can be derived from `F(i, n)`, which at the end, would recursively refer to `G(n)`.

First of all, given the above definitions, we can see that the total number of unique BST `G(n)`, is the sum of BST `F(i)` using each number i as a root. *i.e.*

```
G(n) = F(1, n) + F(2, n) + ... + F(n, n).
```

Particularly, the bottom cases, there is only one combination to construct a BST out of a sequence of length 1 (only a root) or 0 (empty tree). *i.e.*

```
G(0)=1, G(1)=1.
```

Given a sequence 1â€¦n, we pick a number i out of the sequence as the root, then the number of unique BST with the specified root `F(i)`, is the cartesian product of the number of BST for its left and right subtrees. For example, `F(3, 7)`: the number of unique BST tree with number 3 as its root. To construct an unique BST out of the entire sequence [1, 2, 3, 4, 5, 6, 7] with 3 as the root, which is to say, we need to construct an unique BST out of its left subsequence [1, 2] and another BST out of the right subsequence [4, 5, 6, 7], and then combine them together (*i.e.* cartesian product). The tricky part is that we could consider the number of unique BST out of sequence [1,2] as `G(2)`, and the number of of unique BST out of sequence [4, 5, 6, 7] as `G(4)`. Therefore, `F(3,7) = G(2) * G(4)`.

i.e.

```
F(i, n) = G(i-1) * G(n-i)   1 <= i <= n
```

Combining the above two formulas, we obtain the recursive formula for `G(n)`. *i.e.*

```
G(n) = G(0) * G(n-1) + G(1) * G(n-2) + â€¦ + G(n-1) * G(0)
```

In terms of calculation, we need to start with the lower number, since the value of `G(n)` depends on the values of `G(0) â€¦ G(n-1)`.

With the above explanation and formulas, here is the implementation in Java.

```
public int numTrees(int n) {
    int [] G = new int[n+1];
    G[0] = G[1] = 1;

    for(int i=2; i<=n; ++i) {
        for(int j=1; j<=i; ++j) {
            G[i] += G[j-1] * G[i-j];
        }
    }

    return G[n];
}
```

written by liaison original link here

Answer 2

```
/**
 * Taking 1~n as root respectively:
 *      1 as root: # of trees = F(0) * F(n-1)  // F(0) == 1
 *      2 as root: # of trees = F(1) * F(n-2)
 *      3 as root: # of trees = F(2) * F(n-3)
 *      ...
 *      n-1 as root: # of trees = F(n-2) * F(1)
 *      n as root:   # of trees = F(n-1) * F(0)
 *
 * So, the formulation is:
 *      F(n) = F(0) * F(n-1) + F(1) * F(n-2) + F(2) * F(n-3) + ... + F(n-2) * F(1
) + F(n-1) * F(0)
 */

int numTrees(int n) {
    int dp[n+1];
    dp[0] = dp[1] = 1;
    for (int i=2; i<=n; i++) {
        dp[i] = 0;
        for (int j=1; j<=i; j++) {
            dp[i] += dp[j-1] * dp[i-j];
        }
    }
    return dp[n];
}
```

written by leo.mao original link here

Answer 3

WE can know that by zero we can have 1 bst of null by 1 we have 1 bst of 1 and for 2 we can arrange using two ways Now idea is simple for rest numbers. for n=3 make 1 as root node so there will be 0 nodes in left subtree and 2 nodes in right subtree. we know the solution for 2 nodes that they can be used to make 2 bsts. Now making 2 as the root node , there will be 1 in left subtree and 1 node in right subtree. ! node results in 1 way for making a BST. Now making 3 as root node. There will be 2 nodes in left subtree and 0 nodes in right subtree. We know 2 will give 2 BST and zero will give 1 BST. Totalling the result of all the 3 nodes as root will give 5. Same process can be applied for more numbers.

```
    public int number(int n){
    if(n==0)return 1;
        if(n==1)return 1;

        int result[]=new int [n+1];
        result[0]=1;
        result[1]=1;
        result[2]=2;
        if(n<3){
            return result[n];
        }

        for(int i=3;i<=n;i++){
            for(int k=1;k<=i;k++){

                result[i]=result[i]+result[k-1]*result[i-k];
            }
        }


        return result[n];
    }
```

written by ajak6 original link here

Interleaving String(97)

Answer 1

```
 bool isInterleave(string s1, string s2, string s3) {

    if(s3.length() != s1.length() + s2.length())
        return false;

    bool table[s1.length()+1][s2.length()+1];

    for(int i=0; i<s1.length()+1; i++)
        for(int j=0; j< s2.length()+1; j++){
            if(i==0 && j==0)
                table[i][j] = true;
            else if(i == 0)
                table[i][j] = ( table[i][j-1] && s2[j-1] == s3[i+j-1]);
            else if(j == 0)
                table[i][j] = ( table[i-1][j] && s1[i-1] == s3[i+j-1]);
            else
                table[i][j] = (table[i-1][j] && s1[i-1] == s3[i+j-1] ) || (table[
i][j-1] && s2[j-1] == s3[i+j-1] );
        }

    return table[s1.length()][s2.length()];
}
```

Here is some explanation:

DP table represents if s3 is interleaving at (i+j)th position when s1 is at ith position, and s2 is at jth position. 0th position means empty string.

So if both s1 and s2 is currently empty, s3 is empty too, and it is considered interleaving. If only s1 is empty, then if previous s2 position is interleaving and current s2 position char is equal to s3 current position char, it is considered interleaving. similar idea applies to when s2 is empty. when both s1 and s2 is not empty, then if we arrive i, j from i-1, j, then if i-1,j is already interleaving and i and current s3 position equal, it s interleaving. If we arrive i,j from i, j-1, then if i, j-1 is already interleaving and j and current s3 position equal. it is interleaving.

written by sherryxmhe original link here

Answer 2

If we expand the two strings s1 and s2 into a chessboard, then this problem can be transferred into a path seeking problem from the top-left corner to the bottom-right corner. The key is, each cell (y, x) in the board corresponds to an interval between y-th character in s1 and x-th character in s2. And adjacent cells are connected with like a grid. A BFS can then be efficiently performed to find the path.

Better to illustrate with an example here:

Say s1 = "aab" and s2 = "abc". s3 = "aaabcb". Then the board looks like

```
o--a--o--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--o--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--o--b--o--c--o
|     |     |     |
b     b     b     b
|     |     |     |
o--a--o--b--o--c--o
```

Each "o" is a cell in the board. We start from the top-left corner, and try to move right or down. If the next char in s3 matches the edge connecting the next cell, then we're able to move. When we hit the bottom-right corner, this means s3 can be represented by interleaving s1 and s2. One possible path for this example is indicated with "x"es below:

```
x--a--x--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--x--b--o--c--o
|     |     |     |
a     a     a     a
|     |     |     |
o--a--x--b--x--c--x
|     |     |     |
b     b     b     b
|     |     |     |
o--a--o--b--o--c--x
```

Note if we concatenate the chars on the edges we went along, it's exactly s3. And we went through all the chars in s1 and s2, in order, exactly once.

Therefore if we view this board as a graph, such path finding problem is trivial with BFS. I use an `unordered_map` to store the visited nodes, which makes the code look a bit complicated. But a `vector` should be enough to do the job.

Although the worse case timeis also O(mn), typically it doesn't require us to go through every node to find a path. Therefore it's faster than regular DP than average.

```
struct MyPoint {
    int y, x;
    bool operator==(const MyPoint &p) const {
        return p.y == y && p.x == x;
    }
};
namespace std {
    template <>
    struct hash<MyPoint> {
        size_t operator () (const MyPoint &f) const {
            return (std::hash<int>()(f.x) << 1) ^ std::hash<int>()(f.y);
        }
    };
}

class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;

        queue<MyPoint> q;
        unordered_set<MyPoint> visited;
        bool isSuccessful = false;
        int i = 0;

        q.push(MyPoint { 0, 0 });
        q.push(MyPoint { -1, -1 });
        while (!(1 == q.size() && -1 == q.front().x)) {
            auto p = q.front();
            q.pop();
            if (p.y == s1.size() && p.x == s2.size()) {
                return true;
            }
            if (-1 == p.y) {
                q.push(p);
                i++;
                continue;
            }
            if (visited.find(p) != visited.end()) { continue; }
            visited.insert(p);

            if (p.y < s1.size()) { // down
                if (s1[p.y] == s3[i]) { q.push(MyPoint { p.y + 1, p.x }); }
            }
            if (p.x < s2.size()) { // right
                if (s2[p.x] == s3[i]) { q.push(MyPoint { p.y, p.x + 1 }); }
            }
        }
        return false;
    }
};
```

written by grapeot original link here

Answer 3

```
public boolean isInterleave(String s1, String s2, String s3) {

    if ((s1.length()+s2.length())!=s3.length()) return false;

    boolean[][] matrix = new boolean[s2.length()+1][s1.length()+1];

    matrix[0][0] = true;

    for (int i = 1; i < matrix[0].length; i++){
        matrix[0][i] = matrix[0][i-1]&&(s1.charAt(i-1)==s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        matrix[i][0] = matrix[i-1][0]&&(s2.charAt(i-1)==s3.charAt(i-1));
    }

    for (int i = 1; i < matrix.length; i++){
        for (int j = 1; j < matrix[0].length; j++){
            matrix[i][j] = (matrix[i-1][j]&&(s2.charAt(i-1)==s3.charAt(i+j-1)))
                    || (matrix[i][j-1]&&(s1.charAt(j-1)==s3.charAt(i+j-1)));
        }
    }

    return matrix[s2.length()][s1.length()];

}
```

written by ECLAT original link here

Validate Binary Search Tree(98)

Answer 1

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode* prev = NULL;
        return validate(root, prev);
    }
    bool validate(TreeNode* node, TreeNode* &prev) {
        if (node == NULL) return true;
        if (!validate(node->left, prev)) return false;
        if (prev != NULL && prev->val >= node->val) return false;
        prev = node;
        return validate(node->right, prev);
    }
};
```

> Update:
>
> If we use in-order traversal to serialize a binary search tree, we can get a list of values in ascending order. It can be proved with the definition of BST. And here I use the reference of TreeNode pointer `prev` as a global variable to mark the address of previous node in the list.
>
> â€œIn-order Traversalâ€: https://en.wikipedia.org/wiki/Tree_traversal#In-order

If you know what `INT_MAX` or `INT_MIN` is, then it is no excuse for your carelessness.

written by jakwings original link here

Answer 2

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root, long minVal, long maxVal) {
        if (root == null) return true;
        if (root.val >= maxVal || root.val <= minVal) return false;
        return isValidBST(root.left, minVal, root.val) && isValidBST(root.right,
root.val, maxVal);
    }
}
```

Basically what I am doing is recursively iterating over the tree while defining interval `<minVal, maxVal>` for each node which value must fit in.

written by sruzic original link here

Answer 3

```
bool isValidBST(TreeNode* root) {
    return isValidBST(root, NULL, NULL);
}

bool isValidBST(TreeNode* root, TreeNode* minNode, TreeNode* maxNode) {
    if(!root) return true;
    if(minNode && root->val <= minNode->val || maxNode && root->val >= maxNode->va
l)
        return false;
    return isValidBST(root->left, minNode, root) && isValidBST(root->right, root,
maxNode);
}
```

written by jaewoo original link here

Recover Binary Search Tree(99)

Answer 1

This question appeared difficult to me but it is really just a simple in-order traversal! I got really frustrated when other people are showing off Morris Traversal which is totally not necessary here.

Let's start by writing the in order traversal:

```
private void traverse (TreeNode root) {
   if (root == null)
      return;
   traverse(root.left);
   // Do some business
   traverse(root.right);
}
```

So when we need to print the node values in order, we insert System.out.println(root.val) in the place of "Do some business".

What is the business we are doing here? We need to find the first and second elements that are not in order right?

How do we find these two elements? For example, we have the following tree that is printed as in order traversal:

6, 3, 4, 5, 2

We compare each node with its next one and we can find out that 6 is the first element to swap because 6 > 3 and 2 is the second element to swap because 2 < 5.

Really, what we are comparing is the current node and its previous node in the "in order traversal".

Let us define three variables, firstElement, secondElement, and prevElement. Now we just need to build the "do some business" logic as finding the two elements. See the code below:

```
public class Solution {

    TreeNode firstElement = null;
    TreeNode secondElement = null;
    // The reason for this initialization is to avoid null pointer exception in th
e first comparison when prevElement has not been initialized
    TreeNode prevElement = new TreeNode(Integer.MIN_VALUE);

    public void recoverTree(TreeNode root) {

        // In order traversal to find the two elements
        traverse(root);

        // Swap the values of the two nodes
        int temp = firstElement.val;
        firstElement.val = secondElement.val;
        secondElement.val = temp;
    }

    private void traverse(TreeNode root) {

        if (root == null)
            return;

        traverse(root.left);

        // Start of "do some business",
        // If first element has not been found, assign it to prevElement (refer t
o 6 in the example above)
        if (firstElement == null && prevElement.val >= root.val) {
            firstElement = prevElement;
        }

        // If first element is found, assign the second element to the root (refe
r to 2 in the example above)
        if (firstElement != null && prevElement.val >= root.val) {
            secondElement = root;
        }
        prevElement = root;

        // End of "do some business"

        traverse(root.right);
}
```

And we are done, it is just that easy!

written by qwl5004 original link here

Answer 2

To understand this, you need to first understand Morris Traversal or Morris Threading Traversal. It take use of leaf nodes' right/left pointer to achieve O(1) space Traversal on a Binary Tree. Below is a standard Inorder Morris Traversal, referred from

http://www.cnblogs.com/AnnieKim/archive/2013/06/15/morristraversal.html (a Chinese Blog, while the graphs are great for illustration)

```
public void morrisTraversal(TreeNode root){
        TreeNode temp = null;
        while(root!=null){
            if(root.left!=null){
                // connect threading for root
                temp = root.left;
                while(temp.right!=null && temp.right != root)
                    temp = temp.right;
                // the threading already exists
                if(temp.right!=null){
                    temp.right = null;
                    System.out.println(root.val);
                    root = root.right;
                }else{
                    // construct the threading
                    temp.right = root;
                    root = root.left;
                }
            }else{
                System.out.println(root.val);
                root = root.right;
            }
        }
    }
```

In the above code, `System.out.println(root.val);` appear twice, which functions as outputing the Node in ascending order (BST). Since these places are in order, replace them with

```
    if(pre!=null && pre.val > root.val){
        if(first==null){first = pre;second = root;}
        else{second = root;}
  }
pre = root;
```

each time, the pre node and root are in order as `System.out.println(root.val);` outputs them in order.

Then, come to how to specify the first wrong node and second wrong node.

When they are not consecutive, the first time we meet `pre.val > root.val` ensure us the first node is the pre node, since root should be traversal ahead of pre, pre should be at least at small as root. The second time we meet `pre.val > root.val` ensure us the second node is the root node, since we are now looking for a node to replace with out first node, which is found before.

When they are consecutive, which means the case `pre.val > cur.val` will appear only once. We need to take case this case without destroy the previous analysis. So the first node will still be pre, and the second will be just set to root. Once we meet

this case again, the first node will not be affected.

Below is the updated version on Morris Traversal.

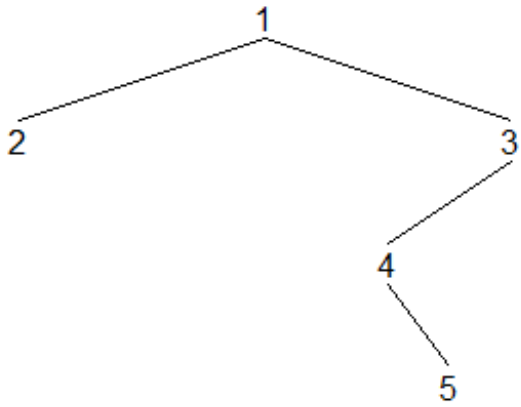
```
public void recoverTree(TreeNode root) {
        TreeNode pre = null;
        TreeNode first = null, second = null;
        // Morris Traversal
        TreeNode temp = null;
        while(root!=null){
            if(root.left!=null){
                // connect threading for root
                temp = root.left;
                while(temp.right!=null && temp.right != root)
                    temp = temp.right;
                // the threading already exists
                if(temp.right!=null){
                    if(pre!=null && pre.val > root.val){
                        if(first==null){first = pre;second = root;}
                        else{second = root;}
                    }
                    pre = root;

                    temp.right = null;
                    root = root.right;
                }else{
                    // construct the threading
                    temp.right = root;
                    root = root.left;
                }
            }else{
                if(pre!=null && pre.val > root.val){
                    if(first==null){first = pre;second = root;}
                    else{second = root;}
                }
                pre = root;
                root = root.right;
            }
        }
        // swap two node values;
        if(first!= null && second != null){
            int t = first.val;
            first.val = second.val;
            second.val = t;
        }
    }
```

written by siyang3 original link here

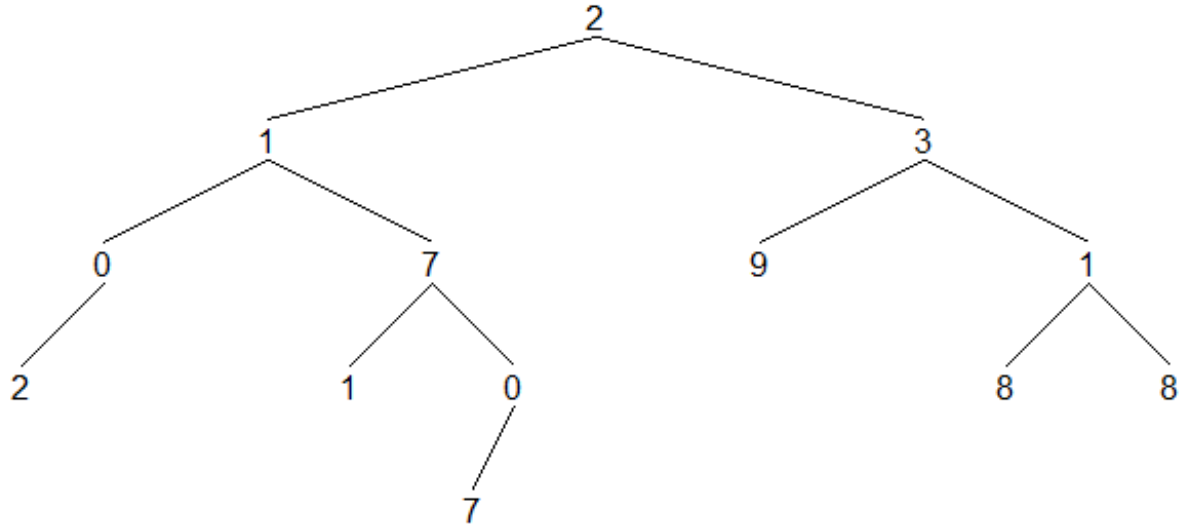
Answer 3

Wrote some tools for my own local testing. For example `deserialize('[1,2,3,null,null,4,null,null,5]')` will turn that into a tree and return the root as explained in the FAQ. I also wrote a visualizer. Two examples:

`drawtree(deserialize('[1,2,3,null,null,4,null,null,5]'))`:



`drawtree(deserialize('[2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,null,7]'))`:



Here's the code. If you save it as a Python script and run it, it should as a demo show the above two pictures in turtle windows (one after the other). And you can of course import it from other scripts and then it will only provide the class/functions and not show the demo.

```
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
    def __repr__(self):
        return 'TreeNode({})'.format(self.val)

def deserialize(string):
    if string == '{}':
        return None
    nodes = [None if val == 'null' else TreeNode(int(val))
             for val in string.strip('[]{}').split(',')]
    kids = nodes[::-1]
    root = kids.pop()
    for node in nodes:
        if node:
            if kids: node.left  = kids.pop()
            if kids: node.right = kids.pop()
    return root

def drawtree(root):
    def height(root):
        return 1 + max(height(root.left), height(root.right)) if root else -1
    def jumpto(x, y):
        t.penup()
        t.goto(x, y)
        t.pendown()
    def draw(node, x, y, dx):
        if node:
            t.goto(x, y)
            jumpto(x, y-20)
            t.write(node.val, align='center', font=('Arial', 12, 'normal'))
            draw(node.left, x-dx, y-60, dx/2)
            jumpto(x, y-20)
            draw(node.right, x+dx, y-60, dx/2)
    import turtle
    t = turtle.Turtle()
    t.speed(0); turtle.delay(0)
    h = height(root)
    jumpto(0, 30*h)
    draw(root, 0, 30*h, 40*h)
    t.hideturtle()
    turtle.mainloop()

if __name__ == '__main__':
    drawtree(deserialize('[1,2,3,null,null,4,null,null,5]'))
    drawtree(deserialize('[2,1,3,0,7,9,1,2,null,1,0,null,null,8,8,null,null,null,
null,7]'))
```

written by StefanPochmann original link here

Same Tree(100)

Answer 1

```
public boolean isSameTree(TreeNode p, TreeNode q) {
    if(p == null && q == null) return true;
    if(p == null || q == null) return false;
    if(p.val == q.val)
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    return false;
}
```

written by micheal.zhou original link here

Answer 2

```
//
// Algorithm for the recursion:
// 1)
// If one of the node is NULL then return the equality result of p an q.
// This boils down to if both are NULL then return true,
// but if one of them is NULL but not the other one then return false
// 2)
// At this point both root nodes represent valid pointers.
// Return true if the root nodes have same value and
// the left tree of the roots are same (recursion)
// and the right tree of the roots are same (recursion).
// Otherwise return false.
//

bool isSameTree(TreeNode *p, TreeNode *q) {
    if (p == NULL || q == NULL) return (p == q);
    return (p->val == q->val && isSameTree(p->left, q->left) && isSameTree(p->righ
t, q->right));
}
```

written by satyakam original link here

Answer 3

the idea is to use stack for preorder traverse

```java
public boolean isSameTree(TreeNode p, TreeNode q) {
        Stack<TreeNode> stack_p = new Stack <> ();
        Stack<TreeNode> stack_q = new Stack <> ();
        if (p != null) stack_p.push( p ) ;
        if (q != null) stack_q.push( q ) ;
        while (!stack_p.isEmpty() && !stack_q.isEmpty()) {
            TreeNode pn = stack_p.pop() ;
            TreeNode qn = stack_q.pop() ;
            if (pn.val != qn.val) return false ;
            if (pn.right != null) stack_p.push(pn.right) ;
            if (qn.right != null) stack_q.push(qn.right) ;
            if (stack_p.size() != stack_q.size()) return false ;
            if (pn.left != null) stack_p.push(pn.left) ;
            if (qn.left != null) stack_q.push(qn.left) ;
            if (stack_p.size() != stack_q.size()) return false ;
        }
        return stack_p.size() == stack_q.size() ;
    }
```

written by scott original link here

Symmetric Tree(101)

Answer 1

Recursive--400ms:

```
public boolean isSymmetric(TreeNode root) {
    return root==null || isSymmetricHelp(root.left, root.right);
}

private boolean isSymmetricHelp(TreeNode left, TreeNode right){
    if(left==null || right==null)
        return left==right;
    if(left.val!=right.val)
        return false;
    return isSymmetricHelp(left.left, right.right) && isSymmetricHelp(left.right,
right.left);
}
```

Non-recursive(use Stack)--460ms:

```java
public boolean isSymmetric(TreeNode root) {
    if(root==null)  return true;

    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode left, right;
    if(root.left!=null){
        if(root.right==null) return false;
        stack.push(root.left);
        stack.push(root.right);
    }
    else if(root.right!=null){
        return false;
    }

    while(!stack.empty()){
        if(stack.size()%2!=0)   return false;
        right = stack.pop();
        left = stack.pop();
        if(right.val!=left.val) return false;

        if(left.left!=null){
            if(right.right==null)   return false;
            stack.push(left.left);
            stack.push(right.right);
        }
        else if(right.right!=null){
            return false;
        }

        if(left.right!=null){
            if(right.left==null)   return false;
            stack.push(left.right);
            stack.push(right.left);
        }
        else if(right.left!=null){
            return false;
        }
    }

    return true;
}
```

written by lvlolitte original link here

Answer 2

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode *root) {
        TreeNode *left, *right;
        if (!root)
            return true;

        queue<TreeNode*> q1, q2;
        q1.push(root->left);
        q2.push(root->right);
        while (!q1.empty() && !q2.empty()){
            left = q1.front();
            q1.pop();
            right = q2.front();
            q2.pop();
            if (NULL == left && NULL == right)
                continue;
            if (NULL == left || NULL == right)
                return false;
            if (left->val != right->val)
                return false;
            q1.push(left->left);
            q1.push(left->right);
            q2.push(right->right);
            q2.push(right->left);
        }
        return true;
    }
};
```

written by JayfonLin original link here

Answer 3

```
bool isSymmetric(TreeNode *root) {
        if (!root) return true;
        return helper(root->left, root->right);
    }

    bool helper(TreeNode* p, TreeNode* q) {
        if (!p && !q) {
            return true;
        } else if (!p || !q) {
            return false;
        }

        if (p->val != q->val) {
            return false;
        }

        return helper(p->left,q->right) && helper(p->right, q->left);
    }
```

written by pankit original link here

Binary Tree Level Order Traversal(102)

Answer 1

```
vector<vector<int>> ret;

void buildVector(TreeNode *root, int depth)
{
    if(root == NULL) return;
    if(ret.size() == depth)
        ret.push_back(vector<int>());

    ret[depth].push_back(root->val);
    buildVector(root->left, depth + 1);
    buildVector(root->right, depth + 1);
}

vector<vector<int> > levelOrder(TreeNode *root) {
    buildVector(root, 0);
    return ret;
}
```

written by nilath original link here

Answer 2

```
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        List<List<Integer>> wrapList = new LinkedList<List<Integer>>();

        if(root == null) return wrapList;

        queue.offer(root);
        while(!queue.isEmpty()){
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for(int i=0; i<levelNum; i++) {
                if(queue.peek().left != null) queue.offer(queue.peek().left);
                if(queue.peek().right != null) queue.offer(queue.peek().right);
                subList.add(queue.poll().val);
            }
            wrapList.add(subList);
        }
        return wrapList;
    }
}
```

written by SOY original link here

Answer 3

```
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> >  result;
        if (!root) return result;
        queue<TreeNode*> q;
        q.push(root);
        q.push(NULL);
        vector<int> cur_vec;
        while(!q.empty()) {
            TreeNode* t = q.front();
            q.pop();
            if (t==NULL) {
                result.push_back(cur_vec);
                cur_vec.resize(0);
                if (q.size() > 0) {
                    q.push(NULL);
                }
            } else {
                cur_vec.push_back(t->val);
                if (t->left) q.push(t->left);
                if (t->right) q.push(t->right);
            }
        }
        return result;
    }
};
```

written by pankit original link here

Binary Tree Zigzag Level Order Traversal(103)

Answer 1

```
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root)
    {
        List<List<Integer>> sol = new ArrayList<>();
        travel(root, sol, 0);
        return sol;
    }

    private void travel(TreeNode curr, List<List<Integer>> sol, int level)
    {
        if(curr == null) return;

        if(sol.size() <= level)
        {
            List<Integer> newLevel = new LinkedList<>();
            sol.add(newLevel);
        }

        List<Integer> collection  = sol.get(level);
        if(level % 2 == 0) collection.add(curr.val);
        else collection.add(0, curr.val);

        travel(curr.left, sol, level + 1);
        travel(curr.right, sol, level + 1);
    }
}
```

1. O(n) solution by using LinkedList along with ArrayList. So insertion in the inner list and outer list are both O(1),
2. Using DFS and creating new lists when needed.

should be quite straightforward. any better answer?

written by wayne.s.lu original link here

Answer 2

Assuming after traversing the 1st level, nodes in queue are {9, 20, 8}, And we are going to traverse 2nd level, which is even line and should print value from right to left [8, 20, 9].

We know there are 3 nodes in current queue, so the vector for this level in final result should be of size 3. Then, queue [i] -> goes to -> vector[queue.size() - 1 - i] i.e. the ith node in current queue should be placed in (queue.size() - 1 - i) position in vector for that line.

For example, for node(9), it's index in queue is 0, so its index in vector should be (3-1-0) = 2.

```
vector<vector<int> > zigzagLevelOrder(TreeNode* root) {
    if (root == NULL) {
        return vector<vector<int> > ();
    }
    vector<vector<int> > result;

    queue<TreeNode*> nodesQueue;
    nodesQueue.push(root);
    bool leftToRight = true;

    while ( !nodesQueue.empty()) {
        int size = nodesQueue.size();
        vector<int> row(size);
        for (int i = 0; i < size; i++) {
            TreeNode* node = nodesQueue.front();
            nodesQueue.pop();

            // find position to fill node's value
            int index = (leftToRight) ? i : (size - 1 - i);

            row[index] = node->val;
            if (node->left) {
                nodesQueue.push(node->left);
            }
            if (node->right) {
                nodesQueue.push(node->right);
            }
        }
        // after this level
        leftToRight = !leftToRight;
        result.push_back(row);
    }
    return result;
}
```

written by StevenCooks original link here

Answer 3

Simple algorithm:

1. do depth first recursive tree search
2. populate all vectors for each tree level from left to right
3. reverse even levels to conform with zigzar requirement

.

```
class Solution {
vector<vector<int> > result;
public:
vector<vector<int> > zigzagLevelOrder(TreeNode *root) {

    if(root!=NULL)
    {
        traverse(root, 0);
    }

    for(int i=1;i<result.size();i+=2)
    {
        vector<int>* v = &result[i];
        std:reverse(v->begin(), v->end());
    }
    return result;
}

void traverse(TreeNode* node, int level)
{
    if(node == NULL) return;

    vector<int>* row = getRow(level);
    row->push_back(node->val);

    traverse(node->left, level+1);
    traverse(node->right, level+1);
}

vector<int>* getRow(int level)
{
    if(result.size()<=level)
    {
        vector<int> newRow;
        result.push_back(newRow);
    }
    return &result[level];
}
};
```

written by paul7 original link here

Maximum Depth of Binary Tree(104)

Answer 1

After a solution is accepted it would be very helpful to know how to make it run faster looking at better performing solution(s).

written by rainhacker original link here

Answer 2

if the node does not exist, simply return 0. Otherwise, return the 1+the longer distance of its subtree.

```
public int maxDepth(TreeNode root) {
        if(root==null){
            return 0;
        }
        return 1+Math.max(maxDepth(root.left),maxDepth(root.right));
    }
```

written by ray050899 original link here

Answer 3

1. Depth-first-search

Only one line code.

```
int maxDepth(TreeNode *root)
{
    return root == NULL ? 0 : max(maxDepth(root -> left), maxDepth(root -> right)
) + 1;
}
```

2. Breadth-first-search

Calculate the count of the last level.

```
int maxDepth(TreeNode *root)
{
    if(root == NULL)
        return 0;

    int res = 0;
    queue<TreeNode *> q;
    q.push(root);
    while(!q.empty())
    {
        ++ res;
        for(int i = 0, n = q.size(); i < n; ++ i)
        {
            TreeNode *p = q.front();
            q.pop();

            if(p -> left != NULL)
                q.push(p -> left);
            if(p -> right != NULL)
                q.push(p -> right);
        }
    }

    return res;
}
```

written by makuiyu original link here

Construct Binary Tree from Preorder and Inorder Traversal(105)

Answer 1

I din't find iterative solutions discussed in the old Discuss. So, I thought, I will add my solution in here.

The idea is as follows:

1) Keep pushing the nodes from the preorder into a stack (and keep making the tree by adding nodes to the left of the previous node) until the top of the stack matches the inorder.

2) At this point, pop the top of the stack until the top does not equal inorder (keep a flag to note that you have made a pop).

3) Repeat 1 and 2 until preorder is empty. The key point is that whenever the flag is set, insert a node to the right and reset the flag.

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {

        if(preorder.size()==0)
            return NULL;

        stack<int> s;
        stack<TreeNode *> st;
        TreeNode *t,*r,*root;
        int i,j,f;

        f=i=j=0;
        s.push(preorder[i]);

        root = new TreeNode(preorder[i]);
        st.push(root);
        t = root;
        i++;

        while(i<preorder.size())
        {
            if(!st.empty() && st.top()->val==inorder[j])
            {
                t = st.top();
                st.pop();
                s.pop();
                f = 1;
                j++;
            }
            else
            {
                if(f==0)
                {
                    s.push(preorder[i]);
                    t -> left = new TreeNode(preorder[i]);
                    t = t -> left;
                    st.push(t);
                    i++;
                }
                else
                {
                    f = 0;
                    s.push(preorder[i]);
                    t -> right = new TreeNode(preorder[i]);
                    t = t -> right;
                    st.push(t);
                    i++;
                }
            }
        }

        return root;
    }
};
```

written by gpraveenkumar original link here

Answer 2

Hi guys, this is my Java solution. I read this post, which is very helpful.

The basic idea is here: Say we have 2 arrays, PRE and IN. Preorder traversing implies that PRE[0] is the root node. Then we can find this PRE[0] in IN, say it's IN[5]. Now we know that IN[5] is root, so we know that IN[0] - IN[4] is on the left side, IN[6] to the end is on the right side. Recursively doing this on subarrays, we can build a tree out of it :)

Hope this helps.

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
    return helper(0, 0, inorder.length - 1, preorder, inorder);
}

public TreeNode helper(int preStart, int inStart, int inEnd, int[] preorder, int[]
inorder) {
    if (preStart > preorder.length - 1 || inStart > inEnd) {
        return null;
    }
    TreeNode root = new TreeNode(preorder[preStart]);
    int inIndex = 0; // Index of current root in inorder
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == root.val) {
            inIndex = i;
        }
    }
    root.left = helper(preStart + 1, inStart, inIndex - 1, preorder, inorder);
    root.right = helper(preStart + inIndex - inStart + 1, inIndex + 1, inEnd, pre
order, inorder);
    return root;
}
```

written by jiaming2 original link here

Answer 3

```
TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
    return create(preorder, inorder, 0, preorder.size() - 1, 0, inorder.size() -
1);
}

TreeNode* create(vector<int>& preorder, vector<int>& inorder, int ps, int pe, int
is, int ie){
    if(ps > pe){
        return nullptr;
    }
    TreeNode* node = new TreeNode(preorder[ps]);
    int pos;
    for(int i = is; i <= ie; i++){
        if(inorder[i] == node->val){
            pos = i;
            break;
        }
    }
    node->left = create(preorder, inorder, ps + 1, ps + pos - is, is, pos - 1);
    node->right = create(preorder, inorder, pe - ie + pos + 1, pe, pos + 1, ie);
    return node;
}
```

The first element in preorder array can divide inorder array into two parts. Then we can divide preorder array into two parts. Make this element a node. And the left sub-tree of this node is the left part, right sub-tree of this node is the right part. This problem can be solved following this logic.

written by zxyperfect original link here

Construct Binary Tree from Inorder and Postorder Traversal(106)

Answer 1

Below is the O(n) solution from @hongzhi but that discuss is closed now 'cause @hongzhi says little about his code.

https://oj.leetcode.com/discuss/6334/here-is-my-o-n-solution-is-it-neat

I've modified some of and tried this code and got AC. Just share about some comprehension about his code.

I've modified vtn(vector) to stn(stack) in that **stack** is probably what this algs means and needs.

What matters most is the meaning of *stn*.

Only nodes whoes left side **hasn't been** handled will be pushed into *stn*.

And inorder is organized as (inorder of left) root (inorder of right),

And postorder is as (postorder of left) (postorder of right) root.

So at the very begin, we only have root in stn and we check if *inorder.back() == root->val* and in most cases it's **false**(see Note 1). Then we make this node root's right sub-node and push it into stn.

Note 1: this is actually *(inorder of right).back() == (postorder of right).back()*, so if only there's no right subtree or the answer will always be false.

Note 2: we delete one node from *postorder* as we push one into stn.

Now we have [root, root's right] as stn and we check *inorder.back() == stn.top()->val* again.

- **true** means *inorder.back()* is the root node and needs handled left case.
- **false** means *inorder.back()* is the next right sub-node

So when we encounter a true, we will cache *stn.top()* as p and **delete both nodes from inorder and stn**.

Then we check inorder.size(), if there's no nodes left, it means p has no left node.

Else the next node in inorder could be *p's left node* or *p's father* which equals to the now *stn.top()* (remember we popped *p* from *stn* above).

If the latter happens, it means *p* has **no left node** and we need to move on to *p's father(stn.top())*.

If the former happens, it means *p* has one left node and it's *postorder.back()*, so we put it to p's left and delete it from the *postorder* and push the left node into *stn* 'cause **it** should be the next check node as the *postorder* is organized as above.

That's all of it. The algs just build a binary tree. :)

Inform me if there's anything vague or wrong, I'm open to any suggestions.

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
        if(inorder.size() == 0)return NULL;
        TreeNode *p;
        TreeNode *root;
        stack<TreeNode *> stn;

        root = new TreeNode(postorder.back());
        stn.push(root);
        postorder.pop_back();

        while(true)
        {
            if(inorder.back() == stn.top()->val)
            {
                p = stn.top();
                stn.pop();
                inorder.pop_back();
                if(inorder.size() == 0) break;
                if(stn.size() && inorder.back() == stn.top()->val)
                    continue;
                p->left = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.push(p->left);
            }
            else
            {
                p = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.top()->right = p;
                stn.push(p);
            }
        }
        return root;
    }
};
```

written by Lancelod_Liu original link here

Answer 2

The the basic idea is to take the last element in postorder array as the root, find the position of the root in the inorder array; then locate the range for left sub-tree and right sub-tree and do recursion. Use a HashMap to record the index of root in the inorder array.

```
public TreeNode buildTreePostIn(int[] inorder, int[] postorder) {
    if (inorder == null || postorder == null || inorder.length != postorder.length
)
        return null;
    HashMap<Integer, Integer> hm = new HashMap<Integer,Integer>();
    for (int i=0;i<inorder.length;++i)
        hm.put(inorder[i], i);
    return buildTreePostIn(inorder, 0, inorder.length-1, postorder, 0,
                           postorder.length-1,hm);
}

private TreeNode buildTreePostIn(int[] inorder, int is, int ie, int[] postorder,
int ps, int pe,
                                 HashMap<Integer,Integer> hm){
    if (ps>pe || is>ie) return null;
    TreeNode root = new TreeNode(postorder[pe]);
    int ri = hm.get(postorder[pe]);
    TreeNode leftchild = buildTreePostIn(inorder, is, ri-1, postorder, ps, ps+ri-
is-1, hm);
    TreeNode rightchild = buildTreePostIn(inorder,ri+1, ie, postorder, ps+ri-is,
pe-1, hm);
    root.left = leftchild;
    root.right = rightchild;
    return root;
}
```

written by lurklurk original link here

Answer 3

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
        if(inorder.size() == 0)return NULL;
        TreeNode* p;
        TreeNode* root;
        vector<int> vint;
        vector<TreeNode*> vtn;
        root = new TreeNode(postorder.back());
        vtn.push_back(root);
        postorder.pop_back();
        while(true)
        {
            if(inorder.back() == vtn.back()->val)
            {
                p = vtn.back();
                vtn.pop_back();
                inorder.pop_back();
                if(inorder.size() == 0) break;
                if(vtn.size())
                    if(inorder.back() == vtn.back()->val)continue;
                p->left = new TreeNode(postorder.back());
                postorder.pop_back();
                vtn.push_back(p->left);
            }
            else
            {
                p = new TreeNode(postorder.back());
                postorder.pop_back();
                vtn.back()->right = p;
                vtn.push_back(p);
            }
        }
        return root;
    }
};
```

written by hongzhi original link here

Binary Tree Level Order Traversal II(107)

Answer 1

The way I see this problem is that it is EXACTLY the same as "Level-Order Traversal I" except that we need to reverse the final container for output, which is trivial. Is there a better idea that fits this problem specifically?

The attached is my current recursive solution. In each function call, we pass in the current node and its level. If this level does not yet exist in the output container, then we should add a new empty level. Then, we add the current node to the end of the current level, and recursively call the function passing the two children of the current node at the next level. This algorithm is really a DFS, but it saves the level information for each node and produces the same result as BFS would.

```
vector<vector<int> > res;

void DFS(TreeNode* root, int level)
{
    if (root == NULL) return;
    if (level == res.size()) // The level does not exist in output
    {
        res.push_back(vector<int>()); // Create a new level
    }

    res[level].push_back(root->val); // Add the current value to its level
    DFS(root->left, level+1); // Go to the next level
    DFS(root->right,level+1);
}

vector<vector<int> > levelOrderBottom(TreeNode *root) {
    DFS(root, 0);
    return vector<vector<int> > (res.rbegin(), res.rend());
}
```

written by stellari original link here

Answer 2

DFS solution:

```
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        List<List<Integer>> wrapList = new LinkedList<List<Integer>>();

        if(root == null) return wrapList;

        queue.offer(root);
        while(!queue.isEmpty()){
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for(int i=0; i<levelNum; i++) {
                if(queue.peek().left != null) queue.offer(queue.peek().left);
                if(queue.peek().right != null) queue.offer(queue.peek().right);
                subList.add(queue.poll().val);
            }
            wrapList.add(0, subList);
        }
        return wrapList;
    }
}
```

BFS solution:

```
public class Solution {
        public List<List<Integer>> levelOrderBottom(TreeNode root) {
            List<List<Integer>> wrapList = new LinkedList<List<Integer>>();
            levelMaker(wrapList, root, 0);
            return wrapList;
        }

        public void levelMaker(List<List<Integer>> list, TreeNode root, int level
) {
            if(root == null) return;
            if(level >= list.size()) {
                list.add(0, new LinkedList<Integer>());
            }
            levelMaker(list, root.left, level+1);
            levelMaker(list, root.right, level+1);
            list.get(list.size()-level-1).add(root.val);
        }
    }
```

written by SOY original link here

Answer 3

The addFirst() method of LinkedLinked save us from reverse final result.

```java
public List<List<Integer>> levelOrderBottom(TreeNode root) {
    LinkedList<List<Integer>> list = new LinkedList<List<Integer>>();
    addLevel(list, 0, root);
    return list;
}

private void addLevel(LinkedList<List<Integer>> list, int level, TreeNode node) {
    if (node == null) return;
    if (list.size()-1 < level) list.addFirst(new LinkedList<Integer>());
    list.get(list.size()-1-level).add(node.val);
    addLevel(list, level+1, node.left);
    addLevel(list, level+1, node.right);
}
```

written by pavel-shlyk original link here

Convert Sorted Array to Binary Search Tree(108)

Answer 1

Hi everyone, this is my accepted recursive Java solution. I get overflow problems at first because I didn't use mid - 1 and mid + 1 as the bound. Hope this helps :)

```
public TreeNode sortedArrayToBST(int[] num) {
    if (num.length == 0) {
        return null;
    }
    TreeNode head = helper(num, 0, num.length - 1);
    return head;
}

public TreeNode helper(int[] num, int low, int high) {
    if (low > high) { // Done
        return null;
    }
    int mid = (low + high) / 2;
    TreeNode node = new TreeNode(num[mid]);
    node.left = helper(num, low, mid - 1);
    node.right = helper(num, mid + 1, high);
    return node;
}
```

written by jiaming2 original link here

Answer 2

Recursively call the **sortedArrayToBST()** method providing new vector for each call to construct left and right children:

```
class Solution {
public:
    TreeNode *sortedArrayToBST(vector<int> &num) {
        if(num.size() == 0) return NULL;
        if(num.size() == 1)
        {
            return new TreeNode(num[0]);
        }

        int middle = num.size()/2;
        TreeNode* root = new TreeNode(num[middle]);

        vector<int> leftInts(num.begin(), num.begin()+middle);
        vector<int> rightInts(num.begin()+middle+1, num.end());

        root->left = sortedArrayToBST(leftInts);
        root->right = sortedArrayToBST(rightInts);

        return root;
    }
};
```

written by paul7 original link here

Answer 3

I came up with the recursion solution first and tried to translate it into an iterative solution. It is very similar to doing a tree inorder traversal, I use three stacks - nodeStack stores the node I am going to process next, and **leftIndexStack** and **rightIndexStack** store the range where this node need to read from the**nums**.

```
public class Solution {

    public TreeNode sortedArrayToBST(int[] nums) {

        int len = nums.length;
        if ( len == 0 ) { return null; }

        // 0 as a placeholder
        TreeNode head = new TreeNode(0);

        Deque<TreeNode> nodeStack       = new LinkedList<TreeNode>() {{ push(head
);  }};
        Deque<Integer>  leftIndexStack  = new LinkedList<Integer>()  {{ push(0);
}};
        Deque<Integer>  rightIndexStack = new LinkedList<Integer>()  {{ push(len-
1); }};

        while ( !nodeStack.isEmpty() ) {
            TreeNode currNode = nodeStack.pop();
            int left  = leftIndexStack.pop();
            int right = rightIndexStack.pop();
            int mid   = left + (right-left)/2; // avoid overflow
            currNode.val = nums[mid];
            if ( left <= mid-1 ) {
                currNode.left = new TreeNode(0);
                nodeStack.push(currNode.left);
                leftIndexStack.push(left);
                rightIndexStack.push(mid-1);
            }
            if ( mid+1 <= right ) {
                currNode.right = new TreeNode(0);
                nodeStack.push(currNode.right);
                leftIndexStack.push(mid+1);
                rightIndexStack.push(right);
            }
        }
        return head;
    }

}
```

written by benjamin19890721 original link here

Convert Sorted List to Binary Search Tree(109)

Answer 1

count is a function to calculate the size of list.

Key words: inorder traversal.

```
class Solution {
public:
    ListNode *list;
    int count(ListNode *node){
        int size = 0;
        while (node) {
            ++size;
            node = node->next;
        }
        return size;
    }

    TreeNode *generate(int n){
        if (n == 0)
            return NULL;
        TreeNode *node = new TreeNode(0);
        node->left = generate(n / 2);
        node->val = list->val;
        list = list->next;
        node->right = generate(n - n / 2 - 1);
        return node;
    }

    TreeNode *sortedListToBST(ListNode *head) {
        this->list = head;
        return generate(count(head));
    }
};
```

written by vaputa original link here

Answer 2

```java
private ListNode node;

public TreeNode sortedListToBST(ListNode head) {
    if(head == null){
        return null;
    }

    int size = 0;
    ListNode runner = head;
    node = head;

    while(runner != null){
        runner = runner.next;
        size ++;
    }

    return inorderHelper(0, size - 1);
}

public TreeNode inorderHelper(int start, int end){
    if(start > end){
        return null;
    }

    int mid = start + (end - start) / 2;
    TreeNode left = inorderHelper(start, mid - 1);

    TreeNode treenode = new TreeNode(node.val);
    treenode.left = left;
    node = node.next;

    TreeNode right = inorderHelper(mid + 1, end);
    treenode.right = right;

    return treenode;
}
```

written by treesbug original link here

Answer 3

```
class Solution {
public:
    TreeNode *sortedListToBST(ListNode *head)
    {
        return sortedListToBST( head, NULL );
    }

private:
    TreeNode *sortedListToBST(ListNode *head, ListNode *tail)
    {
        if( head == tail )
            return NULL;
        if( head->next == tail )    //
        {
            TreeNode *root = new TreeNode( head->val );
            return root;
        }
        ListNode *mid = head, *temp = head;
        while( temp != tail && temp->next != tail )    // å¯»æ¾ä¸é—´èŠ‚ç¹
        {
            mid = mid->next;
            temp = temp->next->next;
        }
        TreeNode *root = new TreeNode( mid->val );
        root->left = sortedListToBST( head, mid );
        root->right = sortedListToBST( mid->next, tail );
        return root;
    }
};
```

written by AllenYick original link here

Balanced Binary Tree(110)

Answer 1

This problem is generally believed to have two solutions: the top down approach and the bottom up way.

1.The first method checks whether the tree is balanced strictly according to the definition of balanced binary tree: the difference between the heights of the two sub trees are not bigger than 1, and both the left sub tree and right sub tree are also balanced. With the helper function depth(), we could easily write the code;

```
class solution {
public:
    int depth (TreeNode *root) {
        if (root == NULL) return 0;
        return max (depth(root -> left), depth (root -> right)) + 1;
    }

    bool isBalanced (TreeNode *root) {
        if (root == NULL) return true;

        int left=depth(root->left);
        int right=depth(root->right);

        return abs(left - right) <= 1 && isBalanced(root->left) && isBalanced(roo
t->right);
    }
};
```

For the current node root, calling depth() for its left and right children actually has to access all of its children, thus the complexity is O(N). We do this for each node in the tree, so the overall complexity of isBalanced will be O(N^2). This is a top down approach.

2.The second method is based on DFS. Instead of calling depth() explicitly for each child node, we return the height of the current node in DFS recursion. When the sub tree of the current node (inclusive) is balanced, the function dfsHeight() returns a non-negative value as the height. Otherwise -1 is returned. According to the leftHeight and rightHeight of the two children, the parent node could check if the sub tree is balanced, and decides its return value.

```
class solution {
public:
int dfsHeight (TreeNode *root) {
        if (root == NULL) return 0;

        int leftHeight = dfsHeight (root -> left);
        if (leftHeight == -1) return -1;
        int rightHeight = dfsHeight (root -> right);
        if (rightHeight == -1) return -1;

        if (abs(leftHeight - rightHeight) > 1)  return -1;
        return max (leftHeight, rightHeight) + 1;
    }
    bool isBalanced(TreeNode *root) {
        return dfsHeight (root) != -1;
    }
};
```

In this bottom up approach, each node in the tree only need to be accessed once. Thus the time complexity is O(N), better than the first solution.

written by benlong original link here

Answer 2

Input: {1,2,2,3,3,3,3,4,4,4,4,4,4,#,#,5,5}

Output: false (based on balanced binary definition **"no 2 leaf nodes differ in distance from the root by more than 1"**)

Expected: true (base on balanced binary definition **"two subtrees of every node never differ by more than 1"**)

written by Ethan original link here

Answer 3

```java
public boolean isBalanced(TreeNode root) {
    if(root==null){
        return true;
    }
    return height(root)!=-1;

}
public int height(TreeNode node){
    if(node==null){
        return 0;
    }
    int lH=height(node.left);
    if(lH==-1){
        return -1;
    }
    int rH=height(node.right);
    if(rH==-1){
        return -1;
    }
    if(lH-rH<-1 || lH-rH>1){
        return -1;
    }
    return Math.max(lH,rH)+1;
}
```

written by mingyuan original link here

Minimum Depth of Binary Tree(111)

Answer 1

```
public class Solution {
    public int minDepth(TreeNode root) {
        if(root == null) return 0;
        int left = minDepth(root.left);
        int right = minDepth(root.right);
        return (left == 0 || right == 0) ? left + right + 1: Math.min(left,right)
+ 1;

    }
}
```

written by caiqi8877 original link here

Answer 2

```
class Solution {
public:
    int minDepth(TreeNode *root) {
        if(!root) return 0;
        if(!root->left) return 1 + minDepth(root->right);
        if(!root->right) return 1 + minDepth(root->left);
        return 1+min(minDepth(root->left),minDepth(root->right));
    }
};
```

written by wulinjiansheng original link here

Answer 3

Why expected result for input of {1,2} is 2? Shouldn't it be 1?

written by ruilingrandy original link here

Path Sum(112)

Answer 1

The basic idea is to subtract the value of current node from sum until it reaches a leaf node and the subtraction equals 0, then we know that we got a hit. Otherwise the subtraction at the end could not be 0.

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null) return false;

        if(root.left == null && root.right == null && sum - root.val == 0) return
true;

        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, su
m - root.val);
    }
}
```

written by boy27910230 original link here

Answer 2

```
bool hasPathSum(TreeNode *root, int sum) {
        if (root == NULL) return false;
        if (root->val == sum && root->left ==  NULL && root->right == NULL) retur
n true;
        return hasPathSum(root->left, sum-root->val) || hasPathSum(root->right, s
um-root->val);
    }
```

written by pankit original link here

Answer 3

In the postorder traversal, the node will be removed from the stack only when the right sub-tree has been visited.so the path will be stored in the stack. we can keep check the SUM, the length from root to leaf node. at leaf node, if SUM == sum, OK, return true. After postorder traversal, return false.

I have compared this solution with recursion solutions. In the leetcode OJ, the run time of two solutions is very near.

below is my iterator code.

```
class Solution {
public:
    bool hasPathSum(TreeNode *root, int sum) {
        stack<TreeNode *> s;
        TreeNode *pre = NULL, *cur = root;
        int SUM = 0;
        while (cur || !s.empty()) {
            while (cur) {
                s.push(cur);
                SUM += cur->val;
                cur = cur->left;
            }
            cur = s.top();
            if (cur->left == NULL && cur->right == NULL && SUM == sum) {
                return true;
            }
            if (cur->right && pre != cur->right) {
                cur = cur->right;
            } else {
                pre = cur;
                s.pop();
                SUM -= cur->val;
                cur = NULL;
            }
        }
        return false;
    }
};
```

written by SJames original link here

Path Sum II(113)

Answer 1

```
public List<List<Integer>> pathSum(TreeNode root, int sum){
    List<List<Integer>> result  = new LinkedList<List<Integer>>();
    List<Integer> currentResult  = new LinkedList<Integer>();
    pathSum(root,sum,currentResult,result);
    return result;
}

public void pathSum(TreeNode root, int sum, List<Integer> currentResult,
        List<List<Integer>> result) {

    if (root == null)
        return;
    currentResult.add(new Integer(root.val));
    if (root.left == null && root.right == null && sum == root.val) {
        result.add(new LinkedList(currentResult));
        currentResult.remove(currentResult.size() - 1);//don't forget to remove t
he last integer
        return;
    } else {
        pathSum(root.left, sum - root.val, currentResult, result);
        pathSum(root.right, sum - root.val, currentResult, result);
    }
    currentResult.remove(currentResult.size() - 1);
}
```

written by wdjoxda original link here

Answer 2

Well, a typical backtracking problem. The code is as follows. You may walk through it using the example in the problem statement to see how it works.

```
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        vector<vector<int> > paths;
        vector<int> path;
        findPaths(root, sum, path, paths);
        return paths;
    }
private:
    void findPaths(TreeNode* node, int sum, vector<int>& path, vector<vector<int>
>& paths) {
        if (!node) return;
        path.push_back(node -> val);
        if (!(node -> left) && !(node -> right) && sum == node -> val)
            paths.push_back(path);
        findPaths(node -> left, sum - node -> val, path, paths);
        findPaths(node -> right, sum - node -> val, path, paths);
        path.pop_back();
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```
vector<vector<int> > pathSum(TreeNode *root, int sum) {
        vector<vector<int> > result;
        vector<int> cur_path(0);
        pathSumRec(root, sum, result, cur_path);
        return result;
    }

    // pass the current path as a reference and remember to pop out the last added
element
    // this improves the performance by 5 times
    void pathSumRec(TreeNode* root, int sum, vector<vector<int> >& result, vector
<int>& cur_path) {
        if (root == NULL) {
            return;
        }

        if (root->val == sum && root->left == NULL && root->right == NULL) {
            cur_path.push_back(root->val);
            result.push_back(cur_path);
            cur_path.pop_back();
            return;
        }

        int sum_left = sum - root->val;
        cur_path.push_back(root->val);
        pathSumRec(root->left, sum_left, result, cur_path);
        //cur_path.pop_back();
        pathSumRec(root->right, sum_left, result, cur_path);
        cur_path.pop_back();
    }
```

written by pankit original link here

Flatten Binary Tree to Linked List(114)

Answer 1

```
private TreeNode prev = null;

public void flatten(TreeNode root) {
    if (root == null)
        return;
    flatten(root.right);
    flatten(root.left);
    root.right = prev;
    root.left = null;
    prev = root;
}
```

written by tusizi original link here

Answer 2

```
class Solution {
public:
    void flatten(TreeNode *root) {
        TreeNode*now = root;
        while (now)
        {
            if(now->left)
            {
                //Find current node's prenode that links to current node's right
subtree
                TreeNode* pre = now->left;
                while(pre->right)
                {
                    pre = pre->right;
                }
                pre->right = now->right;
                //Use current node's left subtree to replace its right subtree(or
iginal right
                //subtree is already linked by current node's prenode
                now->right = now->left;
                now->left = NULL;
            }
            now = now->right;
        }
    }
};
```

written by zjulyx original link here

Answer 3

```
void flatten(TreeNode *root) {
    while (root) {
        if (root->left && root->right) {
            TreeNode* t = root->left;
            while (t->right)
                t = t->right;
            t->right = root->right;
        }

        if(root->left)
            root->right = root->left;
        root->left = NULL;
        root = root->right;
    }
}
```

written by jaewoo original link here

Distinct Subsequences(115)

Answer 1

My solution is using O(n^2) space and running in O(n^2) time. I wonder is there a better way to do that which consumes less memory? I guess run time could not be improved though. Any thought/input would be highly appreciated, thanks!

```
/**
 * Solution (DP):
 * We keep a m*n matrix and scanning through string S, while
 * m = T.length() + 1 and n = S.length() + 1
 * and each cell in matrix Path[i][j] means the number of distinct subsequences o
f
 * T.substr(1...i) in S(1...j)
 *
 * Path[i][j] = Path[i][j-1]            (discard S[j])
 *              +     Path[i-1][j-1]    (S[j] == T[i] and we are going to use S[j
])
 *                 or 0                 (S[j] != T[i] so we could not use S[j])
 * while Path[0][j] = 1 and Path[i][0] = 0.
 */
int numDistinct(string S, string T) {
    int m = T.length();
    int n = S.length();
    if (m > n) return 0;    // impossible for subsequence
    vector<vector<int>> path(m+1, vector<int>(n+1, 0));
    for (int k = 0; k <= n; k++) path[0][k] = 1;    // initialization

    for (int j = 1; j <= n; j++) {
        for (int i = 1; i <= m; i++) {
            path[i][j] = path[i][j-1] + (T[i-1] == S[j-1] ? path[i-1][j-1] : 0);
        }
    }

    return path[m][n];
}
```

written by dragonmigo original link here

Answer 2

The idea is the following:

- we will build an array `mem` where `mem[i+1][j+1]` means that `S[0..j]` contains `T[0..i]` that many times as distinct subsequences. Therefor the result will be `mem[T.length()][S.length()]`.
- we can build this array rows-by-rows:
  - the first row must be filled with 1. That's because the empty string is a subsequence of any string but only 1 time. So `mem[0][j] = 1` for every `j`. So with this we not only make our lives easier, but we also return correct value if `T` is an empty string.
  - the first column of every rows except the first must be 0. This is because

an empty string cannot contain a non-empty string as a substring -- the very first item of the array: `mem[0][0] = 1`, because an empty string contains the empty string 1 time.

So the matrix looks like this:

```
  S 0123....j
T +----------+
  |1111111111|
0 |0         |
1 |0         |
2 |0         |
. |0         |
. |0         |
i |0         |
```

From here we can easily fill the whole grid: for each `(x, y)`, we check if `S[x] == T[y]` we add the previous item and the previous item in the previous row, otherwise we copy the previous item in the same row. The reason is simple:

- if the current character in S doesn't equal to current character T, then we have the same number of distinct subsequences as we had without the new character.
- if the current character in S equal to the current character T, then the distinct number of subsequences: the number we had before **plus** the distinct number of subsequences we had with less longer T and less longer S.

An example: `S: [acdabefbc]` and `T: [ab]`

first we check with `a`:

```
           *  *
      S = [acdabefbc]
mem[1] = [0111222222]
```

then we check with `ab`:

```
               *  * ]
      S = [acdabefbc]
mem[1] = [0111222222]
mem[2] = [0000022244]
```

And the result is 4, as the distinct subsequences are:

```
      S = [a   b    ]
      S = [a      b ]
      S = [   ab    ]
      S = [   a   b ]
```

See the code in Java:

```
public int numDistinct(String S, String T) {
    // array creation
    int[][] mem = new int[T.length()+1][S.length()+1];

    // filling the first row: with 1s
    for(int j=0; j<=S.length(); j++) {
        mem[0][j] = 1;
    }

    // the first column is 0 by default in every other rows but the first, which
we need.

    for(int i=0; i<T.length(); i++) {
        for(int j=0; j<S.length(); j++) {
            if(T.charAt(i) == S.charAt(j)) {
                mem[i+1][j+1] = mem[i][j] + mem[i+1][j];
            } else {
                mem[i+1][j+1] = mem[i+1][j];
            }
        }
    }

    return mem[T.length()][S.length()];
}
```

written by balint original link here

Answer 3

Could someone please clarify this problem to me?

> Given a string S and a string T, count the number of distinct subsequences of T in S.
>
> A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).
>
> Here is an example: S = "rabbbit", T = "rabbit" count = 3

If I understood correctly, we need to find all distinct subsequences of T and see how many, if any appear in s. How does that equal to 3 in the given example?

written by princessmaja original link here

Populating Next Right Pointers in Each Node(116)

Answer 1

```
void connect(TreeLinkNode *root) {
    if (root == NULL) return;
    TreeLinkNode *pre = root;
    TreeLinkNode *cur = NULL;
    while(pre->left) {
        cur = pre;
        while(cur) {
            cur->left->next = cur->right;
            if(cur->next) cur->right->next = cur->next->left;
            cur = cur->next;
        }
        pre = pre->left;
    }
}
```

you need two additional pointer.

written by ragepyre original link here

Answer 2

```
void connect(TreeLinkNode *root) {
    if(!root)
        return;
    while(root -> left)
    {
        TreeLinkNode *p = root;
        while(p)
        {
            p -> left -> next = p -> right;
            if(p -> next)
                p -> right -> next = p -> next -> left;
            p = p -> next;
        }
        root = root -> left;
    }
}
```

written by Erudy original link here

Answer 3

```
public class Solution {
    public void connect(TreeLinkNode root) {
        TreeLinkNode level_start=root;
        while(level_start!=null){
            TreeLinkNode cur=level_start;
            while(cur!=null){
                if(cur.left!=null) cur.left.next=cur.right;
                if(cur.right!=null && cur.next!=null) cur.right.next=cur.next.lef
t;

                cur=cur.next;
            }
            level_start=level_start.left;
        }
    }
}
```

written by talent58 original link here

Populating Next Right Pointers in Each Node II(117)

Answer 1

Just share my iterative solution with O(1) space and O(n) Time complexity

```
public class Solution {

    //based on level order traversal
    public void connect(TreeLinkNode root) {

        TreeLinkNode head = null; //head of the next level
        TreeLinkNode prev = null; //the leading node on the next level
        TreeLinkNode cur = root;  //current node of current level

        while (cur != null) {

            while (cur != null) { //iterate on the current level
                //left child
                if (cur.left != null) {
                    if (prev != null) {
                        prev.next = cur.left;
                    } else {
                        head = cur.left;
                    }
                    prev = cur.left;
                }
                //right child
                if (cur.right != null) {
                    if (prev != null) {
                        prev.next = cur.right;
                    } else {
                        head = cur.right;
                    }
                    prev = cur.right;
                }
                //move to next node
                cur = cur.next;
            }

            //move to next level
            cur = head;
            head = null;
            prev = null;
        }

    }
}
```

written by flashstone original link here

Answer 2

The idea is simple: level-order traversal. You can see the following code:

```
public class Solution {
    public void connect(TreeLinkNode root) {

        while(root != null){
            TreeLinkNode tempChild = new TreeLinkNode(0);
            TreeLinkNode currentChild = tempChild;
            while(root!=null){
                if(root.left != null) { currentChild.next = root.left; currentChi
ld = currentChild.next;}
                if(root.right != null) { currentChild.next = root.right; currentC
hild = currentChild.next;}
                root = root.next;
            }
            root = tempChild.next;
        }
    }
}
```

written by davidtan1890 original link here

Answer 3

Thanks for liji94188 for adding the explanation:

It's a BFS traversal. now pointer is the current level traveler and head is the left most element at next level and the tail is the right most element at next level till now. We move now pointer at current level and populate the the next-link at its children level. (Here the gist is we can move now to its next because this relationship was already populated in the previous round).

```
void connect(TreeLinkNode *root) {
    TreeLinkNode *now, *tail, *head;

    now = root;
    head = tail = NULL;
    while(now)
    {
        if (now->left)
            if (tail) tail = tail->next =now->left;
            else head = tail = now->left;
        if (now->right)
            if (tail) tail = tail->next =now->right;
            else head = tail = now->right;
        if(!(now = now->next))
        {
            now = head;
            head = tail=NULL;
        }
    }
}
```

written by aileengw original link here

Pascal's Triangle(118)

Answer 1

```
public class Solution {
public List<List<Integer>> generate(int numRows)
{
    List<List<Integer>> allrows = new ArrayList<List<Integer>>();
    ArrayList<Integer> row = new ArrayList<Integer>();
    for(int i=0;i<numRows;i++)
    {
        row.add(0, 1);
        for(int j=1;j<row.size()-1;j++)
            row.set(j, row.get(j)+row.get(j+1));
        allrows.add(new ArrayList<Integer>(row));
    }
    return allrows;

}
```

}

written by rheaxu original link here

Answer 2

two loops, one go through the row, one go through the column

database: pretty straight forward, ArrayList

calculate element value: K(i)(j)=K(i-1)(j-1)+K(i-1)(j) except for the first and last element

```
public class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> triangle = new ArrayList<List<Integer>>();
        if (numRows <=0){
            return triangle;
        }
        for (int i=0; i<numRows; i++){
            List<Integer> row =  new ArrayList<Integer>();
            for (int j=0; j<i+1; j++){
                if (j==0 || j==i){
                    row.add(1);
                } else {
                    row.add(triangle.get(i-1).get(j-1)+triangle.get(i-1).get(j));
                }
            }
            triangle.add(row);
        }
        return triangle;
    }
}
```

written by chennan original link here

Answer 3

```
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int>> r(numRows);

        for (int i = 0; i < numRows; i++) {
            r[i].resize(i + 1);
            r[i][0] = r[i][i] = 1;

            for (int j = 1; j < i; j++)
                r[i][j] = r[i - 1][j - 1] + r[i - 1][j];
        }

        return r;
    }
};
```

written by mzchen original link here

Pascal's Triangle II(119)

Answer 1

The basic idea is to iteratively update the array from the end to the beginning.

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> A(rowIndex+1, 0);
        A[0] = 1;
        for(int i=1; i<rowIndex+1; i++)
            for(int j=i; j>=1; j--)
                A[j] += A[j-1];
        return A;
    }
};
```

written by LongsPeak original link here

Answer 2

```
  public List<Integer> getRow(int rowIndex) {
    List<Integer> list = new ArrayList<Integer>();
    if (rowIndex < 0)
        return list;

    for (int i = 0; i < rowIndex + 1; i++) {
        list.add(0, 1);
        for (int j = 1; j < list.size() - 1; j++) {
            list.set(j, list.get(j) + list.get(j + 1));
        }
    }
    return list;
}
```

written by micheal.zhou original link here

Answer 3

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> vi(rowIndex + 1);
        vi[0] = 1;
        for (int i = 0; i <= rowIndex ; ++i)
        {
            for (int j = i; j > 0; --j)
            {
                vi[j] = vi[j] + vi[j-1];
            }
        }
        return vi;
    }
};
```

written by DaZhang original link here

Triangle(120)

Answer 1

This problem is quite well-formed in my opinion. The triangle has a tree-like structure, which would lead people to think about traversal algorithms such as DFS. However, if you look closely, you would notice that the adjacent nodes always share a 'branch'. In other word, there are **overlapping subproblems**. Also, suppose x and y are 'children' of k. Once minimum paths from x and y to the bottom are known, the minimum path starting from k can be decided in O(1), that is **optimal substructure**. Therefore, dynamic programming would be the best solution to this problem in terms of time complexity.

What I like about this problem even more is that the difference between 'top-down' and 'bottom-up' DP can be 'literally' pictured in the input triangle. For 'top-down' DP, starting from the node on the very top, we recursively find the minimum path sum of each node. When a path sum is calculated, we store it in an array (memoization); the next time we need to calculate the path sum of the same node, just retrieve it from the array. However, you will need a cache that is at least the same size as the input triangle itself to store the pathsum, which takes O(N^2) space. With some clever thinking, it might be possible to release some of the memory that will never be used after a particular point, but the order of the nodes being processed is not straightforwardly seen in a recursive solution, so deciding which part of the cache to discard can be a hard job.

'Bottom-up' DP, on the other hand, is very straightforward: we start from the nodes on the bottom row; the min pathsums for these nodes are the values of the nodes themselves. From there, the min pathsum at the ith node on the kth row would be the lesser of the pathsums of its two children plus the value of itself, i.e.:

```
minpath[k][i] = min( minpath[k+1][i], minpath[k+1][i+1]) + triangle[k][i];
```

Or even better, since the row minpath[k+1] would be useless after minpath[k] is computed, we can simply set minpath as a 1D array, and iteratively update itself:

```
For the kth level:
minpath[i] = min( minpath[i], minpath[i+1]) + triangle[k][i];
```

Thus, we have the following solution

```
int minimumTotal(vector<vector<int> > &triangle) {
    int n = triangle.size();
    vector<int> minlen(triangle.back());
    for (int layer = n-2; layer >= 0; layer--) // For each layer
    {
        for (int i = 0; i <= layer; i++) // Check its every 'node'
        {
            // Find the lesser of its two children, and sum the current value in
the triangle with it.
            minlen[i] = min(minlen[i], minlen[i+1]) + triangle[layer][i];
        }
    }
    return minlen[0];
}
```

written by stellari original link here

Answer 2

```
public class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        for(int i = triangle.size() - 2; i >= 0; i--)
            for(int j = 0; j <= i; j++)
                triangle.get(i).set(j, triangle.get(i).get(j) + Math.min(triangle.
get(i + 1).get(j), triangle.get(i + 1).get(j + 1)));
        return triangle.get(0).get(0);
    }
}
```

The idea is simple.

1) Go from bottom to top.

2) We start form the row above the bottom row [size()-2].

3) Each number add the smaller number of two numbers that below it.

4) And finally we get to the top we the smallest sum.

written by SteveLee1989 original link here

Answer 3

```
class Solution {
public:
    int minimumTotal(vector<vector<int> > &triangle)
    {
        vector<int> mini = triangle[triangle.size()-1];
        for ( int i = triangle.size() - 2; i>= 0 ; --i )
            for ( int j = 0; j < triangle[i].size() ; ++ j )
                mini[j] = triangle[i][j] + min(mini[j],mini[j+1]);
        return mini[0];
    }
};
```

written by rishav2 original link here

Best Time to Buy and Sell Stock(121)

Answer 1

```
int maxProfit(vector<int> &prices) {
    int maxPro = 0;
    int minPrice = INT_MAX;
    for(int i = 0; i < prices.size(); i++){
        minPrice = min(minPrice, prices[i]);
        maxPro = max(maxPro, prices[i] - minPrice);
    }
    return maxPro;
}
```

minPrice is the minimum price from day 0 to day i. And maxPro is the maximum profit we can get from day 0 to day i.

How to get maxPro? Just get the larger one between current maxPro and prices[i] - minPrice.

written by zxyperfect original link here

Answer 2

The logic to solve this problem is same as "max subarray problem" using `Kadane's Algorithm`. Since no body has mentioned this so far, I thought it's a good thing for everybody to know.

All the straight forward solution should work, but if the interviewer twists the question slightly by giving the ***difference array of prices***, Ex: for `{1, 7, 4, 11}`, if he gives `{0, 6, -3, 7}`, you might end up being confused.

Here, the logic is to calculate the difference (`maxCur += prices[i] - prices[i-1]`) of the original array, and find a contiguous subarray giving maximum profit. If the difference falls below 0, reset it to zero.

```
    public int maxProfit(int[] prices) {
        int maxCur = 0, maxSoFar = 0;
        for(int i = 1; i < prices.length; i++) {
            maxCur = Math.max(0, maxCur += prices[i] - prices[i-1]);
            maxSoFar = Math.max(maxCur, maxSoFar);
        }
        return maxSoFar;
    }
```

* `maxCur = current maximum value`
* `maxSoFar = maximum value found so far`

written by andywhite original link here

Answer 3

1.for prices[0] .... prices[n], prices[n+1]..... if (prices[n] < prices[0]) then, the max profit is in prices[0]...prices[n], or begin from prices[n+1], otherwise, suppose prices[n+1] > prices[0], and max profit is happened between prices[n+1] , and prices1">n+k, then if we buy at day 0, and sell at day n+k, we get a bigger profit.

Base on logic above, we can have a O(1*n) solution:

```
public class Solution {
    public int maxProfit(int[] prices) {

        if (prices.length == 0)
        {
            return 0;
        }

        int max = 0, min = prices[0];
        int profit = 0;

        for (int i = 1; i < prices.length; i++)
        {
            if (prices[i] < min)
            {

                min = prices[i];
            }
            else
            {
                if (prices[i] - min > profit)
                {
                    profit = prices[i] - min;
                }

            }
        }



     return profit;

    }
}
```

written by wycl16514 original link here

Best Time to Buy and Sell Stock II(122)

Answer 1

```
public class Solution {
public int maxProfit(int[] prices) {
    int total = 0;
    for (int i=0; i< prices.length-1; i++) {
        if (prices[i+1]>prices[i]) total += prices[i+1]-prices[i];
    }

    return total;
}
```

A simple code like this. The designer of this question must thought of something too complicated.

written by jyan original link here

Answer 2

First we post the code here.

```
int maxProfit(vector<int> &prices) {
    int ret = 0;
    for (size_t p = 1; p < prices.size(); ++p)
      ret += max(prices[p] - prices[p - 1], 0);
    return ret;
}
```

Second, suppose the first sequence is "a <= b <= c <= d", the profit is "d - a = (b - a) + (c - b) + (d - c)" without a doubt. And suppose another one is "a <= b >= b' <= c <= d", the profit is not difficult to be figured out as "(b - a) + (d - b')". So you just target at monotone sequences.

written by tian.xia.568 original link here

Answer 3

Hi guys!

The greedy pair-wise approach mentioned in other posts is great for this problem indeed, but if we're not allowed to buy and sell stocks within the same day it can't be applied (logically, of course; the answer will be the same). Actually, the straight-forward way of finding next local minimum and next local maximum is not much more complicated, so, just for the sake of having an alternative I share the code in Java for such case.

```
public int maxProfit(int[] prices) {
    int profit = 0, i = 0;
    while (i < prices.length) {
        // find next local minimum
        while (i < prices.length-1 && prices[i+1] <= prices[i]) i++;
        int min = prices[i++]; // need increment to avoid infinite loop for "[1]"
        // find next local maximum
        while (i < prices.length-1 && prices[i+1] >= prices[i]) i++;
        profit += i < prices.length ? prices[i++] - min : 0;
    }
    return profit;
}
```

Happy coding!

written by shpolsky original link here

Best Time to Buy and Sell Stock III(123)

Answer 1

The thinking is simple and is inspired by the best solution from Single Number II (I read through the discussion after I use DP). Assume we only have 0 money at first; 4 Variables to maintain some interested 'ceilings' so far: The maximum of if we've just buy 1st stock, if we've just sold 1nd stock, if we've just buy 2nd stock, if we've just sold 2nd stock. Very simple code too and work well. I have to say the logic is simple than those in Single Number II.

```
public class Solution {
    public int maxProfit(int[] prices) {
        int hold1 = Integer.MIN_VALUE, hold2 = Integer.MIN_VALUE;
        int release1 = 0, release2 = 0;
        for(int i:prices){                              // Assume we only have 0
money at first
            release2 = Math.max(release2, hold2+i);     // The maximum if we've j
ust sold 2nd stock so far.
            hold2    = Math.max(hold2,    release1-i);  // The maximum if we've j
ust buy  2nd stock so far.
            release1 = Math.max(release1, hold1+i);     // The maximum if we've j
ust sold 1nd stock so far.
            hold1    = Math.max(hold1,    -i);          // The maximum if we've j
ust buy  1st stock so far.
        }
        return release2; ///Since release1 is initiated as 0, so release2 will alw
ays higher than release1.
    }
}
```

written by weijiac original link here

Answer 2

Solution is commented in the code. Time complexity is O(k*n*), *space complexity can be O(n) because this DP only uses the result from last step. But for cleaness this solution still used O(k*n) space complexity to preserve similarity to the equations in the comments.

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        // f[k, ii] represents the max profit up until prices[ii] (Note: NOT endi
ng with prices[ii]) using at most k transactions.
        // f[k, ii] = max(f[k, ii-1], prices[ii] - prices[jj] + f[k-1, jj]) { jj
in range of [0, ii-1] }
        //          = max(f[k, ii-1], prices[ii] + max(f[k-1, jj] - prices[jj]))
        // f[0, ii] = 0; 0 times transation makes 0 profit
        // f[k, 0] = 0; if there is only one price data point you can't make any
money no matter how many times you can trade
        if (prices.size() <= 1) return 0;
        else {
            int K = 2; // number of max transation allowed
            int maxProf = 0;
            vector<vector<int>> f(K+1, vector<int>(prices.size(), 0));
            for (int kk = 1; kk <= K; kk++) {
                int tmpMax = f[kk-1][0] - prices[0];
                for (int ii = 1; ii < prices.size(); ii++) {
                    f[kk][ii] = max(f[kk][ii-1], prices[ii] + tmpMax);
                    tmpMax = max(tmpMax, f[kk-1][ii] - prices[ii]);
                    maxProf = max(f[kk][ii], maxProf);
                }
            }
            return maxProf;
        }
    }
};
```

written by peterleetcode original link here

Answer 3

It is similar to other buy/sell problems. just do DP and define an array of states to track the current maximum profits at different stages. For example, in the below code

- states[][0]: one buy
- states[][1]: one buy, one sell
- states[][2]: two buys, one sell
- states[][3]: two buy, two sells

The states transistions occurs when buy/sell operations are executed. For example, state[][0] can move to state[][1] via one sell operation.

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int states[2][4] = {INT_MIN, 0, INT_MIN, 0}; // 0: 1 buy, 1: one buy/sell
, 2: 2 buys/1 sell, 3, 2 buys/sells
        int len = prices.size(), i, cur = 0, next =1;
        for(i=0; i<len; ++i)
        {
            states[next][0] = max(states[cur][0], -prices[i]);
            states[next][1] = max(states[cur][1], states[cur][0]+prices[i]);
            states[next][2] = max(states[cur][2], states[cur][1]-prices[i]);
            states[next][3] = max(states[cur][3], states[cur][2]+prices[i]);
            swap(next, cur);
        }
        return max(states[cur][1], states[cur][3]);
    }
};
```

written by dong.wang.1694 original link here

Binary Tree Maximum Path Sum(124)

Answer 1

Here's my ideas:

- A path from start to end, goes up on the tree for 0 or more steps, then goes down for 0 or more steps. Once it goes down, it can't go up. Each path has a highest node, which is also the lowest common ancestor of all other nodes on the path.
- A recursive method `maxPathDown(TreeNode node)` (1) computes the maximum path sum with highest node is the input node, update maximum if necessary (2) returns the maximum sum of the path that can be extended to input node's parent.

Code:

```
public class Solution {
    int maxValue;

    public int maxPathSum(TreeNode root) {
        maxValue = Integer.MIN_VALUE;
        maxPathDown(root);
        return maxValue;
    }

    private int maxPathDown(TreeNode node) {
        if (node == null) return 0;
        int left = Math.max(0, maxPathDown(node.left));
        int right = Math.max(0, maxPathDown(node.right));
        maxValue = Math.max(maxValue, left + right + node.val);
        return Math.max(left, right) + node.val;
    }
}
```

written by wei-bung original link here

Answer 2

```
class Solution {
    int maxToRoot(TreeNode *root, int &re) {
        if (!root) return 0;
        int l = maxToRoot(root->left, re);
        int r = maxToRoot(root->right, re);
        if (l < 0) l = 0;
        if (r < 0) r = 0;
        if (l + r + root->val > re) re = l + r + root->val;
        return root->val += max(l, r);
    }
public:
    int maxPathSum(TreeNode *root) {
        int max = -2147483648;
        maxToRoot(root, max);
        return max;
    }
};
```

update the val of each node of the tree bottom-up, the new val of TreeNode *x stands for the max sum started from any node in subtree x and ended in x, mataining the re for result in traversal at the same time.

written by xt2357 original link here

Answer 3

```
public class Solution {
    int max = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        helper(root);
        return max;
    }

    // helper returns the max branch
    // plus current node's value
    int helper(TreeNode root) {
        if (root == null) return 0;

        int left = Math.max(helper(root.left), 0);
        int right = Math.max(helper(root.right), 0);

        max = Math.max(max, root.val + left + right);

        return root.val + Math.max(left, right);
    }
}
```

written by jeantimex original link here

Valid Palindrome(125)

Answer 1

```
bool isPalindrome(string s) {
    for (int i = 0, j = s.size() - 1; i < j; i++, j--) { // Move 2 pointers from
each end until they collide
        while (isalnum(s[i]) == false && i < j) i++; // Increment left pointer if
not alphanumeric
        while (isalnum(s[j]) == false && i < j) j--; // Decrement right pointer i
f no alphanumeric
        if (toupper(s[i]) != toupper(s[j])) return false; // Exit and return erro
r if not match
    }

    return true;
}
```

written by satyakam original link here

Answer 2

```
bool isPalindrome(string s) {
    int start=0, end=s.length()-1;
    while(start<end) {
        if (!isalnum(s[start])) start++;
        else if (!isalnum(s[end])) end--;
        else {
            if (tolower(s[start++])!=tolower(s[end--])) return false;
        }
    }
    return true;
}
```

written by lester_zhang original link here

Answer 3

```java
public class Solution {
    public boolean isPalindrome(String s) {
        if (s.isEmpty()) {
            return true;
        }
        int head = 0, tail = s.length() - 1;
        char cHead, cTail;
        while(head <= tail) {
            cHead = s.charAt(head);
            cTail = s.charAt(tail);
            if (!Character.isLetterOrDigit(cHead)) {
                head++;
            } else if(!Character.isLetterOrDigit(cTail)) {
                tail--;
            } else {
                if (Character.toLowerCase(cHead) != Character.toLowerCase(cTail))
{
                    return false;
                }
                head++;
                tail--;
            }
        }

        return true;
    }
}
```

written by aleksey.danilevsky.5 original link here

Word Ladder II(126)

Answer 1

The solution contains two steps 1 Use BFS to construct a graph. 2. Use DFS to construct the paths from end to start.Both solutions got AC within 1s.

The first step BFS is quite important. I summarized three tricks

1) Using a **MAP** to store the min ladder of each word, or use a **SET** to store the words visited in current ladder, when the current ladder was completed, delete the visited words from unvisited. That's why I have two similar solutions.

2) Use **Character iteration** to find all possible paths. Do not compare one word to all the other words and check if they only differ by one character.

3) One word is allowed to be inserted into the queue only **ONCE**. See my comments.

```
public class Solution {
    Map<String,List<String>> map;
    List<List<String>> results;
    public List<List<String>> findLadders(String start, String end, Set<String> d
ict) {
        results= new ArrayList<List<String>>();
        if (dict.size() == 0)
            return results;

        int min=Integer.MAX_VALUE;

        Queue<String> queue= new ArrayDeque<String>();
        queue.add(start);

        map = new HashMap<String,List<String>>();

        Map<String,Integer> ladder = new HashMap<String,Integer>();
        for (String string:dict)
            ladder.put(string, Integer.MAX_VALUE);
        ladder.put(start, 0);

        dict.add(end);
        //BFS: Dijisktra search
        while (!queue.isEmpty()) {

            String word = queue.poll();

            int step = ladder.get(word)+1;//'step' indicates how many steps are n
eeded to travel to one word.

            if (step>min) break;

            for (int i = 0; i < word.length(); i++){
               StringBuilder builder = new StringBuilder(word);
                for (char ch='a';  ch <= 'z'; ch++){
                    builder.setCharAt(i,ch);
                    String new_word=builder.toString();
                    if (ladder.containsKey(new_word)) {
```

```
                        if (step>ladder.get(new_word))//Check if it is the shorte
st path to one word.
                            continue;
                        else if (step<ladder.get(new_word)){
                            queue.add(new_word);
                            ladder.put(new_word, step);
                        }else;// It is a KEY line. If one word already appeared i
n one ladder,
                              // Do not insert the same word inside the queue twi
ce. Otherwise it gets TLE.

                        if (map.containsKey(new_word)) //Build adjacent Graph
                            map.get(new_word).add(word);
                        else{
                            List<String> list= new LinkedList<String>();
                            list.add(word);
                            map.put(new_word,list);
                            //It is possible to write three lines in one:
                            //map.put(new_word,new LinkedList<String>(Arrays.asLi
st(new String[]{word})));
                            //Which one is better?
                        }

                        if (new_word.equals(end))
                            min=step;

                    }//End if dict contains new_word
                }//End:Iteration from 'a' to 'z'
            }//End:Iteration from the first to the last
        }//End While

        //BackTracking
        LinkedList<String> result = new LinkedList<String>();
        backTrace(end,start,result);

        return results;
    }
    private void backTrace(String word,String start,List<String> list){
        if (word.equals(start)){
            list.add(0,start);
            results.add(new ArrayList<String>(list));
            list.remove(0);
            return;
        }
        list.add(0,word);
        if (map.get(word)!=null)
            for (String s:map.get(word))
                backTrace(s,start,list);
        list.remove(0);
    }
}
```

Another solution using two sets. This is similar to the answer in the most viewed thread. While I found my solution more readable and efficient.

```
public class Solution {
    List<List<String>> results;
    List<String> list;
    Map<String,List<String>> map;
        public List<List<String>> findLadders(String start, String end, Set<Strin
g> dict) {
            results= new ArrayList<List<String>>();
            if (dict.size() == 0)
                return results;

            int curr=1,next=0;
            boolean found=false;
            list = new LinkedList<String>();
            map = new HashMap<String,List<String>>();

            Queue<String> queue= new ArrayDeque<String>();
            Set<String> unvisited = new HashSet<String>(dict);
            Set<String> visited = new HashSet<String>();

            queue.add(start);
            unvisited.add(end);
            unvisited.remove(start);
            //BFS
            while (!queue.isEmpty()) {

                String word = queue.poll();
                curr--;
                for (int i = 0; i < word.length(); i++){
                   StringBuilder builder = new StringBuilder(word);
                    for (char ch='a';  ch <= 'z'; ch++){
                        builder.setCharAt(i,ch);
                        String new_word=builder.toString();
                        if (unvisited.contains(new_word)){
                            //Handle queue
                            if (visited.add(new_word)){//Key statement,Avoid Dupl
icate queue insertion
                                next++;
                                queue.add(new_word);
                            }

                            if (map.containsKey(new_word))//Build Adjacent Graph
                                map.get(new_word).add(word);
                            else{
                                List<String> l= new LinkedList<String>();
                                l.add(word);
                                map.put(new_word, l);
                            }

                            if (new_word.equals(end)&&!found) found=true;

                        }

                    }//End:Iteration from 'a' to 'z'
                }//End:Iteration from the first to the last
                if (curr==0){
                    if (found) break;
                    curr=next;
```

```
                curr=next;
                next=0;
                unvisited.removeAll(visited);
                visited.clear();
            }
        }//End While

        backTrace(end,start);

        return results;
    }
    private void backTrace(String word,String start){
        if (word.equals(start)){
            list.add(0,start);
            results.add(new ArrayList<String>(list));
            list.remove(0);
            return;
        }
        list.add(0,word);
        if (map.get(word)!=null)
            for (String s:map.get(word))
                backTrace(s,start);
        list.remove(0);
    }
}
```

written by reeclapple original link here

Answer 2

In order to reduce the running time, we should use two-end BFS to slove the problem.

Accepted 68ms c++ solution for Word Ladder.

```
class Solution {
public:
    int ladderLength(std::string beginWord, std::string endWord, std::unordered_s
et<std::string> &dict) {
        if (beginWord == endWord)
            return 1;
        std::unordered_set<std::string> words1, words2;
        words1.insert(beginWord);
        words2.insert(endWord);
        dict.erase(beginWord);
        dict.erase(endWord);
        return ladderLengthHelper(words1, words2, dict, 1);
    }

private:
    int ladderLengthHelper(std::unordered_set<std::string> &words1, std::unordere
d_set<std::string> &words2, std::unordered_set<std::string> &dict, int level) {
        if (words1.empty())
            return 0;
        if (words1.size() > words2.size())
            return ladderLengthHelper(words2, words1, dict, level);
        std::unordered_set<std::string> words3;
        for (auto it = words1.begin(); it != words1.end(); ++it) {
            std::string word = *it;
            for (auto ch = word.begin(); ch != word.end(); ++ch) {
                char tmp = *ch;
                for (*ch = 'a'; *ch <= 'z'; ++(*ch))
                    if (*ch != tmp)
                        if (words2.find(word) != words2.end())
                            return level + 1;
                        else if (dict.find(word) != dict.end()) {
                            dict.erase(word);
                            words3.insert(word);
                        }
                *ch = tmp;
            }
        }
        return ladderLengthHelper(words2, words3, dict, level + 1);
    }
};
```

Accepted 88ms c++ solution for Word Ladder II.

```
class Solution {
public:
    std::vector<std::vector<std::string> > findLadders(std::string beginWord, std
::string endWord, std::unordered_set<std::string> &dict) {
        std::vector<std::vector<std::string> > paths;
        std::vector<std::string> path(1, beginWord);
        if (beginWord == endWord) {
            paths.push_back(path);
            return paths;
        }
        std::unordered_set<std::string> words1, words2;
        words1.insert(beginWord);
```

```
        words2.insert(endWord);
        std::unordered_map<std::string, std::vector<std::string> > nexts;
        bool words1IsBegin = false;
        if (findLaddersHelper(words1, words2, dict, nexts, words1IsBegin))
            getPath(beginWord, endWord, nexts, path, paths);
        return paths;
    }
private:
    bool findLaddersHelper(
        std::unordered_set<std::string> &words1,
        std::unordered_set<std::string> &words2,
        std::unordered_set<std::string> &dict,
        std::unordered_map<std::string, std::vector<std::string> > &nexts,
        bool &words1IsBegin) {
        words1IsBegin = !words1IsBegin;
        if (words1.empty())
            return false;
        if (words1.size() > words2.size())
            return findLaddersHelper(words2, words1, dict, nexts, words1IsBegin);
        for (auto it = words1.begin(); it != words1.end(); ++it)
            dict.erase(*it);
        for (auto it = words2.begin(); it != words2.end(); ++it)
            dict.erase(*it);
        std::unordered_set<std::string> words3;
        bool reach = false;
        for (auto it = words1.begin(); it != words1.end(); ++it) {
            std::string word = *it;
            for (auto ch = word.begin(); ch != word.end(); ++ch) {
                char tmp = *ch;
                for (*ch = 'a'; *ch <= 'z'; ++(*ch))
                    if (*ch != tmp)
                        if (words2.find(word) != words2.end()) {
                            reach = true;
                            words1IsBegin ? nexts[*it].push_back(word) : nexts[wo
rd].push_back(*it);
                        }
                        else if (!reach && dict.find(word) != dict.end()) {
                            words3.insert(word);
                            words1IsBegin ? nexts[*it].push_back(word) : nexts[wo
rd].push_back(*it);
                        }
                *ch = tmp;
            }
        }
        return reach || findLaddersHelper(words2, words3, dict, nexts, words1IsBe
gin);
    }
    void getPath(
        std::string beginWord,
        std::string &endWord,
        std::unordered_map<std::string, std::vector<std::string> > &nexts,
        std::vector<std::string> &path,
        std::vector<std::vector<std::string> > &paths) {
        if (beginWord == endWord)
            paths.push_back(path);
        else
            for (auto it = nexts[beginWord].begin(); it != nexts[beginWord].end()
```

```
; ++it) {
                path.push_back(*it);
                getPath(*it, endWord, nexts, path, paths);
                path.pop_back();
            }
    }
};
```

written by prime_tang original link here

Answer 3

```
class Solution:
# @param start, a string
# @param end, a string
# @param dict, a set of string
# @return a list of lists of string
def findLadders(self, start, end, dic):
    dic.add(end)
    level = {start}
    parents = collections.defaultdict(set)
    while level and end not in parents:
        next_level = collections.defaultdict(set)
        for node in level:
            for char in string.ascii_lowercase:
                for i in range(len(start)):
                    n = node[:i]+char+node[i+1:]
                    if n in dic and n not in parents:
                        next_level[n].add(node)
        level = next_level
        parents.update(next_level)
    res = [[end]]
    while res and res[0][0] != start:
        res = [[p]+r for r in res for p in parents[r[0]]]
    return res
```

Every level we use the defaultdict to get rid of the duplicates

written by tusizi original link here

Word Ladder(127)

Answer 1

```
//BFSï¼Œ two-end method
//traverse the path simultaneously from start node and end node, and merge in the
middle
//the speed will increase (logN/2)^2 times compared with one-end method
int ladderLength(string start, string end, unordered_set<string> &dict) {
    unordered_set<string> begSet, endSet, *set1, *set2;
    begSet.insert(start);
    endSet.insert(end);
    int h=1, K=start.size();
    while(!begSet.empty()&&!endSet.empty()){
        if(begSet.size()<=endSet.size()){   //Make the size of two sets close for
optimization
            set1=&begSet;   //set1 is the forward set
            set2=&endSet;   //set2 provides the target node for set1 to search
        }
        else{
            set1=&endSet;
            set2=&begSet;
        }
        unordered_set<string> itmSet;   //intermediate Set
        h++;
        for(auto i=set1->begin();i!=set1->end();i++){
            string cur=*i;
            for(int k=0;k<K;k++){   //iterate the characters in string cur
                char temp=cur[k];
                for(int l=0;l<26;l++){  //try all 26 alphabets
                    cur[k]='a'+l;
                    auto f=set2->find(cur);
                    if(f!=set2->end())return h;
                    f=dict.find(cur);
                    if(f!=dict.end()){
                        itmSet.insert(cur);
                        dict.erase(f);
                    }
                }
                cur[k]=temp;
            }
        }
        swap(*set1, itmSet);
    }
    return 0;
}
```

written by VaultBoy original link here

Answer 2

Well, this problem has a nice BFS structure.

Let's see the example in the problem statement.

`start = "hit"`

`end = "cog"`

`dict = ["hot", "dot", "dog", "lot", "log"]`

Since only one letter can be changed at a time, if we start from `"hit"`, we can only change to those words which have only one different letter from it, like `"hot"`. Putting in graph-theoretic terms, we can say that `"hot"` is a neighbor of `"hit"`.

The idea is simpy to begin from `start`, then visit its neighbors, then the non-visited neighbors of its neighbors... Well, this is just the typical BFS structure.

To simplify the problem, we insert `end` into `dict`. Once we meet `end` during the BFS, we know we have found the answer. We maintain a variable `dist` for the current distance of the transformation and update it by `dist++` after we finish a round of BFS search (note that it should fit the definition of the distance in the problem statement). Also, to avoid visiting a word for more than once, we erase it from `dict` once it is visited.

The code is as follows.

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, unordered_set<string>& wor
dDict) {
        wordDict.insert(endWord);
        queue<string> toVisit;
        addNextWords(beginWord, wordDict, toVisit);
        int dist = 2;
        while (!toVisit.empty()) {
            int num = toVisit.size();
            for (int i = 0; i < num; i++) {
                string word = toVisit.front();
                toVisit.pop();
                if (word == endWord) return dist;
                addNextWords(word, wordDict, toVisit);
            }
            dist++;
        }
    }
private:
    void addNextWords(string word, unordered_set<string>& wordDict, queue<string>
& toVisit) {
        wordDict.erase(word);
        for (int p = 0; p < (int)word.length(); p++) {
            char letter = word[p];
            for (int k = 0; k < 26; k++) {
                word[p] = 'a' + k;
                if (wordDict.find(word) != wordDict.end()) {
                    toVisit.push(word);
                    wordDict.erase(word);
                }
            }
            word[p] = letter;
        }
    }
};
```

The above code can still be speeded up if we also begin from `end`. Once we meet the same word from `start` and `end`, we know we are done. This link provides a nice two-end search solution. I rewrite the code below for better readability. Note that the use of two pointers `phead` and `ptail` save a lot of time. At each round of BFS, depending on the relative size of `head` and `tail`, we point `phead` to the smaller set to reduce the running time.

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, unordered_set<string>& wor
dDict) {
        unordered_set<string> head, tail, *phead, *ptail;
        head.insert(beginWord);
        tail.insert(endWord);
        int dist = 2;
        while (!head.empty() && !tail.empty()) {
            if (head.size() < tail.size()) {
                phead = &head;
                ptail = &tail;
            }
            else {
                phead = &tail;
                ptail = &head;
            }
            unordered_set<string> temp;
            for (auto itr = phead -> begin(); itr != phead -> end(); itr++) {
                string word = *itr;
                wordDict.erase(word);
                for (int p = 0; p < (int)word.length(); p++) {
                    char letter = word[p];
                    for (int k = 0; k < 26; k++) {
                        word[p] = 'a' + k;
                        if (ptail -> find(word) != ptail -> end())
                            return dist;
                        if (wordDict.find(word) != wordDict.end()) {
                            temp.insert(word);
                            wordDict.erase(word);
                        }
                    }
                    word[p] = letter;
                }
            }
            dist++;
            swap(*phead, temp);
        }
        return 0;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

Shouldn't the output of the test case below be 1? Because "a" could directly be changed to "c", which needs only once edit. Input: "a", "c", ["a","b","c"] Output: 1 Expected: 2

written by wywangywy original link here

Longest Consecutive Sequence(128)

Answer 1

We will use HashMap. The key thing is to keep track of the sequence length and store that in the boundary points of the sequence. For example, as a result, for sequence {1, 2, 3, 4, 5}, map.get(1) and map.get(5) should both return 5.

Whenever a new element **n** is inserted into the map, do two things:

1. See if **n - 1** and **n + 1** exist in the map, and if so, it means there is an existing sequence next to **n**. Variables **left** and **right** will be the length of those two sequences, while **0** means there is no sequence and **n** will be the boundary point later. Store **(left + right + 1)** as the associated value to key **n** into the map.
2. Use **left** and **right** to locate the other end of the sequences to the left and right of **n** respectively, and replace the value with the new length.

Everything inside the **for** loop is O(1) so the total time is O(n). Please comment if you see something wrong. Thanks.

```
public int longestConsecutive(int[] num) {
    int res = 0;
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int n : num) {
        if (!map.containsKey(n)) {
            int left = (map.containsKey(n - 1)) ? map.get(n - 1) : 0;
            int right = (map.containsKey(n + 1)) ? map.get(n + 1) : 0;
            // sum: length of the sequence n is in
            int sum = left + right + 1;
            map.put(n, sum);

            // keep track of the max length
            res = Math.max(res, sum);

            // extend the length to the boundary(s)
            // of the sequence
            // will do nothing if n has no neighbors
            map.put(n - left, sum);
            map.put(n + right, sum);
        }
        else {
            // duplicates
            continue;
        }
    }
    return res;
}
```

written by dcheno215 original link here

Answer 2

use a hash map to store boundary information of consecutive sequence for each

element; there 4 cases when a new element i reached:

1) neither i+1 nor i-1 has been seen: m[i]=1;

2) both i+1 and i-1 have been seen: extend m[i+m[i+1]] and m[i-m[i-1]] to each other;

3) only i+1 has been seen: extend m[i+m[i+1]] and m[i] to each other;

4) only i-1 has been seen: extend m[i-m[i-1]] and m[i] to each other.

```
int longestConsecutive(vector<int> &num) {
    unordered_map<int, int> m;
    int r = 0;
    for (int i : num) {
        if (m[i]) continue;
        r = max(r, m[i] = m[i + m[i + 1]] = m[i - m[i - 1]] = m[i + 1] + m[i - 1]
+ 1);
    }
    return r;
}
```

written by mzchen original link here

Answer 3

First turn the input into a *set* of numbers. That takes O(n) and then we can ask in O(1) whether we have a certain number.

Then go through the numbers. If the number n is the start of a streak (i.e., n-1 is not in the set), then test m = n+1, n+2, n+3, ... and stop at the first number m *not* in the set. The length of the streak is then simply m-n and we update our global best with that. Since we check each streak only once, this is overall O(n). This ran in 44 ms on the OJ, one of the fastest Python submissions.

```
class Solution:
    def longestConsecutive(self, nums):
        nums = set(nums)
        best = 0
        for n in nums:
            if n - 1 not in nums:
                m = n + 1
                while m in nums:
                    m += 1
                best = max(best, m - n)
        return best
```

written by StefanPochmann original link here

Sum Root to Leaf Numbers(129)

Answer 1

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int sumNumbers(TreeNode root) {
        if (root == null)
            return 0;
        return sumR(root, 0);
    }
    public int sumR(TreeNode root, int x) {
        if (root.right == null && root.left == null)
            return 10 * x + root.val;
        int val = 0;
        if (root.left != null)
            val += sumR(root.left, 10 * x + root.val);
        if (root.right != null)
            val += sumR(root.right, 10 * x + root.val);
        return val;
    }
}
```

written by potpie original link here

Answer 2

I use recursive solution to solve the problem.

```
public int sumNumbers(TreeNode root) {
    return sum(root, 0);
}

public int sum(TreeNode n, int s){
    if (n == null) return 0;
    if (n.right == null && n.left == null) return s*10 + n.val;
    return sum(n.left, s*10 + n.val) + sum(n.right, s*10 + n.val);
}
```

written by pavel-shlyk original link here

Answer 3

The idea is to do a preorder traversal of the tree. In the preorder traversal, keep track of the value calculated till the current node, let this value be val. For every node, we update the val as val*10 plus nodeâ€™s

data.

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int sumNumbers(TreeNode *root) {
       return  sumNumberUtil(root,0);

    }
    // preorder
    int sumNumberUtil(struct TreeNode* node, int val)
    {
        if(node==NULL)
        return 0;

        val= val*10+node->val;
        if(node->left==NULL && node->right==NULL)
        {
            return val;
        }

        return sumNumberUtil(node->left,val)+sumNumberUtil(node->right, val);
    }
};
```

written by Deepalaxmi original link here

Surrounded Regions(130)

Answer 1

The algorithm is quite simple: Use BFS starting from 'O's on the boundary and mark them as 'B', then iterate over the whole board and mark 'O' as 'X' and 'B' as 'O'.

```
void bfsBoundary(vector<vector<char> >& board, int w, int l)
{
    int width = board.size();
    int length = board[0].size();
    deque<pair<int, int> > q;
    q.push_back(make_pair(w, l));
    board[w][l] = 'B';
    while (!q.empty()) {
        pair<int, int> cur = q.front();
        q.pop_front();
        pair<int, int> adjs[4] = {{cur.first-1, cur.second},
            {cur.first+1, cur.second},
            {cur.first, cur.second-1},
            {cur.first, cur.second+1}};
        for (int i = 0; i < 4; ++i)
        {
            int adjW = adjs[i].first;
            int adjL = adjs[i].second;
            if ((adjW >= 0) && (adjW < width) && (adjL >= 0)
                    && (adjL < length)
                    && (board[adjW][adjL] == 'O')) {
                q.push_back(make_pair(adjW, adjL));
                board[adjW][adjL] = 'B';
            }
        }
    }
}

void solve(vector<vector<char> > &board) {
    int width = board.size();
    if (width == 0) //Add this to prevent run-time error!
        return;
    int length = board[0].size();
    if  (length == 0) // Add this to prevent run-time error!
        return;

    for (int i = 0; i < length; ++i)
    {
        if (board[0][i] == 'O')
            bfsBoundary(board, 0, i);

        if (board[width-1][i] == 'O')
            bfsBoundary(board, width-1, i);
    }

    for (int i = 0; i < width; ++i)
    {
        if (board[i][0] == 'O')
            bfsBoundary(board, i, 0);
```

```
        if (board[i][length-1] == 'O')
            bfsBoundary(board, i, length-1);
    }

    for (int i = 0; i < width; ++i)
    {
        for (int j = 0; j < length; ++j)
        {
            if (board[i][j] == 'O')
                board[i][j] = 'X';
            else if (board[i][j] == 'B')
                board[i][j] = 'O';
        }
    }
}
```

Note that one of the test cases is when the board is empty. So if you don't check it in your code, you will encounter an run-time error.

written by eaglesky1990 original link here

Answer 2

```
class UF
{
private:
    int* id;     // id[i] = parent of i
    int* rank;  // rank[i] = rank of subtree rooted at i (cannot be more than 31)
    int count;    // number of components
public:
    UF(int N)
    {
        count = N;
        id = new int[N];
        rank = new int[N];
        for (int i = 0; i < N; i++) {
            id[i] = i;
            rank[i] = 0;
        }
    }
    ~UF()
    {
        delete [] id;
        delete [] rank;
    }
    int find(int p) {
        while (p != id[p]) {
            id[p] = id[id[p]];    // path compression by halving
            p = id[p];
        }
        return p;
    }
    int getCount() {
        return count;
    }
    bool connected(int p, int q) {
```

```
    bool connected(int p, int q) {
        return find(p) == find(q);
    }
    void connect(int p, int q) {
        int i = find(p);
        int j = find(q);
        if (i == j) return;
        if (rank[i] < rank[j]) id[i] = j;
        else if (rank[i] > rank[j]) id[j] = i;
        else {
            id[j] = i;
            rank[i]++;
        }
        count--;
    }
};

class Solution {
public:
    void solve(vector<vector<char>> &board) {
        int n = board.size();
        if(n==0)    return;
        int m = board[0].size();
        UF uf = UF(n*m+1);

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if((i==0||i==n-1||j==0||j==m-1)&&board[i][j]=='O') // if a 'O' no
de is on the boundry, connect it to the dummy node
                    uf.connect(i*m+j,n*m);
                else if(board[i][j]=='O') // connect a 'O' node to its neighbour
'O' nodes
                {
                    if(board[i-1][j]=='O')
                        uf.connect(i*m+j,(i-1)*m+j);
                    if(board[i+1][j]=='O')
                        uf.connect(i*m+j,(i+1)*m+j);
                    if(board[i][j-1]=='O')
                        uf.connect(i*m+j,i*m+j-1);
                    if(board[i][j+1]=='O')
                        uf.connect(i*m+j,i*m+j+1);
                }
            }
        }

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(!uf.connected(i*m+j,n*m)){ // if a 'O' node is not connected t
o the dummy node, it is captured
                    board[i][j]='X';
                }
            }
        }
    }
};
```

Hi. So here is my accepted code using **Union Find** data structure. The idea comes from the observation that if a region is NOT captured, it is connected to the boundry. So if we connect all the 'O' nodes on the boundry to a dummy node, and then connect each 'O' node to its neighbour 'O' nodes, then we can tell directly whether a 'O' node is captured by checking whether it is connected to the dummy node. For more about Union Find, the first assignment in the algo1 may help: https://www.coursera.org/course/algs4partI

written by jinming.he.5 original link here

Answer 3

- First, check the four border of the matrix. If there is a element is 'O', alter it and all its neighbor 'O' elements to '1'.
- Then ,alter all the 'O' to 'X'
- At last,alter all the '1' to 'O'

For example:

```
X X X X           X X X X             X X X X
X X O X  ->       X X O X    ->       X X X X
X O X X           X 1 X X             X O X X
X O X X           X 1 X X             X O X X
```

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        int i,j;
        int row=board.size();
        if(!row)
            return;
        int col=board[0].size();

        for(i=0;i<row;i++){
            check(board,i,0,row,col);
            if(col>1)
                check(board,i,col-1,row,col);
        }
        for(j=1;j+1<col;j++){
            check(board,0,j,row,col);
            if(row>1)
                check(board,row-1,j,row,col);
        }
        for(i=0;i<row;i++)
            for(j=0;j<col;j++)
                if(board[i][j]=='O')
                    board[i][j]='X';
        for(i=0;i<row;i++)
            for(j=0;j<col;j++)
                if(board[i][j]=='1')
                    board[i][j]='O';
    }
    void check(vector<vector<char> >&vec,int i,int j,int row,int col){
        if(vec[i][j]=='O'){
            vec[i][j]='1';
            if(i>1)
                check(vec,i-1,j,row,col);
            if(j>1)
                check(vec,i,j-1,row,col);
            if(i+1<row)
                check(vec,i+1,j,row,col);
            if(j+1<col)
                check(vec,i,j+1,row,col);
        }
    }
};
```

written by sugeladi original link here

Palindrome Partitioning(131)

Answer 1

```
public class Solution {
    public static List<List<String>> partition(String s) {
        int len = s.length();
        List<List<String>>[] result = new List[len + 1];
        result[0] = new ArrayList<List<String>>();
        result[0].add(new ArrayList<String>());

        boolean[][] pair = new boolean[len][len];
        for (int i = 0; i < s.length(); i++) {
            result[i + 1] = new ArrayList<List<String>>();
            for (int left = 0; left <= i; left++) {
                if (s.charAt(left) == s.charAt(i) && (i-left <= 1 || pair[left +
1][i - 1])) {
                    pair[left][i] = true;
                    String str = s.substring(left, i + 1);
                    for (List<String> r : result[left]) {
                        List<String> ri = new ArrayList<String>(r);
                        ri.add(str);
                        result[i + 1].add(ri);
                    }
                }
            }
        }
        return result[len];
    }
}
```

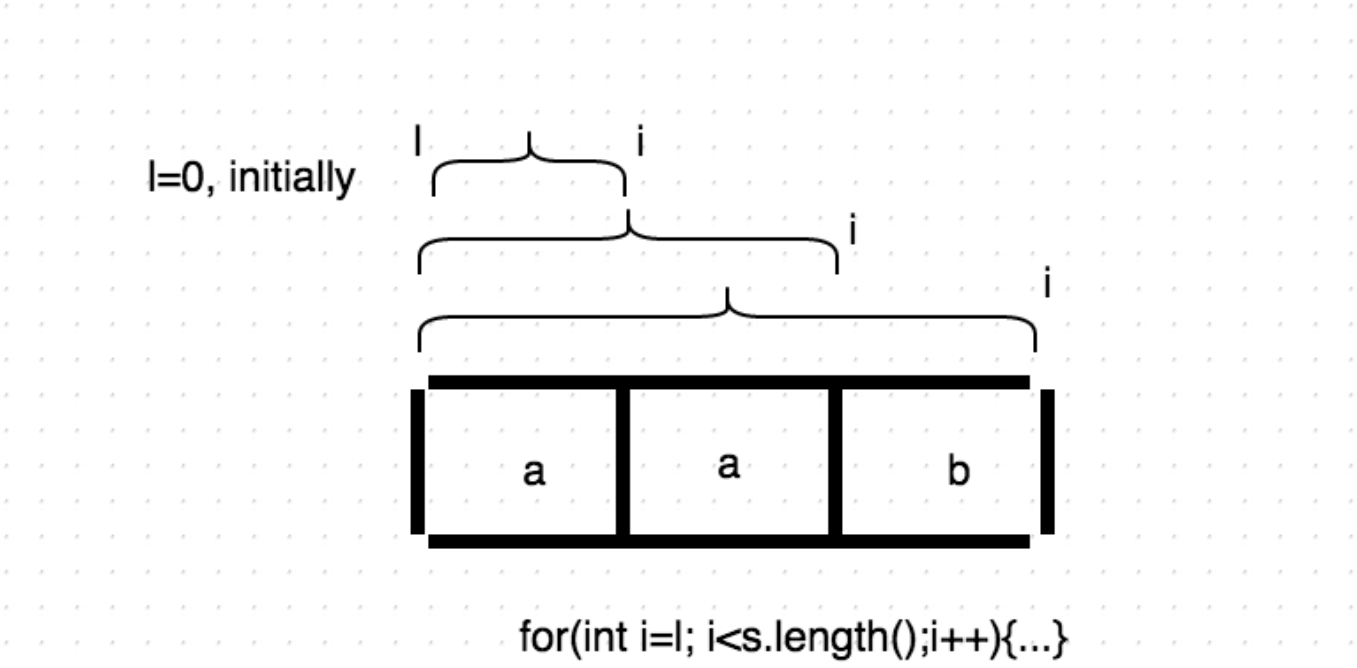
Here the **pair** is to mark a range for the substring is a Pal. if pair[i][j] is true, that means sub string from i to j is pal.

The **result[i]**, is to store from beginng until current index i (Non inclusive), all possible partitions. From the past result we can determine current result.

written by jianwu original link here

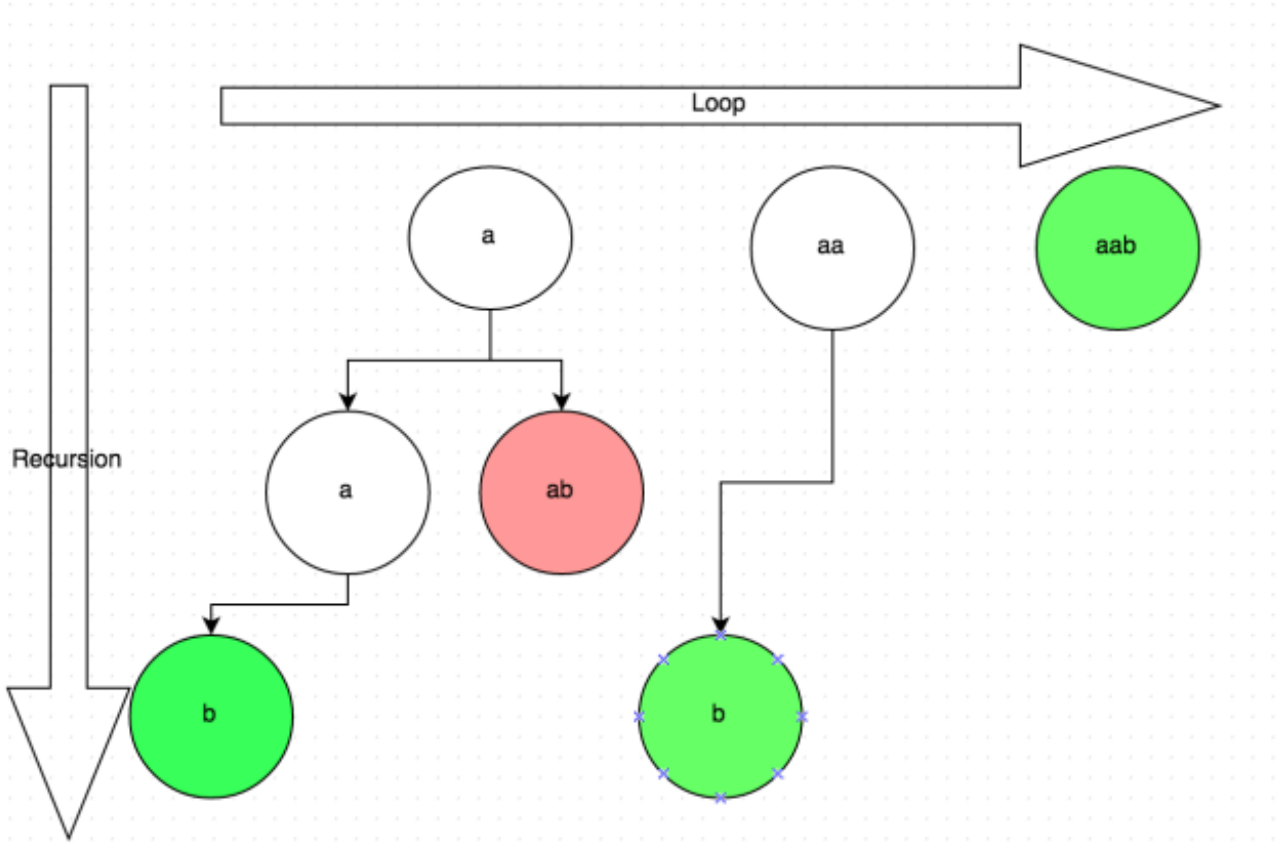
Answer 2

if the input is "aab", check if [0,0] "a" is palindrome. then check [0,1] "aa", then [0,2] "aab". While checking [0,0], the rest of string is "ab", use ab as input to make a recursive call.



in this example, in the loop of i=l+1, a recursive call will be made with input = "ab". Every time a recursive call is made, the position of l move right.

How to define a correct answer? Think about DFS, if the current string to be checked (Palindrome) contains the last position, in this case "c", this path is a correct answer, otherwise, it's a false answer.



line 13: is the boundary to check if the current string contains the last element.

l>=s.length()

```
public class Solution {
        List<List<String>> resultLst;
        ArrayList<String> currLst;
        public List<List<String>> partition(String s) {
            resultLst = new ArrayList<List<String>>();
            currLst = new ArrayList<String>();
            backTrack(s,0);
            return resultLst;
        }
        public void backTrack(String s, int l){
            if(currLst.size()>0 //the initial str could be palindrome
                && l>=s.length()){
                    List<String> r = (ArrayList<String>) currLst.clone();
                    resultLst.add(r);
            }
            for(int i=l;i<s.length();i++){
                if(isPalindrome(s,l,i)){
                    if(l==i)
                        currLst.add(Character.toString(s.charAt(i)));
                    else
                        currLst.add(s.substring(l,i+1));
                    backTrack(s,i+1);
                    currLst.remove(currLst.size()-1);
                }
            }
        }
        public boolean isPalindrome(String str, int l, int r){
            if(l==r) return true;
            while(l<r){
                if(str.charAt(l)!=str.charAt(r)) return false;
                l++;r--;
            }
            return true;
        }
}
```

written by charlie+yupeng original link here

Answer 3

The Idea is simple: loop through the string, check if substr(0, i) is palindrome. If it is, recursively call dfs() on the rest of sub string: substr(i+1, length). keep the current palindrome partition so far in the 'path' argument of dfs(). When reaching the end of string, add current partition in the result.

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string> > ret;
        if(s.empty()) return ret;

        vector<string> path;
        dfs(0, s, path, ret);

        return ret;
    }

    void dfs(int index, string& s, vector<string>& path, vector<vector<string> >&
ret) {
        if(index == s.size()) {
            ret.push_back(path);
            return;
        }
        for(int i = index; i < s.size(); ++i) {
            if(isPalindrome(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
                dfs(i+1, s, path, ret);
                path.pop_back();
            }
        }
    }

    bool isPalindrome(const string& s, int start, int end) {
        while(start <= end) {
            if(s[start++] != s[end--])
                return false;
        }
        return true;
    }
};
```

written by zhangyu917 original link here

Palindrome Partitioning II(132)

Answer 1

```
class Solution {
public:
    int minCut(string s) {
        int n = s.size();
        vector<int> cut(n+1, 0);  // number of cuts for the first k characters
        for (int i = 0; i <= n; i++) cut[i] = i-1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; i-j >= 0 && i+j < n && s[i-j]==s[i+j] ; j++) // odd l
ength palindrome
                cut[i+j+1] = min(cut[i+j+1],1+cut[i-j]);

            for (int j = 1; i-j+1 >= 0 && i+j < n && s[i-j+1] == s[i+j]; j++) //
even length palindrome
                cut[i+j+1] = min(cut[i+j+1],1+cut[i-j+1]);
        }
        return cut[n];
    }
};
```

written by tqlong original link here

Answer 2

Calculate and maintain 2 DP states:

1. pal[i][j] , which is whether s[i..j] forms a pal
2. d[i], which is the minCut for s[i..n-1]

Once we comes to a pal[i][j]==true:

- if j==n-1, the string s[i..n-1] is a Pal, minCut is 0, d[i]=0;
- else: the current cut num (first cut s[i..j] and then cut the rest s[j+1...n-1]) is 1+d[j+1], compare it to the exisiting minCut num d[i], repalce if smaller.

d[0] is the answer.

```
class Solution {
   public:
       int minCut(string s) {
           if(s.empty()) return 0;
           int n = s.size();
           vector<vector<bool>> pal(n,vector<bool>(n,false));
           vector<int> d(n);
           for(int i=n-1;i>=0;i--)
           {
               d[i]=n-i-1;
               for(int j=i;j<n;j++)
               {
                   if(s[i]==s[j] && (j-i<2 || pal[i+1][j-1]))
                   {
                      pal[i][j]=true;
                      if(j==n-1)
                          d[i]=0;
                      else if(d[j+1]+1<d[i])
                          d[i]=d[j+1]+1;
                   }
               }
           }
           return d[0];
       }
   };
```

written by heiyanbin original link here

Answer 3

One typical solution is DP based. Such solution first constructs a two-dimensional bool array isPalin to indicate whether the sub-string s[i..j] is palindrome. To get such array, we need O(N^2) time complexity. Moreover, to get the minimum cuts, we need another array minCuts to do DP and minCuts[i] saves the minimum cuts found for the sub-string s[0..i-1]. minCuts[i] is initialized to i-1, which is the maximum cuts needed (cuts the string into one-letter characters) and minCuts[0] initially sets to -1, which is needed in the case that s[0..i-1] is a palindrome. When we construct isPalin array, we update minCuts everytime we found a palindrome sub-string, i.e. if s[i..j] is a palindrome, then minCuts[j+1] will be updated to the minimum of the current minCuts[j+1] and minCut[i]+1(i.e. cut s[0..j] into s[0,i-1] and s[i,j]). At last, we return minCuts[N]. So the complexity is O(N^2). However, it can be further improved since as described above, we only update minCuts when we find a palindrome substring, while the DP algorithm spends lots of time to calculate isPalin, most of which is false (i.e. not a palindrome substring). If we can reduce such unnecessary calculation, then we can speed up the algorithm. This can be achieved with a Manancher-like solution, which is also given as following.

```
// DP solution
    class Solution {
    public:
        int minCut(string s) {
            const int N = s.size();
            if(N<=1) return 0;
            int i,j;
            bool isPalin[N][N];
            fill_n(&isPalin[0][0], N*N, false);
            int minCuts[N+1];
            for(i=0; i<=N; ++i) minCuts[i] = i-1;

            for(j=1; j<N; ++j)
            {
                for(i=j; i>=0; --i)
                {
                    if( (s[i] == s[j]) && ( ( j-i < 2 ) || isPalin[i+1][j-1] ) )
                    {
                        isPalin[i][j] = true;
                        minCuts[j+1] = min(minCuts[j+1], 1 + minCuts[i]);
                    }
                }
            }
            return minCuts[N];

        }
    };
```

The Manancher-like solution scan the array from left to right (for i loop) and only check those sub-strings centered at s[i]; once a non-palindrome string is found, it will stop and move to i+1. Same as the DP solution, minCUTS[i] is used to save the minimum cuts for s[0:i-1]. For each i, we do two for loops (for j loop) to check if the substrings s[i-j .. i+j] (odd-length substring) and s[i-j-1.. i+j] (even-length substring) are palindrome. By increasing j from 0, we can find all the palindrome sub-strings centered at i and update minCUTS accordingly. Once we meet one non-palindrome sub-string, we stop for-j loop since we know there no further palindrome substring centered at i. This helps us avoid unnecessary palindrome substring checks, as we did in the DP algorithm. Therefore, this version is faster.

```
//Manancher-like solution
class Solution {
public:
    int minCut(string s) {
        const int N = s.size();
        if(N<=1) return 0;

        int i, j, minCUTS[N+1];
        for(i=0; i<=N; ++i) minCUTS[i] = i-1;

        for(i=1;i<N;i++)
        {
            for(j=0;(i-j)>=0 && (i+j)<N && s[i-j]== s[i+j]; ++j) // odd-length su
bstrings
                minCUTS[i+j+1] = min(minCUTS[i+j+1], 1 + minCUTS[i-j]);

            for(j=0;(i-j-1)>=0 && (i+j)<N && s[i-j-1]== s[i+j]; ++j) // even-leng
th substrings
                minCUTS[i+j+1] = min(minCUTS[i+j+1], 1 + minCUTS[i-j-1]);
        }
        return minCUTS[N];
    }
};
```

written by dong.wang.1694 original link here

Clone Graph(133)

Answer 1

```
public class Solution {
    private HashMap<Integer, UndirectedGraphNode> map = new HashMap<>();
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        return clone(node);
    }

    private UndirectedGraphNode clone(UndirectedGraphNode node) {
        if (node == null) return null;

        if (map.containsKey(node.label)) {
            return map.get(node.label);
        }
        UndirectedGraphNode clone = new UndirectedGraphNode(node.label);
        map.put(clone.label, clone);
        for (UndirectedGraphNode neighbor : node.neighbors) {
            clone.neighbors.add(clone(neighbor));
        }
        return clone;
    }
}
```

written by mohamed+ebrahim original link here

Answer 2

The solution is same as https://oj.leetcode.com/discuss/22244/simple-c-solution-using-dfs-and-recursion I just make it shorter;

```
/**
*  author : s2003zy
*  weibo  : http://weibo.com/songzy982
*  blog   : s2003zy.com
*  date   : 2015.02.27
*/
class Solution {
public:
    unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> hash;
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
       if (!node) return node;
       if(hash.find(node) == hash.end()) {
           hash[node] = new UndirectedGraphNode(node -> label);
           for (auto x : node -> neighbors) {
                (hash[node] -> neighbors).push_back( cloneGraph(x) );
           }
       }
       return hash[node];
    }
};
```

written by s2003zy original link here

Answer 3

Use HashMap to look up nodes and add connection to them while performing BFS.

```
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) return null;

        UndirectedGraphNode newNode = new UndirectedGraphNode(node.label); //new
node for return
        HashMap<Integer, UndirectedGraphNode> map = new HashMap(); //store visite
d nodes

        map.put(newNode.label, newNode); //add first node to HashMap

        LinkedList<UndirectedGraphNode> queue = new LinkedList(); //to store **or
iginal** nodes need to be visited
        queue.add(node); //add first **original** node to queue

        while (!queue.isEmpty()) { //if more nodes need to be visited
            UndirectedGraphNode n = queue.pop(); //search first node in the queue
            for (UndirectedGraphNode neighbor : n.neighbors) {
                if (!map.containsKey(neighbor.label)) { //add to map and queue if
this node hasn't been searched before
                    map.put(neighbor.label, new UndirectedGraphNode(neighbor.labe
l));
                    queue.add(neighbor);
                }
                map.get(n.label).neighbors.add(map.get(neighbor.label)); //add ne
ighbor to new created nodes
            }
        }

        return newNode;
    }
}
```

written by shu3 original link here

Gas Station(134)

Answer 1

I have thought for a long time and got two ideas:

- If car starts at A and can not reach B. Any station between A and B can not reach B.(B is the first station that A can not reach.)
- If the total number of gas is bigger than the total number of cost. There must be a solution.
- (Should I prove them?)

Here is my solution based on those ideas:

```
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
        int start(0),total(0),tank(0);
        //if car fails at 'start', record the next station
        for(int i=0;i<gas.size();i++) if((tank=tank+gas[i]-cost[i])<0) {start=i+1
;total+=tank;tank=0;}
        return (total+tank<0)? -1:start;
    }
};
```

written by daxianji007 original link here

Answer 2

I have got one solution to this problem. I am not sure whether somebody has already posted this solution.

```
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {

       int start = gas.size()-1;
       int end = 0;
       int sum = gas[start] - cost[start];
       while (start > end) {
          if (sum >= 0) {
             sum += gas[end] - cost[end];
             ++end;
          }
          else {
             --start;
             sum += gas[start] - cost[start];
          }
       }
       return sum >= 0 ? start : -1;
    }
};
```

written by xuewuxiao original link here

Answer 3

```
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
        int i, j, n = gas.size();

        /*
         * If start from i, stop before station x -> no station k from i + 1 to x
- 1 can reach x.
         * Bcoz if so, i can reach k and k can reach x, then i reaches x. Contrad
iction.
         * Thus i can jump directly to x instead of i + 1, bringing complexity fr
om O(n^2) to O(n).
         */
        // start from station i
        for (i = 0; i < n; i += j) {
            int gas_left = 0;
            // forward j stations
            for (j = 1; j <= n; j++) {
                int k = (i + j - 1) % n;
                gas_left += gas[k] - cost[k];
                if (gas_left < 0)
                    break;
            }
            if (j > n)
                return i;
        }

        return -1;
    }
};
```

written by xiaohui7 original link here

Candy(135)

Answer 1

```
 int candy(vector<int> &ratings)
{
    int size=ratings.size();
    if(size<=1)
        return size;
    vector<int> num(size,1);
    for (int i = 1; i < size; i++)
    {
        if(ratings[i]>ratings[i-1])
            num[i]=num[i-1]+1;
    }
    for (int i= size-1; i>0 ; i--)
    {
        if(ratings[i-1]>ratings[i])
            num[i-1]=max(num[i]+1,num[i-1]);
    }
    int result=0;
    for (int i = 0; i < size; i++)
    {
        result+=num[i];
       // cout<<num[i]<<" ";
    }
    return result;
}
```

written by 1145520074 original link here

Answer 2

Hi guys!

This solution picks each element from the input array only once. First, we give a candy to the first child. Then for each child we have three cases:

1. His/her rating is equal to the previous one -> give 1 candy.
2. His/her rating is greater than the previous one -> give him (previous + 1) candies.
3. His/her rating is less than the previous one -> don't know what to do yet, let's just count the number of such consequent cases.

When we enter 1 or 2 condition we can check our count from 3. If it's not zero then we know that we were descending before and we have everything to update our total candies amount: number of children in descending sequence of raitings - coundDown, number of candies given at peak - prev (we don't update prev when descending). Total number of candies for "descending" children can be found through arithmetic progression formula (1+2+...+countDown). Plus we need to update our peak child if his number of candies is less then or equal to countDown.

Here's a pretty concise code below.

```java
public class Solution {
    public int candy(int[] ratings) {
        if (ratings == null || ratings.length == 0) return 0;
        int total = 1, prev = 1, countDown = 0;
        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] >= ratings[i-1]) {
                if (countDown > 0) {
                    total += countDown*(countDown+1)/2; // arithmetic progression
                    if (countDown >= prev) total += countDown - prev + 1;
                    countDown = 0;
                    prev = 1;
                }
                prev = ratings[i] == ratings[i-1] ? 1 : prev+1;
                total += prev;
            } else countDown++;
        }
        if (countDown > 0) { // if we were descending at the end
            total += countDown*(countDown+1)/2;
            if (countDown >= prev) total += countDown - prev + 1;
        }
        return total;
    }
}
```

Have a nice coding!

written by shpolsky original link here

Answer 3

The question requires us to make sure a child with a higher rate has more candies than its left and right neighbors. One simple solution is to do two scans: one foward scan (from 1 to N-1) to make sure child i has more candies than its left neighbor if its rate is higher than its left neighbor. After the forward scan, we can guarantee that the left neighbor relationship is correct but we have to do more to make the right neighbor relationship in order; so we do the backwarad scan (from N-2 to 0) to make child i has more candies than its right neighbor i+1 if its rate is higher than its right neighbor. In the following implementation, we need a O(N) array number to save the number of candies needed for children, so it has O(N) space complexity and we do two linear scans so the time complexity is O(N)

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int len = ratings.size(), res = 0, i;
        if(len>0)
        {
            vector<int> number(len,0); // to save the number of candies for child
[0:N-1]
            number[0] = 1;
// forward scan to calculate how many candies needed for child i to make sure it
has more candies than its left neighbor if it has a higher rate, otherwise, give o
ne candy to it
            for(i=1; i<len;++i) number[i] = ratings[i]>ratings[i-1]?number[i-1]+1
:1;

// backward scan to calculate to make sure child i has more candies than its right
neighbor if it has a higher rate, pick the bigger one from forward and backward sc
ans as the final number for child i
            for(i=len-2, res = number[len-1]; i>=0;--i)
            {
                if( (ratings[i]>ratings[i+1]) && number[i]<(number[i+1]+1) ) numb
er[i] = number[i+1]+1;
                res += number[i];
            }
        }
        return res;
    }
};
```

Now, the question is can we do better? Do we really need two scans? If we do only forward scan, then the problem is we can not guarantee the right neighbor relationship holds. i.e. we don't know if the following order is descending (i>i+1>i+2>...). and that may cause issues. To fix that, we will detect the dips (the points at which the order switchs from increasing to decreasng). We will make sure all the local dips (minimum points) has only one candy and update its previous neighbors (which has hgher rates than its rate) accordingly. To do such update, we need to know when the decrease starts, so we use pPos to save that starting points. So the solution becomes: do the forward scan, if it is in an increasing order (child i rate > child i-1 order), check if it is a local dip (neg_peak == true): if so, update the candy number to make sure child i-1 has one candy. if not, just give one more candy to child i. If it is in an decreasing order (child i rate < child i-1 order) , just give one less candy to i. don't forget at last, we still need to make sure child N-1 has one or more candy. So O(1) space , O(N) time

```
    class Solution {
    public:
        int candy(vector<int>& ratings) {
            const int len = ratings.size();
            if(len<=1) return len;

            int i, pPos, res=1, peak=1; // peak: # candies given to the i-1 child
            bool neg_peak = false; // flag to indicate if it is a local dip
            for(i=1; i<len;i++)
            {
                if(ratings[i] >= ratings[i-1])
                {   // it is increasing
                    if(neg_peak)
                    {  // it is a local dip, we need to make sure i-1 has one can
dy
                        res -= (peak-1) * (i-pPos - (peak>0));
                        peak = 1;
                        neg_peak = false;
                    }
                   // update child i candy number, if equal, set to 1
                    peak = (ratings[i] == ratings[i-1])? 1:++peak;
                    res += peak;
                }
                else
                { // decreasing, just give one less candy, if it is the starting
point of a decrease, update pPos
                    if(!neg_peak) {pPos = i-1; neg_peak = true;}
                    res += --peak;
                }
            }
// don't forget to update res, if the last one is a local dip
            return !neg_peak? res : res - (peak-1) * (i-pPos - (peak>0));

        }
    };
```

written by dong.wang.1694 original link here

Single Number(136)

Answer 1

known that A XOR A = 0 and the XOR operator is commutative, the solution will be very straightforward. `

```
int singleNumber(int A[], int n) {
    int result = 0;
    for (int i = 0; i<n; i++)
    {
        result ^=A[i];
    }
    return result;
}
```

`

written by Ivantsang original link here

Answer 2

Logic: XOR will return 1 only on two different bits. So if two numbers are the same, XOR will return 0. Finally only one number left. A ^ A = 0 and A ^ B ^ A = B.

```
class Solution {
    public:
        int singleNumber(int A[], int n) {
            int result=A[0];
            for(int i=1;i<n;i++)
            {
                result= result^A[i];  /* Get the xor of all elements */
            }
            return result;
        }
    };
```

written by Deepalaxmi original link here

Answer 3

XOR of two equal numbers is 0 : a^a=0. This is the main idea of the algorithm.

```
class Solution {
    public:
        int singleNumber(int A[], int n) {
            for (int i = 1; i < n; ++i)
                A[0] ^= A[i];
             return A[0];
        }
    };
```

written by dkotsur original link here

Single Number II(137)

Answer 1

```
public int singleNumber(int[] A) {
    int ones = 0, twos = 0;
    for(int i = 0; i < A.length; i++){
        ones = (ones ^ A[i]) & ~twos;
        twos = (twos ^ A[i]) & ~ones;
    }
    return ones;
}
```

written by againest1 original link here

Answer 2

Statement of our problem: "Given an array of integers, every element appears k (k >1) times except for one, which appears p times(p>=1, p % k != 0). Find that single one."

As others pointed out, in order to apply the bitwise operations, we should rethink how integers are represented in computers -- by bits. To start, let's consider only one bit for now. Suppose we have an array of 1-bit numbers (which can only be 0 or 1), we'd like to count the number of '1's in the array such that whenever the counted number of '1' reaches a certain value, say k, the count returns to zero and starts over (In case you are curious, this k will be the same as the one in the problem statement above). To keep track of how many '1's we have encountered so far, we need a counter. Suppose the counter has m bits in binary form: xm, ..., x1 (from most significant bit to least significant bit). We can conclude at least the following four properties of the counter:

1. There is an initial state of the counter, which for simplicity is zero;
2. For each input from the array, if we hit a '0', the counter should remain unchanged;
3. For each input from the array, if we hit a '1', the counter should increase by one;
4. In order to cover k counts, we require 2^m >= k, which implies m >= logk.

Here is the key part: how each bit in the counter (x1 to xm) changes as we are scanning the array. Note we are prompted to use bitwise operations. In order to satisfy the second property, recall what bitwise operations will not change the operand if the other operand is 0? Yes, you got it: x = x | 0 and x = x ^ 0.

Okay, we have an expression now: x = x | i or x = x ^ i, where i is the scanned element from the array. Which one is better? We don't know yet. So, let's just do the actual counting:

At the beginning, all bits of the counter is initialized to zero, i.e., xm = 0, ..., x1 = 0. Since we are gonna choose bitwise operations that guarantees all bits of the counter remain unchanged if we hit '0's, the counter will be 0 until we hit the first '1' in the array. After we hit the first '1', we got: xm = 0, ...,x2 = 0, x1 = 1. Let's continue until

we hit the second '1', after which we have: xm = 0, ..., x2 = 1, x1 = 0. Note that x1 changes from 1 to 0. For x1 = x1 | i, after the second count, x1 will still be 1. So it's clear we should use x1 = x1 ^ i. What about x2, ..., xm? The idea is to find the condition under which x2, ..., xm will change their values. Take x2 as an example. If we hit a '1', and we need to change the value of x2, what must the value of x1 be before we do the change? The answer is: x1 must be 1 otherwise we shouldn't change x2 because changing x1 from 0 to 1 will do the job. So x2 will change only if x1 and i are both 1, or mathematically, x2 = x2 ^ (x1 & i). Similarly xm will change only when xm-1, ..., x1 and i are all 1: xm = xm ^ (xm-1 & ... & x1 & i); Bingo, we've found the bitwise operations!

However, you may notice that the bitwise operations found above will count from 0 until 2^m - 1, instead of k. If k < 2^m - 1, we need some "cutting" mechanism to reinitialize the counter to 0 when the count reaches k. To this end, we apply bitwise AND to xm,..., x1 with some variable called mask, i.e., xm = xm & mask, ..., x1 = x1 & mask. If we can make sure the mask will be 0 only when the count reaches k and be 1 for all other count cases, then we are done. How do we achieve that? Try to think what distinguishes the case with k count from all other count cases. Yes, it's the count of '1's! For each count, we have unique values for each bit of the counter, which can be regarded as its state. If we write k in its binary form: km,..., k1. we can construct the mask as follows:

mask = ~(x1' & x2' & ... xm'), where xj' = xj if kj = 1 and xj' = ~xj if kj = 0 (j = 1 to m).

Let's do some examples:

k = 3: k1 = 1, k2 = 1, mask = ~(x1 & x2);

k = 5: k1 = 1, k2 = 0, k3 = 1, mask = ~(x1 & ~x2 & x3);

In summary, our algorithm will go like this:

for (int i : array) {

xm ^= (xm-1 & ... & x1 & i);

xm-1 ^= (xm-2 & ... & x1 & i);

.....

x1 ^= i;

mask = ~(x1' & x2' & ... xm') where xj' = xj if kj = 1 and xj' = ~xj if kj = 0 (j = 1 to m).

xm &= mask;

.....

x1 &= mask;

}

Now it's time to generalize our results from 1-bit number case to 32-bit integers. One straightforward way would be creating 32 counters for each bit in the integer. You've probably already seen this in other posted codes. But if we take advantage of

bitwise operations, we may be able to manage all the 32 counters "collectively". By saying "collectively" we mean using m 32-bit integers instead of 32 m-bit counters, where m is the minimum integer that satisfies m >= logk. The reason is that bitwise operations apply only to each bit so operations on different bits are independent of each other(kind obvious, right?). This allows us to group the corresponding bits of the 32 counters into one 32-bit integer. Since each counter has m bits, we end up with m 32-bit integers. Therefore, in the algorithm developed above, we just need to regard x1 to xm as 32-bit integers instead of 1-bit numbers and we are done. Easy, hum?

The last thing is what value we should return, or equivalently which one of x1 to xm will equal the single element. To get the correct answer, we need to understand what the m 32-bit integers x1 to xm represent. Take x1 as an example. x1 has 32 bits and let's label them as r (r = 1 to 32), After we are done scanning the input array, the value for the r-th bit of x1 will be determined by the r-th bit of all the elements in the array (more specifically, suppose the total count of '1' for the r-th bit of all the elements in the array is q, q' = q % k and in its binary form: q'm,...,q'1, then by definition the r-th bit of x1 will equal q'1). Now you can ask yourself this question: what does it imply if the r-th bit of x1 is '1'?

The answer is to find what can contribute to this '1'. Will an element that appears k times contribute? No. Why? Because for an element to contribute, it has to satisfy at least two conditions at the same time: the r-th bit of this element is '1' and the number of appearance of this '1' is not an integer multiple of k. The first condition is trivial. The second comes from the fact that whenever the number of '1' hit is k, the counter will go back to zero, which means the corresponding bit in x1 will be set to 0. For an element that appears k times, it's impossible to meet these two conditions simultaneously so it won't contribute. At last, only the single element which appears p (p%k != 0) times will contribute. If p > k, then the first [p/k] (denotes the integer part of p/k) single elements won't contribute either. Then we can always set p' = p % k and say the single element appears effectively p' times.

Let's write p' in its binary form: p'm, ..., p'1. (note that p' < k, so it will fit into m bits). Here I claim the condition for x1 to equal the single element is p'1 = 1. Quick proof: if the r-th bit of x1 is '1', we can safely say the r-th bit of the single element is also '1'. We are left to prove that if the r-th bit of x1 is '0', then the r-th bit of the single element can only be '0'. Just suppose in this case the r-th bit of the single element is '1', let's see what will happen. At the end of the scan, this '1' will be counted p' times. If we write p' in its binary form: p'm, ..., p'1, then by definition the r-th bit of x1 will equal p'1, which is '1'. This contradicts with the presumption that the r-th bit of x1 is '0'. Since this is true for all bits in x1, we can conclude x1 will equal the single element if p'1 = 1. Similarly we can show xj will equal the single element if p'j = 1(j = 1 to m). Now it's clear what we should return. Just express p' = p % k in its binary form, and return any of the corresponding xj as long as p'j = 1.

In total, the algorithm will run in O(n*logk) time and O(logk) space.

Hope this helps!

written by fun4LeetCode original link here

Answer 3

this kind of question the key idea is design a counter that record state. the problem can be every one occurs K times except one occurs M times. for this question, K =3 ,M = 1(or 2) . so to represent 3 state, we need two bit. let say it is a and b, and c is the incoming bit. then we can design a table to implement the state move.

```
current   incoming  next
a b            c    a b
0 0            0    0 0
0 1            0    0 1
1 0            0    1 0
0 0            1    0 1
0 1            1    1 0
1 0            1    0 0
```

like circuit design, we can find out what the next state will be with the incoming bit.(we only need find the ones) then we have for a to be 1, we have

```
    current   incoming  next
    a b            c    a b
    1 0            0    1 0
    0 1            1    1 0
```

and this is can be represented by

```
a=a&~b&~c + ~a&b&c
```

and b can do the same we , and we find that

```
b= ~a&b&~c+~a&~b&c
```

and this is the final formula of a and b and just one of the result set, because for different state move table definition, we can generate different formulas, and this one is may not the most optimised. as you may see other's answer that have a much simple formula, and that formula also corresponding to specific state move table. (if you like ,you can reverse their formula to a state move table, just using the same way but reversely)

for this questions we need to find the except one as the question don't say if the one appears one time or two time , so for ab both

```
01 10 => 1
00 => 0
```

we should return a|b; this is the key idea , we can design any based counter and find the occurs any times except one . here is my code. with comment.

```java
public class Solution {

    public int singleNumber(int[] nums) {
        //we need to implement a tree-time counter(base 3) that if a bit appears three time ,it will be zero.
        //#curent  income  ouput
        //# ab      c/c       ab/ab
        //# 00      1/0       01/00
        //# 01      1/0       10/01
        //# 10      1/0       00/10
        // a=~abc+a~b~c;
        // b=~a~bc+~ab~c;
        int a=0;
        int b=0;
        for(int c:nums){
            int ta=(~a&b&c)|(a&~b&~c);
            b=(~a&~b&c)|(~a&b&~c);
            a=ta;
        }
        //we need find the number that is 01,10 => 1, 00 => 0.
        return a|b;

    }
}
```

this is a general solution . and it comes from the Circuit Design on course digital logic.

written by ziyihao original link here

Copy List with Random Pointer(138)

Answer 1

An intuitive solution is to keep a hash table for each node in the list, via which we just need to iterate the list in 2 rounds respectively to create nodes and assign the values for their random pointers. As a result, the space complexity of this solution is `O(N)`, although with a linear time complexity.

As an optimised solution, we could reduce the space complexity into constant. ***The idea is to associate the original node with its copy node in a single linked list. In this way, we don't need extra space to keep track of the new nodes.***

The algorithm is composed of the follow three steps which are also 3 iteration rounds.

1. Iterate the original list and duplicate each node. The duplicate of each node follows its original immediately.
2. Iterate the new list and assign the random pointer for each duplicated node.
3. Restore the original list and extract the duplicated nodes.

The algorithm is implemented as follows:

```
public RandomListNode copyRandomList(RandomListNode head) {
    RandomListNode iter = head, next;

    // First round: make copy of each node,
    // and link them together side-by-side in a single list.
    while (iter != null) {
        next = iter.next;

        RandomListNode copy = new RandomListNode(iter.label);
        iter.next = copy;
        copy.next = next;

        iter = next;
    }

    // Second round: assign random pointers for the copy nodes.
    iter = head;
    while (iter != null) {
        if (iter.random != null) {
            iter.next.random = iter.random.next;
        }
        iter = iter.next.next;
    }

    // Third round: restore the original list, and extract the copy list.
    iter = head;
    RandomListNode pseudoHead = new RandomListNode(0);
    RandomListNode copy, copyIter = pseudoHead;

    while (iter != null) {
        next = iter.next.next;

        // extract the copy
        copy = iter.next;
        copyIter.next = copy;
        copyIter = copy;

        // restore the original list
        iter.next = next;

        iter = next;
    }

    return pseudoHead.next;
}
```

written by liaison original link here

Answer 2

The idea is: Step 1: create a new node for each existing node and join them together eg: A->B->C will be A->A'->B->B'->C->C'

Step2: copy the random links: for each new node n', n'.random = n.random.next

Step3: detach the list: basically n.next = n.next.next; n'.next = n'.next.next

Here is the code:

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        if(head==null){
            return null;
        }
        RandomListNode n = head;
        while (n!=null){
            RandomListNode n2 = new RandomListNode(n.label);
            RandomListNode tmp = n.next;
            n.next = n2;
            n2.next = tmp;
            n = tmp;
        }

        n=head;
        while(n != null){
            RandomListNode n2 = n.next;
            if(n.random != null)
                n2.random = n.random.next;
            else
                n2.random = null;
            n = n.next.next;
        }

        //detach list
        RandomListNode n2 = head.next;
        n = head;
        RandomListNode head2 = head.next;
        while(n2 != null && n != null){
            n.next = n.next.next;
            if (n2.next == null){
                break;
            }
            n2.next = n2.next.next;

            n2 = n2.next;
            n = n.next;
        }
        return head2;

    }
}
```

written by sharon2 original link here

Answer 3

```
//
// Here's how the 1st algorithm goes.
// Consider l1 as a node on the 1st list and l2 as the corresponding node on 2nd l
ist.
// Step 1:
// Build the 2nd list by creating a new node for each node in 1st list.
// While doing so, insert each new node after it's corresponding node in the 1st l
ist.
// Step 2:
// The new head is the 2nd node as that was the first inserted node.
// Step 3:
// Fix the random pointers in the 2nd list: (Remember that l1->next is actually l2
)
// l2->random will be the node in 2nd list that corresponds l1->random,
// which is next node of l1->random.
// Step 4:
// Separate the combined list into 2: Splice out nodes that are part of second lis
t.
// Return the new head that we saved in step 2.
//

RandomListNode *copyRandomList(RandomListNode *head) {
    RandomListNode *newHead, *l1, *l2;
    if (head == NULL) return NULL;
    for (l1 = head; l1 != NULL; l1 = l1->next->next) {
        l2 = new RandomListNode(l1->label);
        l2->next = l1->next;
        l1->next = l2;
    }

    newHead = head->next;
    for (l1 = head; l1 != NULL; l1 = l1->next->next) {
        if (l1->random != NULL) l1->next->random = l1->random->next;
    }

    for (l1 = head; l1 != NULL; l1 = l1->next) {
        l2 = l1->next;
        l1->next = l2->next;
        if (l2->next != NULL) l2->next = l2->next->next;
    }

    return newHead;
}


//
// Here's how the 2nd algorithm goes.
// Consider l1 as a node on the 1st list and l2 as the corresponding node on 2nd l
ist.
// Step 1:
// Build the 2nd list by creating a new node for each node in 1st list.
// While doing so, set the next pointer of the new node to the random pointer
```

```
// of the corresponding node in the 1st list.  And set the random pointer of the
// 1st list's node to the newly created node.
// Step 2:
// The new head is the node pointed to by the random pointer of the 1st list.
// Step 3:
// Fix the random pointers in the 2nd list: (Remember that l1->random is l2)
// l2->random will be the node in 2nd list that corresponds to the node in the
// 1st list that is pointed to by l2->next,
// Step 4:
// Restore the random pointers of the 1st list and fix the next pointers of the
// 2nd list. random pointer of the node in 1st list is the next pointer of the
// corresponding node in the 2nd list.  This is what we had done in the
// 1st step and now we are reverting back. next pointer of the node in
// 2nd list is the random pointer of the node in 1st list that is pointed to
// by the next pointer of the corresponding node in the 1st list.
// Return the new head that we saved in step 2.
//

RandomListNode *copyRandomList(RandomListNode *head) {
    RandomListNode *newHead, *l1, *l2;
    if (head == NULL) return NULL;

    for (l1 = head; l1 != NULL; l1 = l1->next) {
        l2 = new RandomListNode(l1->label);
        l2->next = l1->random;
        l1->random = l2;
    }

    newHead = head->random;
    for (l1 = head; l1 != NULL; l1 = l1->next) {
        l2 = l1->random;
        l2->random = l2->next ? l2->next->random : NULL;
    }

    for (l1 = head; l1 != NULL; l1 = l1->next) {
        l2 = l1->random;
        l1->random = l2->next;
        l2->next = l1->next ? l1->next->random : NULL;
    }

    return newHead;
}
```

written by satyakam original link here

Word Break(139)

Answer 1

```
public class Solution {
    public boolean wordBreak(String s, Set<String> dict) {

        boolean[] f = new boolean[s.length() + 1];

        f[0] = true;


        /* First DP
        for(int i = 1; i <= s.length(); i++){
            for(String str: dict){
                if(str.length() <= i){
                    if(f[i - str.length()]){
                        if(s.substring(i-str.length(), i).equals(str)){
                            f[i] = true;
                            break;
                        }
                    }
                }
            }
        }*/

        //Second DP
        for(int i=1; i <= s.length(); i++){
            for(int j=0; j < i; j++){
                if(f[j] && dict.contains(s.substring(j, i))){
                    f[i] = true;
                    break;
                }
            }
        }

        return f[s.length()];
    }
}
```

written by segfault original link here

Answer 2

We use a boolean vector dp[]. dp[***i***] is set to true if a valid word (word sequence) ends there. The optimization is to look from current position ***i*** back and only substring and do dictionary look up in case the preceding position ***j*** with *dp[**j**]* == *true* is found.

```
bool wordBreak(string s, unordered_set<string> &dict) {
        if(dict.size()==0) return false;

        vector<bool> dp(s.size()+1,false);
        dp[0]=true;

        for(int i=1;i<=s.size();i++)
        {
            for(int j=i-1;j>=0;j--)
            {
                if(dp[j])
                {
                    string word = s.substr(j,i-j);
                    if(dict.find(word)!= dict.end())
                    {
                        dp[i]=true;
                        break; //next i
                    }
                }
            }
        }

        return dp[s.size()];
    }
```

written by paul7 original link here

Answer 3

People have posted elegant solutions using DP. The solution I post below using BFS is no better than those. Just to share some new thoughts.

We can use a graph to represent the possible solutions. The vertices of the graph are simply the positions of the first characters of the words and each edge actually represents a word. For example, the input string is "nightmare", there are two ways to break it, "night mare" and "nightmare". The graph would be

0-->5-->9

|__ __ _^

The question is simply to check if there is a path from 0 to 9. The most efficient way is traversing the graph using BFS with the help of a queue and a hash set. The hash set is used to keep track of the visited nodes to avoid repeating the same work.

For this problem, the time complexity is O(n^2) and space complexity is O(n), the same with DP. This idea can be used to solve the problem word break II. We can simple construct the graph using BFS, save it into a map and then find all the paths using DFS.

```
bool wordBreak(string s, unordered_set<string> &dict) {
    // BFS
    queue<int> BFS;
    unordered_set<int> visited;

    BFS.push(0);
    while(BFS.size() > 0)
    {
        int start = BFS.front();
        BFS.pop();
        if(visited.find(start) == visited.end())
        {
            visited.insert(start);
            for(int j=start; j<s.size(); j++)
            {
                string word(s, start, j-start+1);
                if(dict.find(word) != dict.end())
                {
                    BFS.push(j+1);
                    if(j+1 == s.size())
                        return true;
                }
            }
        }
    }

    return false;
}
```

written by GuaGua original link here

Word Break II(140)

Answer 1

firstly I used DP from head of the string to traverse the dp-map: and then got a â€œTime Limit Exceededâ€ Error with the unpassed case "aaaaaaaaa....ab", but this method can pass such case like "baaaaaa....a"

secondly I found the answer on internet with the dp-strategy, and saw the dp-method from tail of the string to traverse the dp-map, then got an "Accepted" ,but i tested the case like "baaaaaa....a" on my own computer ,finally the result is â€œTime Limit Exceededâ€

above all, i think the two strategies are the same ; and the OJ's test cases may have some infulence on different methods!

written by CodingGod original link here

Answer 2

```
public class Solution {
public List<String> wordBreak(String s, Set<String> dict) {
    List<String> result = new ArrayList<String>();
    for(int j = s.length() - 1; j >= 0; j--){
        if(dict.contains(s.substring(j)))
            break;
        else{
            if(j == 0)
                return result;
        }
    }
    for(int i = 0; i < s.length()-1; i++)
    {
        if(dict.contains(s.substring(0,i+1)))
        {
            List<String> strs = wordBreak(s.substring(i+1,s.length()),dict);
            if(strs.size() != 0)
                for(Iterator<String> it = strs.iterator();it.hasNext();)
                {
                    result.add(s.substring(0,i+1)+" "+it.next());
                }
        }
    }
    if(dict.contains(s)) result.add(s);
    return result;
}
```

}

written by XingLiu original link here

Answer 3

```
class Solution {
    unordered_map<string, vector<string>> m;

    vector<string> combine(string word, vector<string> prev){
        for(int i=0;i<prev.size();++i){
            prev[i]+=" "+word;
        }
        return prev;
    }

public:
    vector<string> wordBreak(string s, unordered_set<string>& dict) {
        if(m.count(s)) return m[s]; //take from memory
        vector<string> result;
        if(dict.count(s)){ //a whole string is a word
            result.push_back(s);
        }
        for(int i=1;i<s.size();++i){
            string word=s.substr(i);
            if(dict.count(word)){
                string rem=s.substr(0,i);
                vector<string> prev=combine(word,wordBreak(rem,dict));
                result.insert(result.end(),prev.begin(), prev.end());
            }
        }
        m[s]=result; //memorize
        return result;
    }
};
```

written by samoshka original link here

Linked List Cycle(141)

Answer 1

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
 /**
 use faster and lower runner solution. (2 pointers)
 the faster one move 2 steps, and slower one move only one step.
 if there's a circle, the faster one will finally "catch" the slower one.
 (the distance between these 2 pointers will decrease one every time.)

 if there's no circle, the faster runner will reach the end of linked list. (NULL
)
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head == NULL || head -> next == NULL)
            return false;

        ListNode *fast = head;
        ListNode *slow = head;

        while(fast -> next && fast -> next -> next){
            fast = fast -> next -> next;
            slow = slow -> next;
            if(fast == slow)
                return true;
        }

        return false;
    }
};
```

written by autekroy original link here

Answer 2

```
    public boolean hasCycle(ListNode head) {
        if(head==null) return false;
        ListNode walker = head;
        ListNode runner = head;
        while(runner.next!=null && runner.next.next!=null) {
            walker = walker.next;
            runner = runner.next.next;
            if(walker==runner) return true;
        }
        return false;
    }
```

1. Use two pointers, **walker** and **runner**.
2. **walker** moves step by step. **runner** moves two steps at time.
3. if the Linked List has a cycle**walker** and **runner** will meet at some point.

written by fabrizio3 original link here

Answer 3

I cannot give a solution to make it possible. I can only do it in O(1) space using the two runner solution, which I think is the best one.

```
// set two runners
ListNode slow = head;
ListNode fast = head;

// fast runner move 2 steps at one time while slow runner move 1 step,
// if traverse to a null, there must be no loop
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) {
        return true;
    }
}
return false;
```

written by Ethanlugc original link here

Linked List Cycle II(142)

Answer 1

my solution is like this: using two pointers, one of them one step at a time. another pointer each take two steps. Suppose the first meet at step ***k***,the length of the Cycle is ***r***. so..**2k-k=nr,k=nr** Now, the distance between the start node of list and the start node of cycle is ***s***. the distance between the start of list and the first meeting node is ***k***(the pointer which wake one step at a time waked k steps).the distance between the start node of cycle and the first meeting node is ***m***, so...**s=k-m, s=nr-m=(n-1)r+(r-m),here we takes n = 1**..so, using one pointer start from the start node of list, another pointer start from the first meeting node, all of them wake one step at a time, the first time they meeting each other is the start of the cycle.

```
    ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL) return NULL;

    ListNode* firstp = head;
    ListNode* secondp = head;
    bool isCycle = false;

    while(firstp != NULL && secondp != NULL) {
        firstp = firstp->next;
        if (secondp->next == NULL) return NULL;
        secondp = secondp->next->next;
        if (firstp == secondp) { isCycle = true; break; }
    }

    if(!isCycle) return NULL;
    firstp = head;
    while( firstp != secondp) {
        firstp = firstp->next;
        secondp = secondp->next;
    }

    return firstp;
}
```

written by wallop original link here

Answer 2

Alogrithm Description:

Step 1: Determine whether there is a cycle

1.1) Using a slow pointer that move forward 1 step each time

1.2) Using a fast pointer that move forward 2 steps each time

1.3) If the slow pointer and fast pointer both point to the same location after several moving steps, there is a cycle;

1.4) Otherwise, if (fast->next == NULL || fast->next->next == NULL), there has no

cycle.

Step 2: If there is a cycle, return the entry location of the cycle

2.1) L1 is defined as the distance between the head point and entry point

2.2) L2 is defined as the distance between the entry point and the meeting point

2.3) C is defined as the length of the cycle

2.4) n is defined as the travel times of the fast pointer around the cycle When the first encounter of the slow pointer and the fast pointer

According to the definition of L1, L2 and C, we can obtain:

- the total distance of the slow pointer traveled when encounter is L1 + L2
- the total distance of the fast pointer traveled when encounter is L1 + L2 + n * C
- Because the total distance the fast pointer traveled is twice as the slow pointer, Thus:
- 2 * (L1+L2) = L1 + L2 + n * C => L1 + L2 = n * C =>**L1 = (n - 1)* C + (C - L2)**

It can be concluded that the distance between the head location and entry location is equal to the distance between the meeting location and the entry location along the direction of forward movement.

So, when the slow pointer and the fast pointer encounter in the cycle, we can define a pointer "entry" that point to the head, this "entry" pointer moves one step each time so as the slow pointer. When this "entry" pointer and the slow pointer both point to the same location, this location is the node where the cycle begins.

==========

Here is the code:

```
ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL)
        return NULL;

    ListNode *slow  = head;
    ListNode *fast  = head;
    ListNode *entry = head;

    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {                     // there is a cycle
            while(slow != entry) {              // found the entry location
                slow  = slow->next;
                entry = entry->next;
            }
            return entry;
        }
    }
    return NULL;                                // there has no cycle
}
```

written by ngcl original link here

Answer 3

Define two pointers slow and fast. Both start at head node, fast is twice as fast as slow. If it reaches the end it means there is no cycle, otherwise eventually it will eventually catch up to slow pointer somewhere in the cycle.

Let the distance from the first node to the the node where cycle begins be A, and let say the slow pointer travels travels A+B. The fast pointer must travel 2A+2B to catch up. The cycle size is N. Full cycle is also how much more fast pointer has traveled than slow pointer at meeting point.

```
A+B+N = 2A+2B
N=A+B
```

From our calculation slow pointer traveled exactly full cycle when it meets fast pointer, and since originally it travled A before starting on a cycle, it must travel A to reach the point where cycle begins! We can start another slow pointer at head node, and move both pointers until they meet at the beginning of a cycle.

```
public class Solution {
            public ListNode detectCycle(ListNode head) {
                ListNode slow = head;
                ListNode fast = head;

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

                    if (fast == slow){
                        ListNode slow2 = head;
                        while (slow2 != slow){
                            slow = slow.next;
                            slow2 = slow2.next;
                        }
                        return slow;
                    }
                }
                return null;
            }
        }
```

written by qgambit2 original link here

Reorder List(143)

Answer 1

This question is a combination of **Reverse a linked list I & II** . It should be pretty straight forward to do it in 3 steps :)

```
public void reorderList(ListNode head) {
            if(head==null||head.next==null) return;

            //Find the middle of the list
            ListNode p1=head;
            ListNode p2=head;
            while(p2.next!=null&&p2.next.next!=null){
                p1=p1.next;
                p2=p2.next.next;
            }

            //Reverse the half after middle  1->2->3->4->5->6 to 1->2->3->6->5->
4
            ListNode preMiddle=p1;
            ListNode preCurrent=p1.next;
            while(preCurrent.next!=null){
                ListNode current=preCurrent.next;
                preCurrent.next=current.next;
                current.next=preMiddle.next;
                preMiddle.next=current;
            }

            //Start reorder one by one  1->2->3->6->5->4 to 1->6->2->5->3->4
            p1=head;
            p2=preMiddle.next;
            while(p1!=preMiddle){
                preMiddle.next=p2.next;
                p2.next=p1.next;
                p1.next=p2;
                p1=p2.next;
                p2=preMiddle.next;
            }
        }
```

written by wanqing original link here

Answer 2

```
// O(N) time, O(1) space in total
void reorderList(ListNode *head) {
    if (!head || !head->next) return;

    // find the middle node: O(n)
    ListNode *p1 = head, *p2 = head->next;
    while (p2 && p2->next) {
        p1 = p1->next;
        p2 = p2->next->next;
    }

    // cut from the middle and reverse the second half: O(n)
    ListNode *head2 = p1->next;
    p1->next = NULL;

    p2 = head2->next;
    head2->next = NULL;
    while (p2) {
        p1 = p2->next;
        p2->next = head2;
        head2 = p2;
        p2 = p1;
    }

    // merge two lists: O(n)
    for (p1 = head, p2 = head2; p1; ) {
        auto t = p1->next;
        p1 = p1->next = p2;
        p2 = t;
    }

    //for (p1 = head, p2 = head2; p2; ) {
    //    auto t = p1->next;
    //    p1->next = p2;
    //    p2 = p2->next;
    //    p1 = p1->next->next = t;
    //}
}
```

written by shichaotan original link here

Answer 3

```
# Splits in place a list in two halves, the first half is >= in size than the seco
nd.
# @return A tuple containing the heads of the two halves
def _splitList(head):
    fast = head
    slow = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next
        fast = fast.next

    middle = slow.next
```

```
    slow.next = None

    return head, middle

# Reverses in place a list.
# @return Returns the head of the new reversed list
def _reverseList(head):

  last = None
  currentNode = head

  while currentNode:
    nextNode = currentNode.next
    currentNode.next = last
    last = currentNode
    currentNode = nextNode

  return last

# Merges in place two lists
# @return The newly merged list.
def _mergeLists(a, b):

    tail = a
    head = a

    a = a.next
    while b:
        tail.next = b
        tail = tail.next
        b = b.next
        if a:
            a, b = b, a

    return head


class Solution:

    # @param head, a ListNode
    # @return nothing
    def reorderList(self, head):

        if not head or not head.next:
            return

        a, b = _splitList(head)
        b = _reverseList(b)
        head = _mergeLists(a, b)
```

written by riccardo original link here

Binary Tree Preorder Traversal(144)

Answer 1

Note that in this solution only right children are stored to stack.

```
public List<Integer> preorderTraversal(TreeNode node) {
    List<Integer> list = new LinkedList<Integer>();
    Stack<TreeNode> rights = new Stack<TreeNode>();
    while(node != null) {
        list.add(node.val);
        if (node.right != null) {
            rights.push(node.right);
        }
        node = node.left;
        if (node == null && !rights.isEmpty()) {
            node = rights.pop();
        }
    }
    return list;
}
```

written by pavel-shlyk original link here

Answer 2

1. Create an empty stack, Push root node to the stack.
2. Do following while stack is not empty.

    2.1. pop an item from the stack and print it.

    2.2. push the right child of popped item to stack.

    2.3. push the left child of popped item to stack.

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        stack<TreeNode*> nodeStack;
        vector<int> result;
        //base case
        if(root==NULL)
        return result;
        nodeStack.push(root);
        while(!nodeStack.empty())
        {
            TreeNode* node= nodeStack.top();
            result.push_back(node->val);
            nodeStack.pop();
            if(node->right)
            nodeStack.push(node->right);
            if(node->left)
            nodeStack.push(node->left);
        }
        return result;

    }
};
```

written by Deepalaxmi original link here

Answer 3

```
class Solution {
public:
vector<int> preorderTraversal(TreeNode *root) {
    if (root==NULL) {
        return vector<int>();
    }
    vector<int> result;
    stack<TreeNode *> treeStack;
    treeStack.push(root);
    while (!treeStack.empty()) {
        TreeNode *temp = treeStack.top();
        result.push_back(temp->val);
        treeStack.pop();
        if (temp->right!=NULL) {
            treeStack.push(temp->right);
        }
        if (temp->left!=NULL) {
            treeStack.push(temp->left);
        }
    }
    return result;
}
};
```

written by yulingtianxia original link here

Binary Tree Postorder Traversal(145)

Answer 1

pre-order traversal is **root-left-right**, and post order is **left-right-root**. modify the code for pre-order to make it root-right-left, and then **reverse** the output so that we can get left-right-root .

1. Create an empty stack, Push root node to the stack.
2. Do following while stack is not empty.

   2.1. pop an item from the stack and print it.

   2.2. push the left child of popped item to stack.

   2.3. push the right child of popped item to stack.

3. reverse the ouput.

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        stack<TreeNode*> nodeStack;
        vector<int> result;
        //base case
        if(root==NULL)
        return result;
        nodeStack.push(root);
    while(!nodeStack.empty())
    {
        TreeNode* node= nodeStack.top();
        result.push_back(node->val);
        nodeStack.pop();
        if(node->left)
        nodeStack.push(node->left);
        if(node->right)
        nodeStack.push(node->right);
    }
     reverse(result.begin(),result.end());
     return result;

}
```

};

written by Deepalaxmi original link here

Answer 2

i have saw lots of post in this discussion, but most of them are not concise, just share mine for your reference, writing a concise code is very important

```
vector<int> postorderTraversal(TreeNode *root) {
    vector<int> v;
    if (!root) return v;

    stack<TreeNode *> s;
    s.push(root);

    TreeNode *p = NULL;
    while(!s.empty()) {
        p = s.top();
        s.pop();
        v.insert(v.begin(), p->val);
        if (p->left) s.push(p->left);
        if (p->right) s.push(p->right);
    }

    return v;
}
```

written by shichaotan original link here

Answer 3

Hi, this is a fundamental and yet classic problem. I share my three solutions here:

1. Iterative solution using stack --- `O(n)` time and `O(n)` space;
2. Recursive solution --- `O(n)` time and `O(n)` space (considering the spaces of function call stack);
3. **Morris traversal --- `O(n)` time and `O(1)` space!!!**

Iterative solution using stack:

```
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> nodes;
    stack<TreeNode*> toVisit;
    TreeNode* curNode = root;
    TreeNode* lastNode = NULL;
    while (curNode || !toVisit.empty()) {
        if (curNode) {
            toVisit.push(curNode);
            curNode = curNode -> left;
        }
        else {
            TreeNode* topNode = toVisit.top();
            if (topNode -> right && lastNode != topNode -> right)
                curNode = topNode -> right;
            else {
                nodes.push_back(topNode -> val);
                lastNode = topNode;
                toVisit.pop();
            }
        }
    }
    return nodes;
}
```

Recursive solution:

```
void postorder(TreeNode* root, vector<int>& nodes) {
    if (!root) return;
    postorder(root -> left, nodes);
    postorder(root -> right, nodes);
    nodes.push_back(root -> val);
}
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> nodes;
    postorder(root, nodes);
    return nodes;
}
```

Morris traversal:

```
void reverseNodes(TreeNode* start, TreeNode* end) {
    if (start == end) return;
    TreeNode* x = start;
    TreeNode* y = start -> right;
    TreeNode* z;
    while (x != end) {
        z = y -> right;
        y -> right = x;
        x = y;
        y = z;
    }
}
void reverseAddNodes(TreeNode* start, TreeNode* end, vector<int>& nodes) {
    reverseNodes(start, end);
    TreeNode* node = end;
    while (true) {
        nodes.push_back(node -> val);
        if (node == start) break;
        node = node -> right;
    }
    reverseNodes(end, start);
}
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> nodes;
    TreeNode* dump = new TreeNode(0);
    dump -> left = root;
    TreeNode* curNode = dump;
    while (curNode) {
        if (curNode -> left) {
            TreeNode* predecessor = curNode -> left;
            while (predecessor -> right && predecessor -> right != curNode)
                predecessor = predecessor -> right;
            if (!(predecessor -> right)) {
                predecessor -> right = curNode;
                curNode = curNode -> left;
            }
            else {
                reverseAddNodes(curNode -> left, predecessor, nodes);
                predecessor -> right = NULL;
                curNode = curNode -> right;
            }
        }
        else curNode = curNode -> right;
    }
    return nodes;
}
```

written by jianchao.li.fighter original link here

LRU Cache(146)

Answer 1

The problem can be solved with a hashtable that keeps track of the keys and its values in the double linked list. One interesting property about double linked list is that the node can remove itself without other reference. In addition, it takes constant time to add and remove nodes from the head or tail.

One particularity about the double linked list that I implemented is that I create a pseudo head and tail to mark the boundary, so that we don't need to check the NULL node during the update. This makes the code more concise and clean, and also it is good for the performance as well.

Voila, here is the code.

```
class DLinkedNode {
    int key;
    int value;
    DLinkedNode pre;
    DLinkedNode post;
}

/**
 * Always add the new node right after head;
 */
private void addNode(DLinkedNode node){
    node.pre = head;
    node.post = head.post;

    head.post.pre = node;
    head.post = node;
}

/**
 * Remove an existing node from the linked list.
 */
private void removeNode(DLinkedNode node){
    DLinkedNode pre = node.pre;
    DLinkedNode post = node.post;

    pre.post = post;
    post.pre = pre;
}

/**
 * Move certain node in between to the head.
 */
private void moveToHead(DLinkedNode node){
    this.removeNode(node);
    this.addNode(node);
}

// pop the current tail.
private DLinkedNode popTail(){
```

```
    DLinkedNode res = tail.pre;
    this.removeNode(res);
    return res;
}

private Hashtable<Integer, DLinkedNode>
    cache = new Hashtable<Integer, DLinkedNode>();
private int count;
private int capacity;
private DLinkedNode head, tail;

public LRUCache(int capacity) {
    this.count = 0;
    this.capacity = capacity;

    head = new DLinkedNode();
    head.pre = null;

    tail = new DLinkedNode();
    tail.post = null;

    head.post = tail;
    tail.pre = head;
}

public int get(int key) {

    DLinkedNode node = cache.get(key);
    if(node == null){
        return -1; // should raise exception here.
    }

    // move the accessed node to the head;
    this.moveToHead(node);

    return node.value;
}


public void set(int key, int value) {
    DLinkedNode node = cache.get(key);

    if(node == null){

        DLinkedNode newNode = new DLinkedNode();
        newNode.key = key;
        newNode.value = value;

        this.cache.put(key, newNode);
        this.addNode(newNode);

        ++count;

        if(count > capacity){
            // pop the tail
            DLinkedNode tail = this.popTail();
            this.cache.remove(tail.key);
```

```
            --count;
        }
    }else{
        // update the value.
        node.value = value;
        this.moveToHead(node);
    }

}
```

written by liaison original link here

Answer 2

There is a similar example in Java, but I wanted to share my solution using the new C++11 unordered_map and a list. The good thing about lists is that iterators are never invalidated by modifiers (unless erasing the element itself). This way, we can store the iterator to the corresponding LRU queue in the values of the hash map. Since using erase on a list with an iterator takes constant time, all operations of the LRU cache run in constant time.

```
class LRUCache {
public:
    LRUCache(int capacity) : _capacity(capacity) {}

    int get(int key) {
        auto it = cache.find(key);
        if (it == cache.end()) return -1;
        touch(it);
        return it->second.first;
    }

    void set(int key, int value) {
        auto it = cache.find(key);
        if (it != cache.end()) touch(it);
        else {
            if (cache.size() == _capacity) {
                cache.erase(used.back());
                used.pop_back();
            }
            used.push_front(key);
        }
        cache[key] = { value, used.begin() };
    }

private:
    typedef list<int> LI;
    typedef pair<int, LI::iterator> PII;
    typedef unordered_map<int, PII> HIPII;

    void touch(HIPII::iterator it) {
        int key = it->first;
        used.erase(it->second.second);
        used.push_front(key);
        it->second.second = used.begin();
    }

    HIPII cache;
    LI used;
    int _capacity;
};
```

written by afernandez90 original link here

Answer 3

Solution is unusual - combination of 2 data structures - hash map and linked list.
Algorithm:

1. hash map holds iterators to linked list
2. linked list holds key and value, key to access hash map items
3. when item is accessed, it's promoted - moved to the tail of the list - O(1) operation
4. when item should be removed, we remove head of the list - O(1) operation
5. when item is not promoted long time, it's moved to the head of the list

automatically

6. get() - O(1) performance, set() - O(1) performance

{

```
class LRUCache{
private:
    struct item_t{
        int key, val;
        item_t(int k, int v) :key(k), val(v){}
    };
    typedef list<item_t> list_t;
    typedef unordered_map<int, list_t::iterator> map_t;

    map_t   m_map;
    list_t  m_list;
    int     m_capacity;
public:
    LRUCache(int capacity) : m_capacity(capacity) {
    }
    int get(int key) {
        map_t::iterator i = m_map.find(key);
        if (i == m_map.end()) return -1;
        m_map[key] = promote(i->second);
        return m_map[key]->val;
    }
    void set(int key, int value) {
        map_t::iterator i = m_map.find(key);
        if (i != m_map.end()){
            m_map[key] = promote(i->second);
            m_map[key]->val = value;
        }
        else {
            if (m_map.size() < m_capacity){
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value))
;
            }
            else {
                m_map.erase(m_list.front().key);
                m_list.pop_front();
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value))
;
            }
        }
    }
    list_t::iterator promote(list_t::iterator i){
        list_t::iterator inew = m_list.insert(m_list.end(), *i);
        m_list.erase(i);
        return inew;
    }
};
```

}

btw LeetCode, it was really hard to insert this code, after pressing {} button, class

was improperly formatted. I inserted additional braces around class.

written by yakov.sum original link here

Insertion Sort List(147)

Answer 1

```
public ListNode insertionSortList(ListNode head) {
        if( head == null ){
            return head;
        }

        ListNode helper = new ListNode(0); //new starter of the sorted list
        ListNode cur = head; //the node will be inserted
        ListNode pre = helper; //insert node between pre and pre.next
        ListNode next = null; //the next node will be inserted
        //not the end of input list
        while( cur != null ){
            next = cur.next;
            //find the right place to insert
            while( pre.next != null && pre.next.val < cur.val ){
                pre = pre.next;
            }
            //insert between pre and pre.next
            cur.next = pre.next;
            pre.next = cur;
            pre = helper;
            cur = next;
        }

        return helper.next;
    }
```

written by sbvictory original link here

Answer 2

```
public class Solution {
public ListNode insertionSortList(ListNode head) {
    ListNode helper=new ListNode(0);
    ListNode pre=helper;
    ListNode current=head;
    while(current!=null) {
        pre=helper;
        while(pre.next!=null&&pre.next.val<current.val) {
            pre=pre.next;
        }
        ListNode next=current.next;
        current.next=pre.next;
        pre.next=current;
        current=next;
    }
    return helper.next;
}
```

}

written by isly original link here

Answer 3

Well, life gets difficult pretty soon whenever the same operation on array is transferred to linked list.

First, a quick recap of insertion sort:

Start from the second element (simply `a[1]` in array and the annoying `head -> next -> val` in linked list), each time when we see a node with `val` smaller than its previous node, we scan from the `head` and find the position that the current node should be inserted. Since a node may be inserted before `head`, we create a `new_head` that points to `head`. The insertion operation, however, is a little easier for linked list.

Now comes the code:

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode* new_head = new ListNode(0);
        new_head -> next = head;
        ListNode* pre = new_head;
        ListNode* cur = head;
        while (cur) {
            if (cur -> next && cur -> next -> val < cur -> val) {
                while (pre -> next && pre -> next -> val < cur -> next -> val)
                    pre = pre -> next;
                /* Insert cur -> next after pre.*/
                ListNode* temp = pre -> next;
                pre -> next = cur -> next;
                cur -> next = cur -> next -> next;
                pre -> next -> next = temp;
                /* Move pre back to new_head. */
                pre = new_head;
            }
            else cur = cur -> next;
        }
        ListNode* res = new_head -> next;
        delete new_head;
        return res;
    }
};
```

written by jianchao.li.fighter original link here

Sort List(148)

Answer 1

```java
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode f = head.next.next;
        ListNode p = head;
        while (f != null && f.next != null) {
            p = p.next;
            f =  f.next.next;
        }
        ListNode h2 = sortList(p.next);
        p.next = null;
        return merge(sortList(head), h2);
    }
    public ListNode merge(ListNode h1, ListNode h2) {
        ListNode hn = new ListNode(Integer.MIN_VALUE);
        ListNode c = hn;
        while (h1 != null && h2 != null) {
            if (h1.val < h2.val) {
                c.next = h1;
                h1 = h1.next;
            }
            else {
                c.next = h2;
                h2 = h2.next;
            }
            c = c.next;
        }
        if (h1 != null)
            c.next = h1;
        if (h2 != null)
            c.next = h2;
        return hn.next;
    }
}
```

written by potpie original link here

Answer 2

Nice problem. I use a non-recurisve way to write merge sort. For example, the size of ListNode is 8,

Round #1 block_size = 1

(a1, a2), (a3, a4), (a5, a6), (a7, a8)

Compare a1 with a2, a3 with a4 ...

Round #2 block_size = 2

(a1, a2, a3, a4), (a5, a6, a7, a8)

merge two sorted arrays (a1, a2) and (a3, a4), then merge tow sorted arrays(a5, a6) and (a7, a8)

Round #3 block_size = 4

(a1, a2, a3, a4, a5, a6, a7, a8)

merge two sorted arrays (a1, a2, a3, a4), and (a5, a6, a7, a8)

No need for round #4 cause block_size = 8 >= n = 8

```cpp
class Solution {
public:
    int count_size(ListNode *node){
        int n = 0;
        while (node != NULL){
            node = node->next;
            ++n;
        }
        return n;
    }
    ListNode *sortList(ListNode *head) {
        int block_size = 1, n = count_size(head), iter = 0, i = 0, a = 0, b = 0;
        ListNode virtual_head(0);
        ListNode *last = NULL, *it = NULL, *A = NULL, *B = NULL, *tmp = NULL;
        virtual_head.next = head;
        while (block_size < n){
            iter = 0;
            last = &virtual_head;
            it = virtual_head.next;
            while (iter <  n){
                a = min(n - iter, block_size);
                b = min(n - iter - a, block_size);

                A = it;
                if (b != 0){
                    for (i = 0; i < a - 1; ++i) it = it->next;
                    B = it->next;
                    it->next = NULL;
                    it = B;

                    for (i = 0; i < b - 1; ++i) it = it->next;
                    tmp = it->next;
                    it->next = NULL;
                    it = tmp;
                }

                while (A || B){
                    if (B == NULL || (A != NULL && A->val <= B->val)){
                        last->next = A;
                        last = last->next;
                        A = A->next;
                    } else {
                        last->next = B;
                        last = last->next;
                        B = B->next;
                    }
                }
                last->next = NULL;
                iter += a + b;
            }
            block_size <<= 1;
        }
        return virtual_head.next;
    }
};
```

written by vaputa original link here

Answer 3

```
public class Solution {

  public ListNode sortList(ListNode head) {
    if (head == null || head.next == null)
      return head;

    // step 1. cut the list to two halves
    ListNode prev = null, slow = head, fast = head;

    while (fast != null && fast.next != null) {
      prev = slow;
      slow = slow.next;
      fast = fast.next.next;
    }

    prev.next = null;

    // step 2. sort each half
    ListNode l1 = sortList(head);
    ListNode l2 = sortList(slow);

    // step 3. merge l1 and l2
    return merge(l1, l2);
  }

  ListNode merge(ListNode l1, ListNode l2) {
    ListNode l = new ListNode(0), p = l;

    while (l1 != null && l2 != null) {
      if (l1.val < l2.val) {
        p.next = l1;
        l1 = l1.next;
      } else {
        p.next = l2;
        l2 = l2.next;
      }
      p = p.next;
    }

    if (l1 != null)
      p.next = l1;

    if (l2 != null)
      p.next = l2;

    return l.next;
  }

}
```

written by jeantimex original link here

Max Points on a Line(149)

Answer 1

```
  /*
     *  A line is determined by two factors,say y=ax+b
     *
     *  If two points(x1,y1) (x2,y2) are on the same line(Of course).

     *  Consider the gap between two points.

     *  We have (y2-y1)=a(x2-x1),a=(y2-y1)/(x2-x1) a is a rational, b is canceled
since b is a constant

     *  If a third point (x3,y3) are on the same line. So we must have y3=ax3+b

     *  Thus,(y3-y1)/(x3-x1)=(y2-y1)/(x2-x1)=a

     *  Since a is a rational, there exists y0 and x0, y0/x0=(y3-y1)/(x3-x1)=(y2-
y1)/(x2-x1)=a

     *  So we can use y0&x0 to track a line;
     */

    public class Solution{
        public int maxPoints(Point[] points) {
            if (points==null) return 0;
            if (points.length<=2) return points.length;

            Map<Integer,Map<Integer,Integer>> map = new HashMap<Integer,Map<Integ
er,Integer>>();
            int result=0;
            for (int i=0;i<points.length;i++){
                map.clear();
                int overlap=0,max=0;
                for (int j=i+1;j<points.length;j++){
                    int x=points[j].x-points[i].x;
                    int y=points[j].y-points[i].y;
                    if (x==0&&y==0){
                        overlap++;
                        continue;
                    }
                    int gcd=generateGCD(x,y);
                    if (gcd!=0){
                        x/=gcd;
                        y/=gcd;
                    }

                    if (map.containsKey(x)){
                        if (map.get(x).containsKey(y)){
                            map.get(x).put(y, map.get(x).get(y)+1);
                        }else{
                            map.get(x).put(y, 1);
                        }
                    }else{
                        Map<Integer,Integer> m = new HashMap<Integer,Integer>();
```

```
                    m.put(y, 1);
                    map.put(x, m);
                }
                max=Math.max(max, map.get(x).get(y));
            }
            result=Math.max(result, max+overlap+1);
        }
        return result;


    }
    private int generateGCD(int a,int b){

        if (b==0) return a;
        else return generateGCD(b,a%b);

    }
}
```

written by reeclapple original link here

Answer 2

Hint by @stellari

"For each point pi, calculate the slope of each line it forms with all other points with greater indices, i.e. pi+1, pi+2, ..., and use a map to record how many lines have the same slope (If two lines have the same slope and share a common point, then the two lines must be the same one). By doing so, you can easily find how many points are on the same line that ends at pi in O(n). Thus the amortized running time of the whole algorithm is O(n^2)."

In order to avoid using double type(the slope k) as map key, I used pair (int a, int b) as the key where a=pj.x-pi.x, b=pj.y-pi.y, and k=b/a. Using greatest common divider of a and b to divide both a, b ensures that lines with same slope have the same key.

I also handled two special cases: (1) when two points are on a vertical line (2) when two points are the same.

```
class Solution {
public:
    int maxPoints(vector<Point> &points) {

        if(points.size()<2) return points.size();

        int result=0;

        for(int i=0; i<points.size(); i++) {

            map<pair<int, int>, int> lines;
            int localmax=0, overlap=0, vertical=0;

            for(int j=i+1; j<points.size(); j++) {

                if(points[j].x==points[i].x && points[j].y==points[i].y) {

                    overlap++;
                    continue;
                }
                else if(points[j].x==points[i].x) vertical++;
                else {

                    int a=points[j].x-points[i].x, b=points[j].y-points[i].y;
                    int gcd=GCD(a, b);

                    a/=gcd;
                    b/=gcd;

                    lines[make_pair(a, b)]++;
                    localmax=max(lines[make_pair(a, b)], localmax);
                }

                localmax=max(vertical, localmax);
            }

            result=max(result, localmax+overlap+1);
        }

        return result;
    }

private:
    int GCD(int a, int b) {

        if(b==0) return a;
        else return GCD(b, a%b);
    }
};
```

written by Yoursong original link here

Answer 3

```
int maxPoints(vector<Point> &points) {
    int result = 0;
    for(int i = 0; i < points.size(); i++){
        int samePoint = 1;
        unordered_map<double, int> map;
        for(int j = i + 1; j < points.size(); j++){
            if(points[i].x == points[j].x && points[i].y == points[j].y){
                samePoint++;
            }
            else if(points[i].x == points[j].x){
                map[INT_MAX]++;
            }
            else{
                double slope = double(points[i].y - points[j].y) / double(points[
i].x - points[j].x);
                map[slope]++;
            }
        }
        int localMax = 0;
        for(auto it = map.begin(); it != map.end(); it++){
            localMax = max(localMax, it->second);
        }
        localMax += samePoint;
        result = max(result, localMax);
    }
    return result;
}
```

First, let's talk about mathematics.

How to determine if three points are on the same line?

The answer is to see if slopes of arbitrary two pairs are the same.

Second, let's see what the minimum time complexity can be.

Definitely, O(n^2). It's because you have to calculate all slopes between any two points.

Then let's go back to the solution of this problem.

In order to make this discussion simpler, let's pick a random point A as an example.

Given point A, we need to calculate all slopes between A and other points. There will be three cases:

1. Some other point is the same as point A.
2. Some other point has the same x coordinate as point A, which will result to a positive infinite slope.
3. General case. We can calculate slope.

We can store all slopes in a hash table. And we find which slope shows up mostly. Then add the number of same points to it. Then we know the maximum number of points on the same line for point A.

We can do the same thing to point B, point C...

Finally, just return the maximum result among point A, point B, point C...

written by zxyperfect original link here

Evaluate Reverse Polish Notation(150)

Answer 1

when I test ["10","6","9","3","+","-11","","/","","17","+","5","+"], in this program, the result I got is 12, I think I am right. Because when I calculate 6/(-132) = -1, not 0, so i think the result is 12 not 22.

written by yfdyyy original link here

Answer 2

Hi everyone.
The Reverse Polish Notation is a stack of operations, thus, I decided to use java.util.Stack to solve this problem. As you can see, I add every token as an integer in the stack, unless it's an operation. In that case, I pop two elements from the stack and then save the result back to it. After all operations are done through, the remaining element in the stack will be the result.
Any comments or improvements are welcome.

Cheers.

```
import java.util.Stack;

public class Solution {
    public int evalRPN(String[] tokens) {
        int a,b;
        Stack<Integer> S = new Stack<Integer>();
        for (String s : tokens) {
            if(s.equals("+")) {
                S.add(S.pop()+S.pop());
            }
            else if(s.equals("/")) {
                b = S.pop();
                a = S.pop();
                S.add(a / b);
            }
            else if(s.equals("*")) {
                S.add(S.pop() * S.pop());
            }
            else if(s.equals("-")) {
                b = S.pop();
                a = S.pop();
                S.add(a - b);
            }
            else {
                S.add(Integer.parseInt(s));
            }
        }
        return S.pop();
    }
}
```

written by pvaldes original link here

Answer 3

```
public int evalRPN(String[] a) {
  Stack<Integer> stack = new Stack<Integer>();

  for (int i = 0; i < a.length; i++) {
    switch (a[i]) {
      case "+":
        stack.push(stack.pop() + stack.pop());
        break;

      case "-":
        stack.push(-stack.pop() + stack.pop());
        break;

      case "*":
        stack.push(stack.pop() * stack.pop());
        break;

      case "/":
        int n1 = stack.pop(), n2 = stack.pop();
        stack.push(n2 / n1);
        break;

      default:
        stack.push(Integer.parseInt(a[i]));
    }
  }

  return stack.pop();
}
```

written by jeantimex original link here

Reverse Words in a String(151)

Answer 1

```
class Solution {
public:
    void reverseWords(string &s) {
        string result;
        int pos = 0;
        for (int i = 0; i < s.size(); i ++){
            if (s[i] == ' '){
                if (i > pos )
                    result = s.substr(pos,i-pos)+ " " + result ;
                pos = i + 1;
            }
            else if (i == s.size()-1)
                result = s.substr(pos,s.size()-pos)+" "+result;
        }
        s = result.substr(0,result.size()-1) ;
    }
};
```

written by exodia original link here

Answer 2

First, reverse the whole string, then reverse each word.

```
void reverseWords(string &s) {
    reverse(s.begin(), s.end());
    int storeIndex = 0;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] != ' ') {
            if (storeIndex != 0) s[storeIndex++] = ' ';
            int j = i;
            while (j < s.size() && s[j] != ' ') { s[storeIndex++] = s[j++]; }
            reverse(s.begin() + storeIndex - (j - i), s.begin() + storeIndex);
            i = j;
        }
    }
    s.erase(s.begin() + storeIndex, s.end());
}
```

written by yuruofeifei original link here

Answer 3

The idea is to ignore the extra spaces, reverse words one by one and reverse the whole string in the end. I think for the interview it is good to show that substr or istringstream can be used too. The idea is taken from here

```
class Solution {
public:

    // function to reverse any part of string from i to j (just one word or entir
e string)
    void reverseword(string &s, int i, int j){
        while(i<j){
          char t=s[i];
          s[i++]=s[j];
          s[j--]=t;
        }
    }

    void reverseWords(string &s) {

        int i=0, j=0;
        int l=0;
        int len=s.length();
        int wordcount=0;

        while(true){
            while(i<len && s[i] == ' ') i++;  // skip spaces in front of the word
            if(i==len) break;
            if(wordcount) s[j++]=' ';
            l=j;
            while(i<len && s[i] != ' ') {s[j]=s[i]; j++; i++;}
            reverseword(s,l,j-1);                // reverse word in place
            wordcount++;

        }

        s.resize(j);                            // resize result string
        reverseword(s,0,j-1);                   // reverse whole string
    }
};
```

written by JackBauer original link here

Maximum Product Subarray(152)

Answer 1

```
public int maxProduct(int[] A) {
    if (A.length == 0) {
        return 0;
    }

    int maxherepre = A[0];
    int minherepre = A[0];
    int maxsofar = A[0];
    int maxhere, minhere;

    for (int i = 1; i < A.length; i++) {
        maxhere = Math.max(Math.max(maxherepre * A[i], minherepre * A[i]), A[i]);
        minhere = Math.min(Math.min(maxherepre * A[i], minherepre * A[i]), A[i]);
        maxsofar = Math.max(maxhere, maxsofar);
        maxherepre = maxhere;
        minherepre = minhere;
    }
    return maxsofar;
}
```

Note: There's no need to use O(n) space, as all that you need is a minhere and maxhere. (local max and local min), then you can get maxsofar (which is global max) from them.

There's a chapter in Programming Pearls 2 that discussed the MaxSubArray problem, the idea is similar.

written by rliu054 original link here

Answer 2

```
int maxProduct(int A[], int n) {
    // store the result that is the max we have found so far
    int r = A[0];

    // imax/imin stores the max/min product of
    // subarray that ends with the current number A[i]
    for (int i = 1, imax = r, imin = r; i < n; i++) {
        // multiplied by a negative makes big number smaller, small number bigger
        // so we redefine the extremums by swapping them
        if (A[i] < 0)
            swap(imax, imin);

        // max/min product for the current number is either the current number it
self
        // or the max/min by the previous number times the current one
        imax = max(A[i], imax * A[i]);
        imin = min(A[i], imin * A[i]);

        // the newly computed max value is a candidate for our global result
        r = max(r, imax);
    }
    return r;
}
```

written by mzchen original link here

Answer 3

```
class Solution {
 // author : s2003zy
 // weibo : http://weibo.com/574433433
 // blog : http://s2003zy.com
 // Time : O(n)
 // Space : O(1)
 public:
    int maxProduct(int A[], int n) {
        int frontProduct = 1;
        int backProduct = 1;
        int ans = INT_MIN;
        for (int i = 0; i < n; ++i) {
            frontProduct *= A[i];
            backProduct *= A[n - i - 1];
            ans = max(ans,max(frontProduct,backProduct));
            frontProduct = frontProduct == 0 ? 1 : frontProduct;
            backProduct = backProduct == 0 ? 1 : backProduct;
        }
        return ans;
    }
};
```

written by songzy982 original link here

Find Minimum in Rotated Sorted Array(153)

Answer 1

Classic binary search problem.

Looking at subarray with index [start,end]. We can find out that if the first member is less than the last member, there's no rotation in the array. So we could directly return the first element in this subarray.

If the first element is larger than the last one, then we compute the element in the middle, and compare it with the first element. If value of the element in the middle is larger than the first element, we know the rotation is at the second half of this array. Else, it is in the first half in the array.

Welcome to put your comments and suggestions.

```
int findMin(vector<int> &num) {
        int start=0,end=num.size()-1;

        while (start<end) {
            if (num[start]<num[end])
                return num[start];

            int mid = (start+end)/2;

            if (num[mid]>=num[start]) {
                start = mid+1;
            } else {
                end = mid;
            }
        }

        return num[start];
    }
```

Some corner cases will be discussed here

written by changhaz original link here

Answer 2

In this problem, we have only three cases.

Case 1. The leftmost value is less than the rightmost value in the list: This means that the list is not rotated. e.g> [1 2 3 4 5 6 7]

Case 2. The value in the middle of the list is greater than the leftmost and rightmost values in the list. e.g> [4 5 6 7 0 1 2 3]

Case 3. The value in the middle of the list is less than the leftmost and rightmost values in the list. e.g> [5 6 7 0 1 2 3 4]

As you see in the examples above, if we have case 1, we just return the leftmost value in the list. If we have case 2, we just move to the right side of the list. If we have case

3 we need to move to the left side of the list.

Following is the code that implements the concept described above.

```
int findMin(vector<int>& nums) {
    int left = 0,  right = nums.size() - 1;
    while(left < right) {
        if(nums[left] < nums[right])
            return nums[left];

        int mid = (left + right)/2;
        if(nums[mid] > nums[right])
            left = mid + 1;
        else
            right = mid;
    }

    return nums[left];
}
```

written by jaewoo original link here

Answer 3

Binary search: basically eliminate the impossible elements by half each time by exploiting the sorted property.

```
    int findMin(vector<int> &num) {
        int lo =0, hi = num.size()-1;
        while(lo<hi){
              int mid=(lo+hi)/2;
              if(num[mid]>num[hi]) lo=mid+1;
              else hi=mid;
        }
        return num[lo];
    }
```

written by lucastan original link here

Find Minimum in Rotated Sorted Array II(154)

Answer 1

```
class Solution {
public:
    int findMin(vector<int> &num) {
        int lo = 0;
        int hi = num.size() - 1;
        int mid = 0;

        while(lo < hi) {
            mid = lo + (hi - lo) / 2;

            if (num[mid] > num[hi]) {
                lo = mid + 1;
            }
            else if (num[mid] < num[hi]) {
                hi = mid;
            }
            else { // when num[mid] and num[hi] are same
                hi--;
            }
        }
        return num[lo];
    }
};
```

When num[mid] == num[hi], we couldn't sure the position of minimum in mid's left or right, so just let upper bound reduce one.

written by sheehan original link here

Answer 2

```
class Solution {
public:
    int findMin(vector<int> &num) {
        if(num.empty())
            return 0;
        int i=0,j=num.size()-1;
        while(i<j)
        {
            int mid=(i+j)/2;
            if(num[j]<num[mid]){
                i=mid+1;
            }
            else if(num[mid]<num[j]){
                j=mid;
            }
            else{//num[mid]==num[j]
                if(num[i]==num[mid]){//linear complexity
                    i++;
                    j--;
                }
                else
                    j=mid;
            }
        }
        return num[j];
    }
};
```

written by kmind original link here

Answer 3

```
public int findMin(int[] nums) {
     int l = 0, r = nums.length-1;
     while (l < r) {
         int mid = (l + r) / 2;
         if (nums[mid] < nums[r]) {
             r = mid;
         } else if (nums[mid] > nums[r]){
             l = mid + 1;
         } else {
             r--;  //nums[mid]=nums[r] no idea, but we can eliminate nums[r];
         }
     }
     return nums[l];
}
```

written by jinwu original link here

Min Stack(155)

Answer 1

The question is ask to construct One stack. So I am using one stack.

The idea is to store the gap between the min value and the current value;

The problem for my solution is the cast. I have no idea to avoid the cast. Since the possible gap between the current value and the min value could be Integer.MAX*VALUE-Integer.MIN*VALUE;

```
public class MinStack {
    long min;
    Stack<Long> stack;

    public MinStack(){
        stack=new Stack<>();
    }

    public void push(int x) {
        if (stack.isEmpty()){
            stack.push(0L);
            min=x;
        }else{
            stack.push(x-min);//Could be negative if min value needs to change
            if (x<min) min=x;
        }
    }

    public void pop() {
        if (stack.isEmpty()) return;

        long pop=stack.pop();

        if (pop<0)  min=min-pop;//If negative, increase the min value

    }

    public int top() {
        long top=stack.peek();
        if (top>0){
            return (int)(top+min);
        }else{
           return (int)(min);
        }
    }

    public int getMin() {
        return (int)min;
    }
}
```

written by reeclapple original link here

Answer 2

```
class MinStack {
    int min=Integer.MAX_VALUE;
    Stack<Integer> stack = new Stack<Integer>();
    public void push(int x) {
       // only push the old minimum value when the current
       // minimum value changes after pushing the new value x
        if(x <= min){
            stack.push(min);
            min=x;
        }
        stack.push(x);
    }

    public void pop() {
       // if pop operation could result in the changing of the current minimum va
lue,
       // pop twice and change the current minimum value to the last minimum valu
e.
        if(stack.peek()==min) {
            stack.pop();
            min=stack.peek();
            stack.pop();
        }else{
            stack.pop();
        }
        if(stack.empty()){
            min=Integer.MAX_VALUE;
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return min;
    }
}
```

written by sometimescrazy original link here

Answer 3

```
class MinStack {
private:
    stack<int> s1;
    stack<int> s2;
public:
    void push(int x) {
        s1.push(x);
        if (s2.empty() || x <= getMin())  s2.push(x);
    }
    void pop() {
        if (s1.top() == getMin())  s2.pop();
        s1.pop();
    }
    int top() {
        return s1.top();
    }
    int getMin() {
        return s2.top();
    }
};
```

written by zjchenRice original link here

Binary Tree Upside Down(156)

Answer 1

```
public class Solution {
    public TreeNode UpsideDownBinaryTree(TreeNode root) {
        TreeNode curr = root;
        TreeNode prev = null;
        TreeNode next = null;
        TreeNode temp = null;

        while (curr != null) {
            next = curr.left;
            curr.left = temp;
            temp = curr.right;
            curr.right = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
}

Just think about how you can save the tree information
you need before changing the tree structure.
```

written by Yida original link here

Answer 2

```
TreeNode* upsideDownBinaryTree(TreeNode* root) {
    if (!root || !root->left) return root;
    TreeNode* cur_left = root->left;
    TreeNode* cur_right = root->right;
    TreeNode* new_root = upsideDownBinaryTree(root->left);
    cur_left->right = root;
    cur_left->left = cur_right;
    root->left = nullptr;
    root->right = nullptr;
    return new_root;
}
```

written by lchen77 original link here

Answer 3

```
public TreeNode upsideDownBinaryTree(TreeNode root) {
  if (root == null || root.left == null && root.right == null)
    return root;

  TreeNode newRoot = upsideDownBinaryTree(root.left);

  root.left.left = root.right;
  root.left.right = root;

  root.left = null;
  root.right = null;

  return newRoot;
}
```

written by jeantimex original link here

Read N Characters Given Read4(157)

Answer 1

This question is very unclear to me. what are we supposed to accomplish in this problem. Our read function returns an int but the expected result looks like its a String. if we are forced to read 4 chars at a time how do we ever read n chars that are not a factor of 4 if we actually use the read4 method? is 'n' the max we can return or must we return exactly n assuming at least n items exist other wise return the number of items. clearly I'm missing a lot here can someone please explain this problem to me.

written by mlblount45 original link here

Answer 2

```
/* The read4 API is defined in the parent class Reader4.
      int read4(char[] buf); */

public class Solution extends Reader4 {
    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
     * @return    The number of characters read
     */
    public int read(char[] buf, int n) {

        char[] buffer = new char[4];
        boolean endOfFile = false;
        int readBytes = 0;

        while (readBytes < n && !endOfFile) {
            int currReadBytes = read4(buffer);
            if (currReadBytes !=4) {
                endOfFile = true;
            }
            int length = Math.min(n - readBytes, currReadBytes);
            for (int i=0; i<length; i++) {
                buf[readBytes + i] = buffer[i];
            }
            readBytes += length;
        }
        return readBytes;
    }
}
```

personally, I feel this problem is hard to understand. I would prefer the return result is the buf instead of an int. copy the buf is error prone as it is a fixed array size. lots of assumption.

written by richilee original link here

Answer 3

```java
public int read(char[] buf, int n) {
  boolean eof = false;      // end of file flag
  int total = 0;            // total bytes have read
  char[] tmp = new char[4]; // temp buffer

  while (!eof && total < n) {
    int count = read4(tmp);

    // check if it's the end of the file
    eof = count < 4;

    // get the actual count
    count = Math.min(count, n - total);

    // copy from temp buffer to buf
    for (int i = 0; i < count; i++)
      buf[total++] = tmp[i];
  }

  return total;
}
```

written by jeantimex original link here

Read N Characters Given Read4 II - Call multiple times(158)

Answer 1

```
    private int buffPtr = 0;
    private int buffCnt = 0;
    private char[] buff = new char[4];
    public int read(char[] buf, int n) {
        int ptr = 0;
        while (ptr < n) {
            if (buffPtr == 0) {
                buffCnt = read4(buff);
            }
            if (buffCnt == 0) break;
            while (ptr < n && buffPtr < buffCnt) {
                buf[ptr++] = buff[buffPtr++];
            }
            if (buffPtr >= buffCnt) buffPtr = 0;
        }
        return ptr;
    }
```

I used buffer pointer (buffPtr) and buffer Counter (buffCnt) to store the data received in previous calls. In the while loop, if buffPtr reaches current buffCnt, it will be set as zero to be ready to read new data.

written by totalheap original link here

Answer 2

```
class Solution:
# @param buf, Destination buffer (a list of characters)
# @param n,   Maximum number of characters to read (an integer)
# @return     The number of characters read (an integer)
def __init__(self):
    self.queue = []

def read(self, buf, n):
    idx = 0
    while True:
        buf4 = [""]*4
        l = read4(buf4)
        self.queue.extend(buf4)
        curr = min(len(self.queue), n-idx)
        for i in xrange(curr):
            buf[idx] = self.queue.pop(0)
            idx+=1
        if curr == 0:
            break
    return idx
```

written by ycsung original link here

Answer 3

The key is to store memorized variable in the class level and remember offset position and remaining number of elements.

```java
/* The read4 API is defined in the parent class Reader4.
      int read4(char[] buf); */

public class Solution extends Reader4 {

    private int offSet = 0;
    private int remaining = 0;
    private boolean isEndOfFile = false;
    private char[] buffer = new char[4];

    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
     * @return    The number of characters read
     */
    public int read(char[] buf, int n) {
        int readBytes = 0;
        while (readBytes < n && (remaining != 0 || !isEndOfFile)) {
            int readSize = 0;
            if (remaining != 0) {
                readSize = remaining;
            } else {
                offSet = 0;
                readSize = read4(buffer);
                if (readSize != 4) {
                    isEndOfFile = true;
                }
            }
            int length = Math.min(n - readBytes, readSize);
            for (int i= offSet; i<offSet + length; i++) {
                buf[readBytes++] = buffer[i];
            }
            remaining = readSize - length;
            if (remaining != 0) {
                offSet += length;
            }
        }
        return readBytes;
    }
}
```

written by richilee original link here

Longest Substring with At Most Two Distinct Characters(159)

Answer 1

This question belong to the same category as those such as "longest substring without repeating characters", "minimum window substring", and "substring with concatenation of all words". To solve this kind of question we can use two pointers and a hash table. When the key of the hash table is char, we can simply use an array as the hash table. The most important idea in solving this kind of questions is "how to update the "start" pointer" and the solution to these questions seem usually differ only in this respect.

```
int lengthOfLongestSubstringTwoDistinct(string s) {
    if(s.empty()) return 0;

    int dict[256];
    fill_n(dict,256,0);
    int start = 0, len = 1, count = 0;
    for(int i=0; i<s.length(); i++) {
        dict[s[i]]++;
        if(dict[s[i]] == 1) { // new char
            count++;
            while(count > 2) {
                dict[s[start]]--;
                if(dict[s[start]] == 0) count--;
                start++;
            }
        }
        if(i-start+1 > len) len = i-start+1;
    }
    return len;
}
```

written by morrischen2008 original link here

Answer 2

I submitted this solution two months ago and I can't even remember whether I wrote myself or copied from others. I don't understand it now but find it AC and much shorter than other posts here.

```
int lengthOfLongestSubstringTwoDistinct(string s) {
        int i = 0, j = -1;
        int maxLen = 0;
        for (int k = 1; k < s.size(); k++) {
            if (s[k] == s[k-1]) continue;
            if (j > -1 && s[k] != s[j]) {
                maxLen = max(maxLen, k - i);
                i = j + 1;
            }
            j = k - 1;
        }
        return maxLen > (s.size() - i) ? maxLen : s.size() - i;
    }
```

written by emersonxsu original link here

Answer 3

The main idea is to maintain a sliding window with 2 unique characters. The key is to store the last occurrence of each character as the value in the hashmap. This way, whenever the size of the hashmap exceeds 2, we can traverse through the map to find the character with the left most index, and remove 1 character from our map. Since the range of characters is constrained, we should be able to find the left most index in constant time.

```
public class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        if(s.length() < 1) return 0;
        HashMap<Character,Integer> index = new HashMap<Character,Integer>();
        int lo = 0;
        int hi = 0;
        int maxLength = 0;
        while(hi < s.length()) {
            if(index.size() <= 2) {
                char c = s.charAt(hi);
                index.put(c, hi);
                hi++;
            }
            if(index.size() > 2) {
                int leftMost = s.length();
                for(int i : index.values()) {
                    leftMost = Math.min(leftMost,i);
                }
                char c = s.charAt(leftMost);
                index.remove(c);
                lo = leftMost+1;
            }
            maxLength = Math.max(maxLength, hi-lo);
        }
        return maxLength;
    }
}
```

written by kevinhsu original link here

Intersection of Two Linked Lists(160)

Answer 1

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
{
    ListNode *p1 = headA;
    ListNode *p2 = headB;

    if (p1 == NULL || p2 == NULL) return NULL;

    while (p1 != NULL && p2 != NULL && p1 != p2) {
        p1 = p1->next;
        p2 = p2->next;

        //
        // Any time they collide or reach end together without colliding
        // then return any one of the pointers.
        //
        if (p1 == p2) return p1;

        //
        // If one of them reaches the end earlier then reuse it
        // by moving it to the beginning of other list.
        // Once both of them go through reassigning,
        // they will be equidistant from the collision point.
        //
        if (p1 == NULL) p1 = headB;
        if (p2 == NULL) p2 = headA;
    }

    return p1;
}
```

written by satyakam original link here

Answer 2

1, Get the length of the two lists.

2, Align them to the same start point.

3, Move them together until finding the intersection point, or the end null

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    int lenA = length(headA), lenB = length(headB);
    // move headA and headB to the same start point
    while (lenA > lenB) {
        headA = headA.next;
        lenA--;
    }
    while (lenA < lenB) {
        headB = headB.next;
        lenB--;
    }
    // find the intersection until end
    while (headA != headB) {
        headA = headA.next;
        headB = headB.next;
    }
    return headA;
}

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

written by zkfairytale original link here

Answer 3

1. Scan both lists
2. For each list once it reaches the end, continue scanning the other list
3. Once the two runner equal to each other, return the position

Time O(n+m), space O(1)

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if( null==headA || null==headB )
            return null;

        ListNode curA = headA, curB = headB;
        while( curA!=curB){
            curA = curA==null?headB:curA.next;
            curB = curB==null?headA:curB.next;
        }
        return curA;
    }
```

written by askyfeng7 original link here

One Edit Distance(161)

Answer 1

Java:

```
for (int i = 0; i < Math.min(s.length(), t.length()); i++) {
    if (s.charAt(i) != t.charAt(i)) {
        return s.substring(i + (s.length() >= t.length() ? 1 : 0)).equals(t.subst
ring(i + (s.length() <= t.length() ? 1 : 0)));
    }
}
return Math.abs(s.length() - t.length()) == 1;
```

C++:

```
    for (int i = 0; i < min(s.size(), t.size()); i++) {
        if (s.at(i) != t.at(i)) {
            return s.substr(i + (s.size() >= t.size() ? 1 : 0)).compare(t.substr(
i + (s.size() <= t.size() ? 1 : 0))) == 0;
        }
    }
    return s.size() - t.size() == 1 || s.size() - t.size() == -1;
```

Python:

```
    for i in range(min(len(s), len(t))):
        if s[i] != t[i]:
            return s[i + (1 if len(s) >= len(t) else 0):] == t[i + (1 if len(s) <
= len(t) else 0):]
    return abs(len(s) - len(t)) == 1
```

written by xcv58 original link here

Answer 2

```java
public boolean isOneEditDistance(String s, String t) {
    if(Math.abs(s.length()-t.length()) > 1) return false;
    if(s.length() == t.length()) return isOneModify(s,t);
    if(s.length() > t.length()) return isOneDel(s,t);
    return isOneDel(t,s);
}
public boolean isOneDel(String s,String t){
    for(int i=0,j=0;i<s.length() && j<t.length();i++,j++){
        if(s.charAt(i) != t.charAt(j)){
            return s.substring(i+1).equals(t.substring(j));
        }
    }
    return true;
}
public boolean isOneModify(String s,String t){
    int diff =0;
    for(int i=0;i<s.length();i++){
        if(s.charAt(i) != t.charAt(i)) diff++;
    }
    return diff==1;
}
```

written by zq670067 original link here

Answer 3

To solve this problem, you first need to know what is*edit distance*. You may refer to this wikipedia article for more information.

For this problem, it implicitly assumes to use the classic**Levenshtein distance**, which involves **insertion**, **deletion** and **substitution** operations and all of them are of the same cost. Thus, if `S` is one edit distance apart from `T`, `T` is automatically one edit distance apart from `S`.

Now let's think about all the possible cases for two strings to be one edit distance apart. Well, that means, we can transform `S` to `T` by using exactly one edit operation. There are three possible cases:

1. We insert a character into `S` to get `T`.
2. We delete a character from `S` to get `T`.
3. We substitute a character of `S` to get `T`.

For cases 1 and 2, `S` and `T` will be one apart in their lengths. For cases 3, they are of the same length.

It is relatively easy to handle case 3. We simply traverse both of them and compare the characters at the corresponding positions. If we find exactly one mismatch during the traverse, they are one edit distance apart.

Now let's move on to cases 1 and 2. In fact, they can be merged into one case, that is, to delete a character from the longer string to get the shorter one, or equivalently, to insert a character into the shorter string to get the longer one.

We will handle cases 1 and 2 using the shorter string as the reference. We traverse the two strings, once we find a mismatch. We know this position is where the deletion in the longer string happens. For example, suppose `S = "kitten"` and `T = "kiten"`, we meet the first mismatch in the `4`-th position (`1`-based), which corresponds to the deleted character below, shown in between `*`. We then continue to compare the remaining sub-string of `T` (`en`) with the remaining sub-string of `S` (`en`) and find them to be the same. So they are one edit distance apart.

`S: k i t t e n`

`T: k i t *t* e n`

In fact, cases 1, 2 and 3 can be further handled using the same piece of code. For strings of the same length, once we find a mismatch, we just substitute one to be another and check whether they are now the same. For strings of one apart in lengths, we insert the deleted character of the longer string into the shorter one and compare whether they are the same.

The code is as follows. If you find the first half of the return statement (`!mismatch && (n - m == 1)`) hard to understand, run the code on cases that the mismatch only occurs at the last character of the longer string, like `S = "ab"` and `T = "abc"`.

```
class Solution {
public:
    bool isOneEditDistance(string s, string t) {
        int m = s.length(), n = t.length();
        if (m > n) return isOneEditDistance(t, s);
        if (n - m > 1) return false;
        bool mismatch = false;
        for (int i = 0; i < m; i++) {
            if (s[i] != t[i]) {
                if (m == n) s[i] = t[i];
                else s.insert(i, 1, t[i]);
                mismatch = true;
                break;
            }
        }
        return (!mismatch && n - m == 1) || (mismatch && s == t);
    }
};
```

written by jianchao.li.fighter original link here

Find Peak Element(162)

Answer 1

Consider that each local maximum is one valid peak. My solution is to find one local maximum with binary search. Binary search satisfies the O(logn) computational complexity.

Binary Search: recursion

```
class Solution {
public:

int findPeakElement(const vector<int> &num) {
    return Helper(num, 0, num.size()-1);
}
int Helper(const vector<int> &num, int low, int high)
{
    if(low == high)
        return low;
    else
    {
        int mid1 = (low+high)/2;
        int mid2 = mid1+1;
        if(num[mid1] > num[mid2])
            return Helper(num, low, mid1);
        else
            return Helper(num, mid2, high);
    }
}
};
```

Binary Search: iteration

```
class Solution {
public:
    int findPeakElement(const vector<int> &num)
    {
        int low = 0;
        int high = num.size()-1;

        while(low < high)
        {
            int mid1 = (low+high)/2;
            int mid2 = mid1+1;
            if(num[mid1] < num[mid2])
                low = mid2;
            else
                high = mid1;
        }
        return low;
    }
};
```

Sequential Search:

```
class Solution {
public:
    int findPeakElement(const vector<int> &num) {
        for(int i = 1; i < num.size(); i ++)
        {
            if(num[i] < num[i-1])
            {// <
                return i-1;
            }
        }
        return num.size()-1;
    }
};
```

written by gangan original link here

Answer 2

This problem is similar to Local Minimum. And according to the given condition, num[i] != num[i+1], there must exist a O(logN) solution. So we use binary search for this problem.

- If num[i-1] < num[i] > num[i+1], then num[i] is peak
- If num[i-1] < num[i] < num[i+1], then num[i+1...n-1] must contains a peak
- If num[i-1] > num[i] > num[i+1], then num[0...i-1] must contains a peak
- If num[i-1] > num[i] < num[i+1], then both sides have peak (n is num.length)

Here is the code

```
public int findPeakElement(int[] num) {
    return helper(num,0,num.length-1);
}

public int helper(int[] num,int start,int end){
    if(start == end){
        return start;
    }else if(start+1 == end){
        if(num[start] > num[end]) return start;
        return end;
    }else{

        int m = (start+end)/2;

        if(num[m] > num[m-1] && num[m] > num[m+1]){

            return m;

        }else if(num[m-1] > num[m] && num[m] > num[m+1]){

            return helper(num,start,m-1);

        }else{

            return helper(num,m+1,end);

        }

    }
}
```

written by xctom original link here

Answer 3

```
int findPeakElement(const vector<int> &num) {
    if (num.size() <= 1) return 0;
    int mid = 0, l = 0, h = num.size() - 1;

    while (l < h) {
        mid = (l + h) / 2;
        if (num[mid] > num[mid + 1])
            h = mid;
        else if (num[mid] < num[mid + 1])
            l = mid + 1;
    }

    return l;
}
```

written by shichaotan original link here

Missing Ranges(163)

Answer 1

```
public class Solution {
    public List<String> findMissingRanges(int[] A, int lower, int upper) {
        List<String> result = new ArrayList<String>();
        int pre = lower - 1;
        for(int i = 0 ; i <= A.length  ; i++){
            int after = i == A.length ? upper + 1 : A[i];
            if(pre + 2 == after){
                result.add(String.valueOf(pre + 1));
            }else if(pre + 2 < after){
                result.add(String.valueOf(pre + 1) + "->" + String.valueOf(after
- 1));
            }
            pre = after;
        }
        return result;
    }
}
```

written by jccg1000021953 original link here

Answer 2

```
public List<String> findMissingRanges(int[] a, int lo, int hi) {
  List<String> res = new ArrayList<String>();

  // the next number we need to find
  int next = lo;

  for (int i = 0; i < a.length; i++) {
    // not within the range yet
    if (a[i] < next) continue;

    // continue to find the next one
    if (a[i] == next) {
      next++;
      continue;
    }

    // get the missing range string format
    res.add(getRange(next, a[i] - 1));

    // now we need to find the next number
    next = a[i] + 1;
  }

  // do a final check
  if (next <= hi) res.add(getRange(next, hi));

  return res;
}

String getRange(int n1, int n2) {
  return (n1 == n2) ? String.valueOf(n1) : String.format("%d->%d", n1, n2);
}
```

written by jeantimex original link here

Answer 3

I noticed that OJ currently does not have test cases which involves extreme integer values, i.e. INT*MIN/INT*MAX. For instance, the following code:

```
vector<string> res;
char buf[50];
void addMissingRange(int left, int right, bool inc_left = false, bool inc_right =
false)
{
    if (right < left) return; // The range does not exist
    else if (right == left) sprintf(buf, "%d", left); // The range has only one e
lement
    else sprintf(buf, "%d->%d", left, right); // A two element range

    res.push_back(buf);
}

vector<string> findMissingRanges(int A[], int n, int lower, int upper) {
    int last = lower-1;
    for (int i = 0; i < n; ++i)
    {
        addMissingRange(last+1, A[i]-1);
        last = A[i];
    }
    addMissingRange(last+1, upper); // Add the last range.
    return res;
}
```

would pass OJ, but as a matter of fact, it fails on inputs like this:

```
A = [INT_MAX]; lower = 0, upper = INT_MAX;
```

The expected output should be: ["0->2147483646"],

but the actual output produced by the code above is: ["0->2147483646", "-2147483648->2147483647"]

It is because 'last+1' in the second last row overflows to INT_MIN, thus creating a giant range between 'last' and 'upper'.

So my questions are:

1. Do you guys think that we should add those corner cases to OJ?
2. If I would like to make sure my code works for ALL possible inputs, is there any elegant trick that I can use to avoid the overflow problem?

Thanks.

written by stellari original link here

Maximum Gap(164)

Answer 1

Suppose there are N elements in the array, the min value is***min*** and the max value is ***max***. Then the maximum gap will be no smaller than ceiling[(***max*** - ***min***) / (N - 1)].

Let gap = ceiling[(***max*** - ***min***) / (N - 1)]. We divide all numbers in the array into n-1 buckets, where k-th bucket contains all numbers in [***min*** + (k-1)gap, ***min*** + k*gap). Since there are n-2 numbers that are not equal ***min*** or ***max*** and there are n-1 buckets, at least one of the buckets are empty. We only need to store the largest number and the smallest number in each bucket.

After we put all the numbers into the buckets. We can scan the buckets sequentially and get the max gap. my blog for this problem

```java
public class Solution {
public int maximumGap(int[] num) {
    if (num == null || num.length < 2)
        return 0;
    // get the max and min value of the array
    int min = num[0];
    int max = num[0];
    for (int i:num) {
        min = Math.min(min, i);
        max = Math.max(max, i);
    }
    // the minimum possibale gap, ceiling of the integer division
    int gap = (int)Math.ceil((double)(max - min)/(num.length - 1));
    int[] bucketsMIN = new int[num.length - 1]; // store the min value in that bu
cket
    int[] bucketsMAX = new int[num.length - 1]; // store the max value in that bu
cket
    Arrays.fill(bucketsMIN, Integer.MAX_VALUE);
    Arrays.fill(bucketsMAX, Integer.MIN_VALUE);
    // put numbers into buckets
    for (int i:num) {
        if (i == min || i == max)
            continue;
        int idx = (i - min) / gap; // index of the right position in the buckets
        bucketsMIN[idx] = Math.min(i, bucketsMIN[idx]);
        bucketsMAX[idx] = Math.max(i, bucketsMAX[idx]);
    }
    // scan the buckets for the max gap
    int maxGap = Integer.MIN_VALUE;
    int previous = min;
    for (int i = 0; i < num.length - 1; i++) {
        if (bucketsMIN[i] == Integer.MAX_VALUE && bucketsMAX[i] == Integer.MIN_VA
LUE)
            // empty bucket
            continue;
        // min value minus the previous value is the current gap
        maxGap = Math.max(maxGap, bucketsMIN[i] - previous);
        // update previous bucket value
        previous = bucketsMAX[i];
    }
    maxGap = Math.max(maxGap, max - previous); // updata the final max value gap
    return maxGap;
}
```

}

written by zkfairytale original link here

Answer 2

Suppose you have n pigeons with labels and you put them into m holes based on their label with each hole of the same size. Why bother putting pigeons into holes? Because you want to disregard the distance between pigeons **within** each one hole.

Only when at least one hole is empty can we disregard the distance between pigeons

within each one hole and compute the maximum gap solely by the distance between pigeons **in adjacent holes**. We make sure that at least one hole is empty by using m=n-1 (i.e. n-2 pigeons in n-1 holes => at least one hole is empty).

```
int maximumGap(vector<int>& nums) {
        const int n = nums.size();
        if(n<=1) return 0;
        int maxE = *max_element(nums.begin(),nums.end());
        int minE = *min_element(nums.begin(),nums.end());
        double len = double(maxE-minE)/double(n-1);
        vector<int> maxA(n,INT_MIN);
        vector<int> minA(n,INT_MAX);
        for(int i=0; i<n; i++) {
            int index = (nums[i]-minE)/len;
            maxA[index] = max(maxA[index],nums[i]);
            minA[index] = min(minA[index],nums[i]);
        }
        int gap = 0, prev = maxA[0];
        for(int i=1; i<n; i++) {
            if(minA[i]==INT_MAX) continue;
            gap = max(gap,minA[i]-prev);
            prev = maxA[i];
        }
        return gap;
    }
```

written by morrischen2008 original link here

Answer 3

You can look at radix sort visualization here before reading the code:
https://www.cs.usfca.edu/~galles/visualization/RadixSort.html

```java
public class Solution {
public int maximumGap(int[] nums) {
    if (nums == null || nums.length < 2) {
        return 0;
    }

    // m is the maximal number in nums
    int m = nums[0];
    for (int i = 1; i < nums.length; i++) {
        m = Math.max(m, nums[i]);
    }

    int exp = 1; // 1, 10, 100, 1000 ...
    int R = 10; // 10 digits

    int[] aux = new int[nums.length];

    while (m / exp > 0) { // Go through all digits from LSB to MSB
        int[] count = new int[R];

        for (int i = 0; i < nums.length; i++) {
            count[(nums[i] / exp) % 10]++;
        }

        for (int i = 1; i < count.length; i++) {
            count[i] += count[i - 1];
        }

        for (int i = nums.length - 1; i >= 0; i--) {
            aux[--count[(nums[i] / exp) % 10]] = nums[i];
        }

        for (int i = 0; i < nums.length; i++) {
            nums[i] = aux[i];
        }
        exp *= 10;
    }

    int max = 0;
    for (int i = 1; i < aux.length; i++) {
        max = Math.max(max, aux[i] - aux[i - 1]);
    }

    return max;
}
```

}

1. The first step is to find the maximum value in nums array, it will be the threshold to end while loop.
2. Then use the radix sort algorithm to sort based on each digit from Least Significant Bit (LSB) to Most Significant Bit (MSB), that's exactly what's showing in the link.
3. `(nums[i] / exp) % 10` is used to get the digit, for each digit, basically the

digit itself serves as the index to access the count array. Count array stores the index to access aux array which stores the numbers after sorting based on the current digit.

4. Finally, find the maximum gap from sorted array.

Time and space complexities are both O(n). (Actually time is O(10n) at worst case for Integer.MAX_VALUE 2147483647)

written by Alexpanda original link here

Compare Version Numbers(165)

Answer 1

This code assumes that next level is zero if no mo levels in shorter version number. And than compare levels.

```
public int compareVersion(String version1, String version2) {
    String[] levels1 = version1.split("\\.");
    String[] levels2 = version2.split("\\.");

    int length = Math.max(levels1.length, levels2.length);
    for (int i=0; i<length; i++) {
        Integer v1 = i < levels1.length ? Integer.parseInt(levels1[i]) : 0;
        Integer v2 = i < levels2.length ? Integer.parseInt(levels2[i]) : 0;
        int compare = v1.compareTo(v2);
        if (compare != 0) {
            return compare;
        }
    }

    return 0;
}
```

written by pavel-shlyk original link here

Answer 2

```
int compareVersion(string version1, string version2) {
    int i = 0;
    int j = 0;
    int n1 = version1.size();
    int n2 = version2.size();

    int num1 = 0;
    int num2 = 0;
    while(i<n1 || j<n2)
    {
        while(i<n1 && version1[i]!='.'){
            num1 = num1*10+(version1[i]-'0');
            i++;
        }

        while(j<n2 && version2[j]!='.'){
            num2 = num2*10+(version2[j]-'0');;
            j++;
        }

        if(num1>num2) return 1;
        else if(num1 < num2) return -1;

        num1 = 0;
        num2 = 0;
        i++;
        j++;
    }

    return 0;
}
```

written by XUYAN3 original link here

Answer 3

I checked other Java solution and the basic idea is the same. In addition, I simply the logic by making the two version number same length. For example, if version1 = "1.0.2", and version2 = "1.0", the I will convert the version2 to "1.0.0".

```java
public int compareVersion(String version1, String version2) {

    String[] v1 = version1.split("\\.");
    String[] v2 = version2.split("\\.");

    for ( int i = 0; i < Math.max(v1.length, v2.length); i++ ) {
        int num1 = i < v1.length ? Integer.parseInt( v1[i] ) : 0;
        int num2 = i < v2.length ? Integer.parseInt( v2[i] ) : 0;
        if ( num1 < num2 ) {
            return -1;
        } else if ( num1 > num2 ) {
            return +1;
        }
    }

    return 0;
}
```

written by benjamin19890721 original link here

Fraction to Recurring Decimal(166)

Answer 1

```
// upgraded parameter types
string fractionToDecimal(int64_t n, int64_t d) {
    // zero numerator
    if (n == 0) return "0";

    string res;
    // determine the sign
    if (n < 0 ^ d < 0) res += '-';

    // remove sign of operands
    n = abs(n), d = abs(d);

    // append integral part
    res += to_string(n / d);

    // in case no fractional part
    if (n % d == 0) return res;

    res += '.';

    unordered_map<int, int> map;

    // simulate the division process
    for (int64_t r = n % d; r; r %= d) {

        // meet a known remainder
        // so we reach the end of the repeating part
        if (map.count(r) > 0) {
            res.insert(map[r], 1, '(');
            res += ')';
            break;
        }

        // the remainder is first seen
        // remember the current position for it
        map[r] = res.size();

        r *= 10;

        // append the quotient digit
        res += to_string(r / d);
    }

    return res;
}
```

written by mzchen original link here

Answer 2

The important thing is to consider all edge cases while thinking this problem

through, including: negative integer, possible overflow, etc.

Use HashMap to store a remainder and its associated index while doing the division so that whenever a same remainder comes up, we know there is a repeating fractional part.

Please comment if you see something wrong or can be improved. Cheers!

```
public class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
        if (numerator == 0) {
            return "0";
        }
        StringBuilder res = new StringBuilder();
        // "+" or "-"
        res.append(((numerator > 0) ^ (denominator > 0)) ? "-" : "");
        long num = Math.abs((long)numerator);
        long den = Math.abs((long)denominator);

        // integral part
        res.append(num / den);
        num %= den;
        if (num == 0) {
            return res.toString();
        }

        // fractional part
        res.append(".");
        HashMap<Long, Integer> map = new HashMap<Long, Integer>();
        map.put(num, res.length());
        while (num != 0) {
            num *= 10;
            res.append(num / den);
            num %= den;
            if (map.containsKey(num)) {
                int index = map.get(num);
                res.insert(index, "(");
                res.append(")");
                break;
            }
            else {
                map.put(num, res.length());
            }
        }
        return res.toString();
    }
}
```

written by dchen0215 original link here

Answer 3

```java
public String fractionToDecimal(int numerator, int denominator) {
    StringBuilder result = new StringBuilder();
    String sign = (numerator < 0 == denominator < 0 || numerator == 0) ? "" : "-"
;
    long num = Math.abs((long) numerator);
    long den = Math.abs((long) denominator);
    result.append(sign);
    result.append(num / den);
    long remainder = num % den;
    if (remainder == 0)
        return result.toString();
    result.append(".");
    HashMap<Long, Integer> hashMap = new HashMap<Long, Integer>();
    while (!hashMap.containsKey(remainder)) {
        hashMap.put(remainder, result.length());
        result.append(10 * remainder / den);
        remainder = 10 * remainder % den;
    }
    int index = hashMap.get(remainder);
    result.insert(index, "(");
    result.append(")");
    return result.toString().replace("(0)", "");
}
```

written by tusizi original link here

Two Sum II - Input array is sorted(167)

Answer 1

I know that the best solution is using two pointers like what is done in the previous solution sharing. However, I see the tag contains "binary search". I do not know if I misunderstand but is binary search a less efficient way for this problem.

Say, fix the first element A[0] and do binary search on the remaining n-1 elements. If cannot find any element which equals target-A[0], Try A[1]. That is, fix A[1] and do binary search on A[2]~A[n-1]. Continue this process until we have the last two elements A[n-2] and A[n-1].

Does this gives a time complexity lg(n-1) + lg(n-2) + ... + lg(1) ~ O(lg(n!)) ~ O(nlgn). So it is less efficient than the O(n) solution. Am I missing something here?

The code also passes OJ.

```
vector<int> twoSum(vector<int> &numbers, int target) {
    if(numbers.empty()) return {};
    for(int i=0; i<numbers.size()-1; i++) {
        int start=i+1, end=numbers.size()-1, gap=target-numbers[i];
        while(start <= end) {
            int m = start+(end-start)/2;
            if(numbers[m] == gap) return {i+1,m+1};
            else if(numbers[m] > gap) end=m-1;
            else start=m+1;
        }
    }
}
```

written by morrischen2008 original link here

Answer 2

Without HashMap, just have two pointers, A points to index 0, B points to index len - 1, shrink the scope based on the value and target comparison.

```java
public int[] twoSum(int[] num, int target) {
    int[] indice = new int[2];
    if (num == null || num.length < 2) return indice;
    int left = 0, right = num.length - 1;
    while (left < right) {
        int v = num[left] + num[right];
        if (v == target) {
            indice[0] = left + 1;
            indice[1] = right + 1;
            break;
        } else if (v > target) {
            right --;
        } else {
            left ++;
        }
    }
    return indice;
}
```

written by titanduan3 original link here

Answer 3

My algorithm is O(n), but runs 8ms, I am just wondering whether there is more efficient algorithm?

written by applelooking original link here

Excel Sheet Column Title(168)

Answer 1

Java:

```
return n == 0 ? "" : convertToTitle(--n / 26) + (char)('A' + (n % 26));
```

C++:

```
return n == 0 ? "" : convertToTitle(n / 26) + (char) (--n % 26 + 'A');
```

update: because the behavior of different compilers, the safe version should be:

```
return n == 0 ? "" : convertToTitle((n - 1) / 26) + (char) ((n - 1) % 26 + 'A');
```

Python:

```
return "" if num == 0 else self.convertToTitle((num - 1) / 26) + chr((num - 1) %
26 + ord('A'))
```

written by xcv58 original link here

Answer 2

```
public class Solution {
    public String convertToTitle(int n) {
        StringBuilder result = new StringBuilder();

        while(n>0){
            n--;
            result.insert(0, (char)('A' + n % 26));
            n /= 26;
        }

        return result.toString();
    }
}
```

written by murat original link here

Answer 3

```
string convertToTitle(int n) {
        string ans;
        while (n) {
            ans = char ((n - 1) % 26 + 'A') + ans;
            n = (n - 1) / 26;
        }
        return ans;
    }
```

written by shichaotan original link here

Majority Element(169)

Answer 1

```
public class Solution {
    public int majorityElement(int[] num) {

        int major=num[0], count = 1;
        for(int i=1; i<num.length;i++){
            if(count==0){
                count++;
                major=num[i];
            }else if(major==num[i]){
                count++;
            }else count--;

        }
        return major;
    }
}
```

written by jojocat1010 original link here

Answer 2

Well, if you have got this problem accepted, you may have noticed that there are 7 suggested solutions for this problem. The following passage will implement 6 of them except the `O(n^2)` brute force algorithm.

Hash Table

The hash-table solution is very straightforward. We maintain a mapping from each element to its number of appearances. While constructing the mapping, we update the majority element based on the max number of appearances we have seen. Notice that we do not need to construct the full mapping when we see that an element has appeared more than `n / 2` times.

The code is as follows, which should be self-explanatory.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        unordered_map<int, int> counts;
        int n = nums.size();
        for (int i = 0; i < n; i++)
            if (++counts[nums[i]] > n / 2)
                return nums[i];
    }
};
```

Sorting

Since the majority element appears more than `n / 2` times, the `n / 2`-th element in the sorted `nums` must be the majority element. This can be proved intuitively. Note that the majority element will take more than `n / 2` positions in the sorted `nums` (cover more than half of `nums`). If the first of it appears in the `0`-th position, it will also appear in the `n / 2`-th position to cover more than half of `nums`. It is similar if the last of it appears in the `n - 1`-th position. These two cases are that the contiguous chunk of the majority element is to the leftmost and the rightmost in `nums`. For other cases (imagine the chunk moves between the left and the right end), it must also appear in the `n / 2`-th position.

The code is as follows, being very short if we use the system `nth_element` (thanks for @qeatzy for pointing out such a nice function).

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        nth_element(nums.begin(), nums.begin() + nums.size() / 2, nums.end());
        return nums[nums.size() / 2];
    }
};
```

Randomization

This is a really nice idea and works pretty well (16ms running time on the OJ, almost fastest among the C++ solutions). The proof is already given in the suggested solutions.

The code is as follows, randomly pick an element and see if it is the majority one.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        srand(unsigned(time(NULL)));
        while (true) {
            int idx = rand() % n;
            int candidate = nums[idx];
            int counts = 0;
            for (int i = 0; i < n; i++)
                if (nums[i] == candidate)
                    counts++;
            if (counts > n / 2) return candidate;
        }
    }
};
```

Divide and Conquer

This idea is very algorithmic. However, the implementation of it requires some careful thought about the base cases of the recursion. The base case is that when the array has only one element, then it is the majority one. This solution takes 24ms.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        return majority(nums, 0, nums.size() - 1);
    }
private:
    int majority(vector<int>& nums, int left, int right) {
        if (left == right) return nums[left];
        int mid = left + ((right - left) >> 1);
        int lm = majority(nums, left, mid);
        int rm = majority(nums, mid + 1, right);
        if (lm == rm) return lm;
        return count(nums.begin() + left, nums.begin() + right + 1, lm) > count(n
ums.begin() + left, nums.begin() + right + 1, rm) ? lm : rm;
    }
};
```

Moore Voting Algorithm

A brilliant and easy-to-implement algorithm! It also runs very fast, about 20ms.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int major, counts = 0, n = nums.size();
        for (int i = 0; i < n; i++) {
            if (!counts) {
                major = nums[i];
                counts = 1;
            }
            else counts += (nums[i] == major) ? 1 : -1;
        }
        return major;
    }
};
```

Bit Manipulation

Another nice idea! The key lies in how to count the number of `1`'s on a specific bit. Specifically, you need a `mask` with a `1` on the `i`-the bit and `0` otherwise to get the `i`-th bit of each element in `nums`. The code is as follows.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int major = 0, n = nums.size();
        for (int i = 0, mask = 1; i < 32; i++, mask <<= 1) {
            int bitCounts = 0;
            for (int j = 0; j < n; j++) {
                if (nums[j] & mask) bitCounts++;
                if (bitCounts > n / 2) {
                    major |= mask;
                    break;
                }
            }
        }
        return major;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

This can be solved by Moore's voting algorithm. Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element. Below code loops through each element and maintains a count of the element that has the potential of being the majority element. If next element is same then increments the count, otherwise decrements the count. If the count reaches 0 then update the potential index to the current element and sets count to 1.

```
int majorityElement(vector<int> &num) {
    int majorityIndex = 0;
    for (int count = 1, i = 1; i < num.size(); i++) {
        num[majorityIndex] == num[i] ? count++ : count--;
        if (count == 0) {
            majorityIndex = i;
            count = 1;
        }
    }

    return num[majorityIndex];
}
```

written by satyakam original link here

Two Sum III - Data structure design(170)

Answer 1

I use HashMap to store times of number be added.

When find be called, we iterate the keys of HashMap, then find another number minus by value. Then combine the detections together.

Java:

```
public class TwoSum {
    private HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

    public void add(int number) {
        map.put(number, map.containsKey(number) ? map.get(number) + 1 : 1);
    }

    public boolean find(int value) {
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            int i = entry.getKey();
            int j = value - i;
            if ((i == j && entry.getValue() > 1) || (i != j && map.containsKey(j)
)) {
                return true;
            }
        }
        return false;
    }
}
```

C++:

```
class TwoSum {
    unordered_map<int,int> map;
public:
    void add(int number) {
        map[number]++;
    }

    bool find(int value) {
        for (unordered_map<int,int>::iterator it = map.begin(); it != map.end();
it++) {
            int i = it->first;
            int j = value - i;
            if ((i == j && it->second > 1) || (i != j && map.find(j) != map.end()
)) {
                return true;
            }
        }
        return false;
    }
};
```

Python:

```
class TwoSum:

    # initialize your data structure here
    def __init__(self):
        self.table = dict()

    # @return nothing
    def add(self, number):
        self.table[number] = self.table.get(number, 0) + 1;

    # @param value, an integer
    # @return a Boolean
    def find(self, value):
        for i in self.table.keys():
            j = value - i
            if i == j and self.table.get(i) > 1 or i != j and self.table.get(j, 0
) > 0:
                return True
        return False
```

written by xcv58 original link here

Answer 2

```
class TwoSum {
    unordered_multiset<int> nums;
public:
    void add(int number) {
        nums.insert(number);
    }
    bool find(int value) {
        for (int i : nums) {
            int count = i == value - i ? 1 : 0;
            if (nums.count(value - i) > count) {
                return true;
            }
        }
        return false;
    }
};
```

written by RayZ_O original link here

Answer 3

I copied and pasted solution directly from Handbook, but it returns TLE for the long input test case. The test case is too long to paste here. Please fix it.

written by boa1150 original link here

Excel Sheet Column Number(171)

Answer 1

Java:

```
int result = 0;
for (int i = 0; i < s.length(); result = result * 26 + (s.charAt(i) - 'A' + 1), i
++);
return result;
```

C++:

```
int result = 0;
for (int i = 0; i < s.size(); result = result * 26 + (s.at(i) - 'A' + 1), i++);
return result;
```

Python:

```
return reduce(lambda x, y : x * 26 + y, [ord(c) - 64 for c in list(s)])
```

Python version is beautiful because reduce function and list comprehensive.

I don't know whether exist similar approach to achieve one line solution in Java or C++. One possible way is defining another method like this:

```
public int titleToNumber(int num, String s)
```

to store previous result and make recursive call. But this add much more lines.

written by xcv58 original link here

Answer 2

I was asked of this question during an interview with microsoft. The interviewer asked whether I want a coding question or a brain teaser, I asked for the latter and here comes the question. I did not do it very well at that time, though.

written by dengruixistudent original link here

Answer 3

Here is my Java solution. Similar to the number to title.

```
public int titleToNumber(String s) {
    int result = 0;
    for(int i = 0 ; i < s.length(); i++) {
      result = result * 26 + (s.charAt(i) - 'A' + 1);
    }
    return result;
  }
```

written by darko1002001 original link here

Factorial Trailing Zeroes(172)

Answer 1

This question is pretty straightforward.

Because all trailing 0 is from factors 5 * 2.

But sometimes one number may have several 5 factors, for example, 25 have two 5 factors, 125 have three 5 factors. In the n! operation, factors 2 is always ample. So we just count how many 5 factors in all number from 1 to n.

One line code:

Java:

```
    return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
```

C++:

```
    return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
```

Python:

```
    return 0 if n == 0 else n / 5 + self.trailingZeroes(n / 5)
```

written by xcv58 original link here

Answer 2

The idea is:

1. The ZERO comes from 10.
2. The 10 comes from 2 x 5
3. And we need to account for all the products of 5 and 2. likes 4Ã—5 = 20 ...
4. So, if we take all the numbers with 5 as a factor, we'll have way more than enough even numbers to pair with them to get factors of 10

Example One

How many multiples of 5 are between 1 and 23? There is 5, 10, 15, and 20, for four multiples of 5. Paired with 2's from the even factors, this makes for four factors of 10, so: **23! has 4 zeros**.

Example Two

How many multiples of 5 are there in the numbers from 1 to 100?

because 100 Ã· 5 = 20, so, there are twenty multiples of 5 between 1 and 100.

but wait, actually 25 is 5Ã—5, so each multiple of 25 has an extra factor of 5, e.g. 25 Ã— 4 = 100ï¼Œwhich introduces extra of zero.

So, we need know how many multiples of 25 are between 1 and 100? Since 100 Ã· 25

= 4, there are four multiples of 25 between 1 and 100.

Finally, we get 20 + 4 = 24 trailing zeroes in 100!

The above example tell us, we need care about 5, 5Ã—5, 5Ã—5Ã—5, 5Ã—5Ã—5Ã—5 ....

Example Three

By given number 4617.

5^1 : 4617 Ã· 5 = 923.4, so we get 923 factors of 5

5^2 : 4617 Ã· 25 = 184.68, so we get 184 additional factors of 5

5^3 : 4617 Ã· 125 = 36.936, so we get 36 additional factors of 5

5^4 : 4617 Ã· 625 = 7.3872, so we get 7 additional factors of 5

5^5 : 4617 Ã· 3125 = 1.47744, so we get 1 more factor of 5

5^6 : 4617 Ã· 15625 = 0.295488, which is less than 1, so stop here.

Then 4617! has 923 + 184 + 36 + 7 + 1 = 1151 trailing zeroes.

C/C++ code

```
int trailingZeroes(int n) {
    int result = 0;
    for(long long i=5; n/i>0; i*=5){
        result += (n/i);
    }
    return result;
}
```

---------update----------

To avoid the integer overflow as **@localvar** mentioned below(in case of 'n >=1808548329'), the expression " i <= INT*MAX/5" is not a good way to prevent overflow, because 5^13 is > INT*MAX/5 and it's valid.

So, if you want to use "multiply", consider define the 'i' as 'long long' type.

Or, take the solution **@codingryan** mentioned in below answer!

written by haoel original link here

Answer 3

10 is the product of 2 and 5. In n!, we need to know how many 2 and 5, and the number of zeros is the minimum of the number of 2 and the number of 5.

Since multiple of 2 is more than multiple of 5, the number of zeros is dominant by the number of 5.

Here we expand

```
  2147483647!
=2 * 3 * ...* 5 ... *10 ... 15* ... * 25 ... * 50 ... * 125 ... * 250...
=2 * 3 * ...* 5 ... * (5^1*2)...(5^1*3)...*(5^2*1)...*(5^2*2)...*(5^3*1)...*(5^3*
2)... (Equation 1)
```

We just count the number of 5 in Equation 1.

Multiple of 5 provides one 5, multiple of 25 provides two 5 and so on.

Note the duplication: multiple of 25 is also multiple of 5, so multiple of 25 only provides one extra 5.

Here is the basic solution:

```
return n/5 + n/25 + n/125 + n/625 + n/3125+...;
```

You can easily rewrite it to a loop.

written by gqq original link here

Binary Search Tree Iterator(173)

Answer 1

I use Stack to store directed left children from root. When next() be called, I just pop one element and process its right child as new root. The code is pretty straightforward.

So this can satisfy O(h) memory, hasNext() in O(1) time, But next() is O(h) time.

I can't find a solution that can satisfy both next() in O(1) time, space in O(h).

Java:

```
public class BSTIterator {
    private Stack<TreeNode> stack = new Stack<TreeNode>();

    public BSTIterator(TreeNode root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode tmpNode = stack.pop();
        pushAll(tmpNode.right);
        return tmpNode.val;
    }

    private void pushAll(TreeNode node) {
        for (; node != null; stack.push(node), node = node.left);
    }
}
```

C++:

```
class BSTIterator {
    stack<TreeNode *> myStack;
public:
    BSTIterator(TreeNode *root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        pushAll(tmpNode->right);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for (; node != NULL; myStack.push(node), node = node->left);
    }
};
```

Python:

```
class BSTIterator:
    # @param root, a binary search tree's root node
    def __init__(self, root):
        self.stack = list()
        self.pushAll(root)

    # @return a boolean, whether we have a next smallest number
    def hasNext(self):
        return self.stack

    # @return an integer, the next smallest number
    def next(self):
        tmpNode = self.stack.pop()
        self.pushAll(tmpNode.right)
        return tmpNode.val

    def pushAll(self, node):
        while node is not None:
            self.stack.append(node)
            node = node.left
```

written by xcv58 original link here

Answer 2

My idea comes from this: My first thought was to use inorder traversal to put every

node into an array, and then make an index pointer for the next() and hasNext(). That meets the O(1) run time but not the O(h) memory. O(h) is really much more less than O(n) when the tree is huge.

This means I cannot use a lot of memory, which suggests that I need to make use of the tree structure itself. And also, one thing to notice is the "average O(1) run time". It's weird to say average O(1), because there's nothing below O(1) in run time, which suggests in most cases, I solve it in O(1), while in some cases, I need to solve it in O(n) or O(h). These two limitations are big hints.

Before I come up with this solution, I really draw a lot binary trees and try inorder traversal on them. We all know that, once you get to a TreeNode, in order to get the smallest, you need to go all the way down its left branch. So our first step is to point to pointer to the left most TreeNode. The problem is how to do back trace. Since the TreeNode doesn't have father pointer, we cannot get a TreeNode's father node in O(1) without store it beforehand. Back to the first step, when we are traversal to the left most TreeNode, we store each TreeNode we met (They are all father nodes for back trace).

After that, I try an example, for next(), I directly return where the pointer pointing at, which should be the left most TreeNode I previously found. What to do next? After returning the smallest TreeNode, I need to point the pointer to the next smallest TreeNode. When the current TreeNode has a right branch (It cannot have left branch, remember we traversal to the left most), we need to jump to its right child first and then traversal to its right child's left most TreeNode. When the current TreeNode doesn't have a right branch, it means there cannot be a node with value smaller than itself father node, point the pointer at its father node.

The overall thinking leads to the structure Stack, which fits my requirement so well.

```java
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

public class BSTIterator {

    private Stack<TreeNode> stack;
    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null){
            stack.push(cur);
            if(cur.left != null)
                cur = cur.left;
            else
                break;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode node = stack.pop();
        TreeNode cur = node;
        // traversal right branch
        if(cur.right != null){
            cur = cur.right;
            while(cur != null){
                stack.push(cur);
                if(cur.left != null)
                    cur = cur.left;
                else
                    break;
            }
        }
        return node.val;
    }
}

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = new BSTIterator(root);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

written by siyang2 original link here

Answer 3

the idea is same as using stack to do Binary Tree Inorder Traversal

```
public class BSTIterator {

        Stack<TreeNode> stack =  null ;
        TreeNode current = null ;

        public BSTIterator(TreeNode root) {
              current = root;
              stack = new Stack<> ();
        }

        /** @return whether we have a next smallest number */
        public boolean hasNext() {
              return !stack.isEmpty() || current != null;
        }

            /** @return the next smallest number */
        public int next() {
            while (current != null) {
                stack.push(current);
                current = current.left ;
            }
            TreeNode t = stack.pop() ;
            current = t.right ;
            return t.val ;
        }
    }
```

written by scott original link here

Dungeon Game(174)

Answer 1

Use hp[i][j] to store the min hp needed at position (i, j), then do the calculation from right-bottom to left-up.

Note: adding dummy row and column would make the code cleaner.

```
class Solution {
public:
    int calculateMinimumHP(vector<vector<int> > &dungeon) {
        int M = dungeon.size();
        int N = dungeon[0].size();
        // hp[i][j] represents the min hp needed at position (i, j)
        // Add dummy row and column at bottom and right side
        vector<vector<int> > hp(M + 1, vector<int>(N + 1, INT_MAX));
        hp[M][N - 1] = 1;
        hp[M - 1][N] = 1;
        for (int i = M - 1; i >= 0; i--) {
            for (int j = N - 1; j >= 0; j--) {
                int need = min(hp[i + 1][j], hp[i][j + 1]) - dungeon[i][j];
                hp[i][j] = need <= 0 ? 1 : need;
            }
        }
        return hp[0][0];
    }
};
```

written by sidbai original link here

Answer 2

```
public int calculateMinimumHP(int[][] dungeon) {
    if (dungeon == null || dungeon.length == 0 || dungeon[0].length == 0) return
0;

    int m = dungeon.length;
    int n = dungeon[0].length;

    int[][] health = new int[m][n];

    health[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

    for (int i = m - 2; i >= 0; i--) {
        health[i][n - 1] = Math.max(health[i + 1][n - 1] - dungeon[i][n - 1], 1)
;
    }

    for (int j = n - 2; j >= 0; j--) {
        health[m - 1][j] = Math.max(health[m - 1][j + 1] - dungeon[m - 1][j], 1);
    }

    for (int i = m - 2; i >= 0; i--) {
        for (int j = n - 2; j >= 0; j--) {
            int down = Math.max(health[i + 1][j] - dungeon[i][j], 1);
            int right = Math.max(health[i][j + 1] - dungeon[i][j], 1);
            health[i][j] = Math.min(right, down);
        }
    }

    return health[0][0];
}
```

written by vime2 original link here

Answer 3

Here is my solution using dp and rolling array --Dungeon Game:

```
int calculateMinimumHP(vector<vector<int> > &dungeon) {
    const int m = dungeon.size();
    const int n = dungeon[0].size();
    vector<int> dp(n + 1, INT_MAX);
    dp[n - 1] = 1;
    for(int i = m - 1; i >= 0; --i)
        for(int j = n - 1; j >= 0; --j)
            dp[j] = getMin(min(dp[j], dp[j + 1]) - dungeon[i][j]);
    return dp[0];
}
int getMin(int n){
    return n <= 0 ? 1 : n;
}
```

Note: Update from right to left and from bottom up.

written by flyingPig original link here

Largest Number(179)

Answer 1

```
public  String largestNumber(int[] num) {
    if(num==null || num.length==0)
        return "";
    String[] Snum = new String[num.length];
    for(int i=0;i<num.length;i++)
        Snum[i] = num[i]+"";

    Comparator<String> comp = new Comparator<String>(){
        @Override
        public int compare(String str1, String str2){
            String s1 = str1+str2;
            String s2 = str2+str1;
            return s1.compareTo(s2);
        }
    };

    Arrays.sort(Snum,comp);
    if(Snum[Snum.length-1].charAt(0)=='0')
        return "0";

    StringBuilder sb = new StringBuilder();

    for(String s: Snum)
        sb.insert(0, s);

    return sb.toString();

}
```

written by ran3 original link here

Answer 2

```
class Solution {
public:
    string largestNumber(vector<int> &num) {
        vector<string> arr;
        for(auto i:num)
            arr.push_back(to_string(i));
        sort(begin(arr), end(arr), [](string &s1, string &s2){ return s1+s2>s2+s1
; });
        string res;
        for(auto s:arr)
            res+=s;
        while(res[0]=='0' && res.length()>1)
            res.erase(0,1);
        return  res;
    }
};
```

written by isaac7 original link here

Answer 3

The logic is pretty straightforward. Just compare number by convert it to string.

Thanks for Java 8, it makes code beautiful.

Java:

```
public class Solution {
    public String largestNumber(int[] num) {
        String[] array = Arrays.stream(num).mapToObj(String::valueOf).toArray(Str
ing[]::new);
        Arrays.sort(array, (String s1, String s2) -> (s2 + s1).compareTo(s1 + s2)
);
        return Arrays.stream(array).reduce((x, y) -> x.equals("0") ? y : x + y).g
et();
    }
}
```

Python:

```
class Solution:
    # @param num, a list of integers
    # @return a string
    def largestNumber(self, num):
        num = [str(x) for x in num]
        num.sort(cmp=lambda x, y: cmp(y+x, x+y))
        return ''.join(num).lstrip('0') or '0'
```

written by xcv58 original link here

Reverse Words in a String II(186)

Answer 1

```
public void reverseWords(char[] s) {
    // Three step to reverse
    // 1, reverse the whole sentence
    reverse(s, 0, s.length - 1);
    // 2, reverse each word
    int start = 0;
    int end = -1;
    for (int i = 0; i < s.length; i++) {
        if (s[i] == ' ') {
            reverse(s, start, i - 1);
            start = i + 1;
        }
    }
    // 3, reverse the last word, if there is only one word this will solve the cor
ner case
    reverse(s, start, s.length - 1);
}

public void reverse(char[] s, int start, int end) {
    while (start < end) {
        char temp = s[start];
        s[start] = s[end];
        s[end] = temp;
        start++;
        end--;
    }
}
```

written by xzhu%40okabaminc.com original link here

Answer 2

```
void reverseWords(string &s) {
    reverse(s.begin(), s.end());
    for (int i = 0, j = 0; i < s.size(); i = j + 1) {
        for (j = i; j < s.size() && !isblank(s[j]); ++j);
        reverse(s.begin()+i, s.begin()+j);
    }
}
```

written by weibest original link here

Answer 3

```
class Solution {
public:
    void reverse_s(string& s, int l, int r) {
        if (l==r) return;

        while (l<r) {
            char tmp = s[l];
            s[l] = s[r];
            s[r] = tmp;
            l++;
            r--;
        }
    }

    void reverseWords(string &s) {
        if (s.size() <2) return;

        reverse_s(s, 0, s.size()-1);

        int i = 0;
        while ( i < s.size()) {
            int pos = s.find(" ", i);
            if (pos == -1) pos = s.size();
            reverse_s(s, i, pos-1);
            i = pos+1;
        }
    }
};
```

written by smileyogurt.966 original link here

Repeated DNA Sequences(187)

Answer 1

The main idea is to store the substring as int in map to bypass the memory limits.

There are only four possible character A, C, G, and T, but I want to use 3 bits per letter instead of 2.

Why? It's easier to code.

A is 0x41, C is 0x43, G is 0x47, T is 0x54. Still don't see it? Let me write it in octal.

A is 0101, C is 0103, G is 0107, T is 0124. The last digit in octal are different for all four letters. That's all we need!

We can simply use `s[i] & 7` to get the last digit which are just the last 3 bits, it's much easier than lookup table or switch or a bunch of if and else, right?

We don't really need to generate the substring from the int. While counting the number of occurrences, we can push the substring into result as soon as the count becomes 2, so there won't be any duplicates in the result.

```
vector<string> findRepeatedDnaSequences(string s) {
    unordered_map<int, int> m;
    vector<string> r;
    int t = 0, i = 0, ss = s.size();
    while (i < 9)
        t = t << 3 | s[i++] & 7;
    while (i < ss)
        if (m[t = t << 3 & 0x3FFFFFFF | s[i++] & 7]++ == 1)
            r.push_back(s.substr(i - 10, 10));
    return r;
}
```

BTW, the OJ doesn't seems to have test cases which the given string length is smaller than 9, so I didn't check it to make the code simpler.

Any suggestions?

Update:

I realised that I can use `s[i] >> 1 & 3` to get 2 bits, but then I won't be able to remove the first loop as 1337c0d3r suggested.

written by sen original link here

Answer 2

```java
public List<String> findRepeatedDnaSequences(String s) {
    Set<Integer> words = new HashSet<>();
    Set<Integer> doubleWords = new HashSet<>();
    List<String> rv = new ArrayList<>();
    char[] map = new char[26];
    //map['A' - 'A'] = 0;
    map['C' - 'A'] = 1;
    map['G' - 'A'] = 2;
    map['T' - 'A'] = 3;

    for(int i = 0; i < s.length() - 9; i++) {
        int v = 0;
        for(int j = i; j < i + 10; j++) {
            v <<= 2;
            v |= map[s.charAt(j) - 'A'];
        }
        if(!words.add(v) && doubleWords.add(v)) {
            rv.add(s.substring(i, i + 10));
        }
    }
    return rv;
}
```

written by crazyirontoiletpaper original link here

Answer 3

Hi guys!

The idea is to use rolling hash technique or in case of string search also known as Rabin-Karp algorithm. As our alphabet A consists of only 4 letters we can be not afraid of collisions. The hash for a current window slice could be found in a constant time by subtracting the former first character times size of the A in the power of 9 and updating remaining hash by the standard rule: hash = hash*A.size() + curr_char.

Check out the Java code below.

Hope it helps!

```java
public class Solution {
    private static final Map<Character, Integer> A = new HashMap<>();
    static { A.put('A',0); A.put('C',1); A.put('G',2); A.put('T',3); }
    private final int A_SIZE_POW_9 = (int) Math.pow(A.size(), 9);

    public List<String> findRepeatedDnaSequences(String s) {
        Set<String> res = new HashSet<>();
        Set<Integer> hashes = new HashSet<>();
        for (int i = 0, rhash = 0; i < s.length(); i++) {
            if (i > 9) rhash -= A_SIZE_POW_9 * A.get(s.charAt(i-10));
            rhash = A.size() * rhash + A.get(s.charAt(i));
            if (i > 8 && !hashes.add(rhash)) res.add(s.substring(i-9,i+1));
        }
        return new ArrayList<>(res);
    }
}
```

written by shpolsky original link here

Best Time to Buy and Sell Stock IV(188)

Answer 1

The general idea is DP, while I had to add a "quickSolve" function to tackle some corner cases to avoid TLE.

DP: t(i,j) is the max profit for up to i transactions by time j (0<=i<=K, 0<=j<=T).

```
public int maxProfit(int k, int[] prices) {
    int len = prices.length;
    if (k >= len / 2) return quickSolve(prices);

    int[][] t = new int[k + 1][len];
    for (int i = 1; i <= k; i++) {
        int tmpMax =  -prices[0];
        for (int j = 1; j < len; j++) {
            t[i][j] = Math.max(t[i][j - 1], prices[j] + tmpMax);
            tmpMax =  Math.max(tmpMax, t[i - 1][j - 1] - prices[j]);
        }
    }
    return t[k][len - 1];
}


private int quickSolve(int[] prices) {
    int len = prices.length, profit = 0;
    for (int i = 1; i < len; i++)
        // as long as there is a price gap, we gain a profit.
        if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];
    return profit;
}
```

written by jinrf original link here

Answer 2

We can find all adjacent valley/peak pairs and calculate the profits easily. Instead of accumulating all these profits like Buy&Sell Stock II, we need the highest k ones.

The key point is when there are two v/p pairs (v1, p1) and (v2, p2), satisfying v1 <= v2 and p1 <= p2, we can either make one transaction at [v1, p2], or make two at both [v1, p1] and [v2, p2]. The trick is to treat [v1, p2] as the first transaction, and [v2, p1] as the second. Then we can guarantee the right max profits in both situations, **p2 - v1** for one transaction and **p1 - v1 + p2 - v2** for two.

Finding all v/p pairs and calculating the profits takes O(n) since there are up to n/2 such pairs. And extracting k maximums from the heap consumes another O(klgn).

```
class Solution {
public:
    int maxProfit(int k, vector<int> &prices) {
        int n = (int)prices.size(), ret = 0, v, p = 0;
        priority_queue<int> profits;
        stack<pair<int, int> > vp_pairs;
        while (p < n) {
            // find next valley/peak pair
            for (v = p; v < n - 1 && prices[v] >= prices[v+1]; v++);
            for (p = v + 1; p < n && prices[p] >= prices[p-1]; p++);
            // save profit of 1 transaction at last v/p pair, if current v is low
er than last v
            while (!vp_pairs.empty() && prices[v] < prices[vp_pairs.top().first])
{
                profits.push(prices[vp_pairs.top().second-1] - prices[vp_pairs.to
p().first]);
                vp_pairs.pop();
            }
            // save profit difference between 1 transaction (last v and current p
) and 2 transactions (last v/p + current v/p),
            // if current v is higher than last v and current p is higher than la
st p
            while (!vp_pairs.empty() && prices[p-1] >= prices[vp_pairs.top().seco
nd-1]) {
                profits.push(prices[vp_pairs.top().second-1] - prices[v]);
                v = vp_pairs.top().first;
                vp_pairs.pop();
            }
            vp_pairs.push(pair<int, int>(v, p));
        }
        // save profits of the rest v/p pairs
        while (!vp_pairs.empty()) {
            profits.push(prices[vp_pairs.top().second-1] - prices[vp_pairs.top().
first]);
            vp_pairs.pop();
        }
        // sum up first k highest profits
        for (int i = 0; i < k && !profits.empty(); i++) {
            ret += profits.top();
            profits.pop();
        }
        return ret;
    }
};
```

written by yishiluo original link here

Answer 3

This is my DP solution:

class Solution {

```
public:
    int maxProfit(int k, vector<int> &prices) {
        int len = prices.size();
        if (len<2) return 0;
        if (k>len/2){ // simple case
            int ans = 0;
            for (int i=1; i<len; ++i){
                ans += max(prices[i] - prices[i-1],0);
            }
            return ans;
        }
        int hold[k+1];
        int rele[k+1];
        for (int i=0;i<=k;++i){
            hold[i] = INT_MIN;
            rele[i] = 0;
        }
        int cur;
        for (int i=0; i<len; ++i){
            cur = prices[i];
            for(int j=k; j>0; --j){
                rele[j] = max(rele[j],hold[j] + cur);
                hold[j] = max(hold[j],rele[j-1] - cur);
            }
        }
        return rele[k];
    }
```

};

Inspired by weijiac in Best Time to Buy and Sell Stock III

https://leetcode.com/discuss/18330/is-it-best-solution-with-o-n-o-1

written by hanhongsun original link here

Rotate Array(189)

Answer 1

1. Make an extra copy and then rotate.

Time complexity: O(n). Space complexity: O(n).

```
class Solution
{
public:
    void rotate(int nums[], int n, int k)
    {
        if ((n == 0) || (k <= 0))
        {
            return;
        }

        // Make a copy of nums
        vector<int> numsCopy(n);
        for (int i = 0; i < n; i++)
        {
            numsCopy[i] = nums[i];
        }

        // Rotate the elements.
        for (int i = 0; i < n; i++)
        {
            nums[(i + k)%n] = numsCopy[i];
        }
    }
};
```

2. Start from one element and keep rotating until we have rotated n different elements.

Time complexity: O(n). Space complexity: O(1).

```
class Solution
{
public:
    void rotate(int nums[], int n, int k)
    {
        if ((n == 0) || (k <= 0))
        {
            return;
        }

        int cntRotated = 0;
        int start = 0;
        int curr = 0;
        int numToBeRotated = nums[0];
        int tmp = 0;
        // Keep rotating the elements until we have rotated n
        // different elements.
        while (cntRotated < n)
        {
            do
            {
                tmp = nums[(curr + k)%n];
                nums[(curr+k)%n] = numToBeRotated;
                numToBeRotated = tmp;
                curr = (curr + k)%n;
                cntRotated++;
            } while (curr != start);
            // Stop rotating the elements when we finish one cycle,
            // i.e., we return to start.

            // Move to next element to start a new cycle.
            start++;
            curr = start;
            numToBeRotated = nums[curr];
        }
    }
};
```

3. Reverse the first n - k elements, the last k elements, and then all the n elements.

Time complexity: O(n). Space complexity: O(1).

```
    class Solution
    {
    public:
        void rotate(int nums[], int n, int k)
        {
            k = k%n;

            // Reverse the first n - k numbers.
            // Index i (0 <= i < n - k) becomes n - k - i.
            reverse(nums, nums + n - k);

            // Reverse tha last k numbers.
            // Index n - k + i (0 <= i < k) becomes n - i.
            reverse(nums + n - k, nums + n);

            // Reverse all the numbers.
            // Index i (0 <= i < n - k) becomes n - (n - k - i) = i + k.
            // Index n - k + i (0 <= i < k) becomes n - (n - i) = i.
            reverse(nums, nums + n);
        }
    };
```

4. Swap the last k elements with the first k elements.

Time complexity: O(n). Space complexity: O(1).

```
class Solution
{
public:
    void rotate(int nums[], int n, int k)
    {
        for (; k = k%n; n -= k, nums += k)
        {
            // Swap the last k elements with the first k elements.
            // The last k elements will be in the correct positions
            // but we need to rotate the remaining (n - k) elements
            // to the right by k steps.
            for (int i = 0; i < k; i++)
            {
                swap(nums[i], nums[n - k + i]);
            }
        }
    }
};
```

5. Keep swapping two subarrays.

Time complexity: O(n). Space complexity: O(1).

```
class Solution
{
public:
    void rotate(int nums[], int n, int k)
```

```
    {
        if ((n == 0) || (k <= 0) || (k%n == 0))
        {
            return;
        }

        k = k%n;
        // Rotation to the right by k steps is equivalent to swapping
        // the two subarrays nums[0,...,n - k - 1] and nums[n - k,...,n - 1].
        int start = 0;
        int tmp = 0;
        while (k > 0)
        {
            if (n - k >= k)
            {
                // The left subarray with size n - k is longer than
                // the right subarray with size k. Exchange
                // nums[n - 2*k,...,n - k - 1] with nums[n - k,...,n - 1].
                for (int i = 0; i < k; i++)
                {
                    tmp = nums[start + n - 2*k + i];
                    nums[start + n - 2*k + i] = nums[start + n - k + i];
                    nums[start + n - k + i] = tmp;
                }

                // nums[n - 2*k,...,n - k - 1] are in their correct positions now
.
                // Need to rotate the elements of nums[0,...,n - k - 1] to the ri
ght
                // by k%n steps.
                n = n - k;
                k = k%n;
            }
            else
            {
                // The left subarray with size n - k is shorter than
                // the right subarray with size k. Exchange
                // nums[0,...,n - k - 1] with nums[n - k,...,2*(n - k) - 1].
                for (int i = 0; i < n - k; i++)
                {
                    tmp = nums[start + i];
                    nums[start + i] = nums[start + n - k + i];
                    nums[start + n - k + i] = tmp;
                }

                // nums[n - k,...,2*(n - k) - 1] are in their correct positions n
ow.
                // Need to rotate the elements of nums[n - k,...,n - 1] to the ri
ght
                // by k - (n - k) steps.
                tmp = n - k;
                n = k;
                k -= tmp;
                start += tmp;
            }
        }
    }
```

```
};
```

written by zhukov original link here

Answer 2

```
void rotate(int nums[], int n, int k) {
    reverse(nums,nums+n);
    reverse(nums,nums+k%n);
    reverse(nums+k%n,nums+n);
}
```

written by monaziyi original link here

Answer 3

I really don't like those *something little* line solutions as they are incredibly hard to read. Below is my solution.

```
public void rotate(int[] nums, int k) {
    k %= nums.length;
    reverse(nums, 0, nums.length - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, nums.length - 1);
}

public void reverse(int[] nums, int start, int end) {
    while (start < end) {
        int temp = nums[start];
        nums[start] = nums[end];
        nums[end] = temp;
        start++;
        end--;
    }
}
```

written by danny6514 original link here

Reverse Bits(190)

Answer 1

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        n = (n >> 16) | (n << 16);
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
        return n;
    }
};
```

for 8 bit binary number *abcdefgh*, the process is as follow:

abcdefgh -> efghabcd -> ghefcdab -> hgfedcba

written by tworuler original link here

Answer 2

The Java solution is straightforward, just bitwise operation:

```
public int reverseBits(int n) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result += n & 1;
        n >>>= 1;   // CATCH: must do unsigned shift
        if (i < 31) // CATCH: for last digit, don't shift!
            result <<= 1;
    }
    return result;
}
```

How to optimize if this function is called multiple times? We can divide an int into 4 bytes, and reverse each byte then combine into an int. For each byte, we can use cache to improve performance.

```java
// cache
private final Map<Byte, Integer> cache = new HashMap<Byte, Integer>();
public int reverseBits(int n) {
    byte[] bytes = new byte[4];
    for (int i = 0; i < 4; i++) // convert int into 4 bytes
        bytes[i] = (byte)((n >>> 8*i) & 0xFF);
    int result = 0;
    for (int i = 0; i < 4; i++) {
        result += reverseByte(bytes[i]); // reverse per byte
        if (i < 3)
            result <<= 8;
    }
    return result;
}

private int reverseByte(byte b) {
    Integer value = cache.get(b); // first look up from cache
    if (value != null)
        return value;
    value = 0;
    // reverse by bit
    for (int i = 0; i < 8; i++) {
        value += ((b >>> i) & 1);
        if (i < 7)
            value <<= 1;
    }
    cache.put(b, value);
    return value;
}
```

written by AlexTheGreat original link here

Answer 3

```java
uint32_t reverseBits(uint32_t n) {
    uint32_t m = 0;
    for (int i = 0; i < 32; i++, n >>= 1) {
        m <<= 1;
        m |= n & 1;
    }
    return m;
}
```

The process is straightforward, just iterate over all bits.

written by xcv58 original link here

Number of 1 Bits(191)

Answer 1

```
public static int hammingWeight(int n) {
        int ones = 0;
        while(n!=0) {
            ones = ones + (n & 1);
            n = n>>>1;
        }
        return ones;
    }
```

- An Integer in Java has 32 bits, e.g. 00101000011110010100001000011010.
- To count the 1s in the Integer representation with put the input int n in bit AND with 1 (that is represented as 00000000000000000000000000000001, and if this operation result is 1, that means that the last bit of the input integer is 1. Thus we add it to the 1s count.

> ones = ones + (n & 1);

- Then we shift the input Integer by one on the right, to check for the next bit.

> n = n>>>1;

We need to use bit shifting unsigned operation >>> (while >> depends on sign extension)

- We keep doing this until the input Integer is 0.

In Java we need to put attention on the fact that the maximum integer is 2147483647. Integer type in Java is signed and there is no unsigned int. So the input 2147483648 is represented in Java as -2147483648 (in java int type has a cyclic representation, that means **Integer.MAX*VALUE+1==Integer.MIN*VALUE**). This force us to use

> n!=0

in the while condition and we cannot use

> n>0

because the input 2147483648 would correspond to -2147483648 in java and the code would not enter the while if the condition is n>0 for n=2147483648.

written by fabrizio3 original link here

Answer 2

Each time of "n &= n - 1", we delete one '1' from n.

```
int hammingWeight(uint32_t n)
{
    int res = 0;
    while(n)
    {
        n &= n - 1;
        ++ res;
    }
    return res;
}
```

Another several method of O(1) time.

Add 1 by Tree:

```
// This is a naive implementation, shown for comparison, and to help in understand
ing the better functions.
// It uses 24 arithmetic operations (shift, add, and).
int hammingWeight(uint32_t n)
{
    n = (n & 0x55555555) + (n >>  1 & 0x55555555); // put count of each  2 bits i
nto those  2 bits
    n = (n & 0x33333333) + (n >>  2 & 0x33333333); // put count of each  4 bits i
nto those  4 bits
    n = (n & 0x0F0F0F0F) + (n >>  4 & 0x0F0F0F0F); // put count of each  8 bits i
nto those  8 bits
    n = (n & 0x00FF00FF) + (n >>  8 & 0x00FF00FF); // put count of each 16 bits i
nto those 16 bits
    n = (n & 0x0000FFFF) + (n >> 16 & 0x0000FFFF); // put count of each 32 bits i
nto those 32 bits
    return n;
}

// This uses fewer arithmetic operations than any other known implementation on ma
chines with slow multiplication.
// It uses 17 arithmetic operations.
int hammingWeight(uint32_t n)
{
    n -= (n >> 1) & 0x55555555; //put count of each 2 bits into those 2 bits
    n = (n & 0x33333333) + (n >> 2 & 0x33333333); //put count of each 4 bits into
those 4 bits
    n = (n + (n >> 4)) & 0x0F0F0F0F; //put count of each 8 bits into those 8 bits
    n += n >> 8; // put count of each 16 bits into those 8 bits
    n += n >> 16; // put count of each 32 bits into those 8 bits
    return n & 0xFF;
}

// This uses fewer arithmetic operations than any other known implementation on ma
chines with fast multiplication.
// It uses 12 arithmetic operations, one of which is a multiply.
int hammingWeight(uint32_t n)
{
    n -= (n >> 1) & 0x55555555; // put count of each 2 bits into those 2 bits
    n = (n & 0x33333333) + (n >> 2 & 0x33333333); // put count of each 4 bits int
o those 4 bits
    n = (n + (n >> 4)) & 0x0F0F0F0F; // put count of each 8 bits into those 8 bit
s
    return n * 0x01010101 >> 24; // returns left 8 bits of x + (x<<8) + (x<<16) +
(x<<24)
}
```

â€”â€”From Wikipedia.

written by makuiyu original link here

Answer 3

```c
int hammingWeight(uint32_t n) {
    int count = 0;

    while (n) {
        n &= (n - 1);
        count++;
    }

    return count;
}
```

n & (n - 1) drops the lowest set bit. It's a neat little bit trick.

Let's use n = 00101100 as an example. This binary representation has three 1s.

If n = 00101100, then n - 1 = 00101011, so n & (n - 1) = 00101100 & 00101011 = 00101000. Count = 1.

If n = 00101000, then n - 1 = 00100111, so n & (n - 1) = 00101000 & 00100111 = 00100000. Count = 2.

If n = 00100000, then n - 1 = 00011111, so n & (n - 1) = 00100000 & 00011111 = 00000000. Count = 3.

n is now zero, so the while loop ends, and the final count (the numbers of set bits) is returned.

written by housed original link here

House Robber(198)

Answer 1

```
#define max(a, b) ((a)>(b)?(a):(b))
int rob(int num[], int n) {
    int a = 0;
    int b = 0;

    for (int i=0; i<n; i++)
    {
        if (i%2==0)
        {
            a = max(a+num[i], b);
        }
        else
        {
            b = max(a, b+num[i]);
        }
    }

    return max(a, b);
}
```

written by 452750465%40qq.com original link here

Answer 2

```
public int rob(int[] num) {
    int[][] dp = new int[num.length + 1][2];
    for (int i = 1; i <= num.length; i++) {
        dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
        dp[i][1] = num[i - 1] + dp[i - 1][0];
    }
    return Math.max(dp[num.length][0], dp[num.length][1]);
}
```

dp[i][1] means we rob the current house and dp[i][0] means we don't,

so it is easy to convert this to O(1) space

```
public int rob(int[] num) {
    int prevNo = 0;
    int prevYes = 0;
    for (int n : num) {
        int temp = prevNo;
        prevNo = Math.max(prevNo, prevYes);
        prevYes = n + temp;
    }
    return Math.max(prevNo, prevYes);
}
```

written by tusizi original link here

Answer 3

```
public class Solution {

    public int rob(int[] num) {
        int last = 0;
        int now = 0;
        int tmp;
        for (int n :num) {
            tmp = now;
            now = Math.max(last + n, now);
            last = tmp;
        }
        return now;
    }
}
```

written by wildhusky original link here

Binary Tree Right Side View(199)

Answer 1

The core idea of this algorithm:

1.Each depth of the tree only select one node.
2. View depth is current size of result list.

Here is the code:

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        rightView(root, result, 0);
        return result;
    }

    public void rightView(TreeNode curr, List<Integer> result, int currDepth){
        if(curr == null){
            return;
        }
        if(currDepth == result.size()){
            result.add(curr.val);
        }

        rightView(curr.right, result, currDepth + 1);
        rightView(curr.left, result, currDepth + 1);

    }
}
```

written by zwangbo original link here

Answer 2

```
class Solution {
public:
    void recursion(TreeNode *root, int level, vector<int> &res)
    {
        if(root==NULL) return ;
        if(res.size()<level) res.push_back(root->val);
        recursion(root->right, level+1, res);
        recursion(root->left, level+1, res);
    }

    vector<int> rightSideView(TreeNode *root) {
        vector<int> res;
        recursion(root, 1, res);
        return res;
    }
};
```

written by gavin5 original link here

Answer 3

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        // reverse level traversal
        List<Integer> result = new ArrayList();
        Queue<TreeNode> queue = new LinkedList();
        if (root == null) return result;

        queue.offer(root);
        while (queue.size() != 0) {
            int size = queue.size();
            for (int i=0; i<size; i++) {
                TreeNode cur = queue.poll();
                if (i == 0) result.add(cur.val);
                if (cur.right != null) queue.offer(cur.right);
                if (cur.left != null) queue.offer(cur.left);
            }

        }
        return result;
    }
}
```

written by jiayu.liu.961 original link here

Number of Islands(200)

Answer 1

```
public class NumberofIslands {
    static int[] dx = {-1,0,0,1};
    static int[] dy = {0,1,-1,0};
    public static int numIslands(char[][] grid) {
        if(grid==null || grid.length==0) return 0;
        int islands = 0;
        for(int i=0;i<grid.length;i++) {
            for(int j=0;j<grid[i].length;j++) {
                if(grid[i][j]=='1') {
                    explore(grid,i,j);
                    islands++;
                }
            }
        }
        return islands;
    }
    public static void explore(char[][] grid, int i, int j) {
        grid[i][j]='x';
        for(int d=0;d<dx.length;d++) {
            if(i+dy[d]<grid.length && i+dy[d]>=0 && j+dx[d]<grid[0].length && j+d
x[d]>=0 && grid[i+dy[d]][j+dx[d]]=='1') {
                explore(grid,i+dy[d],j+dx[d]);
            }
        }
    }
}
```

The algorithm works as follow:

1. Scan each cell in the grid.
2. If the cell value is '1' explore that island.
3. Mark the explored island cells with 'x'.
4. Once finished exploring that island, increment islands counter.

The arrays dx[], dy[] store the possible moves from the current cell. Two land cells ['1'] are considered from the same island if they are horizontally or vertically adjacent (possible moves (-1,0),(0,1),(0,-1),(1,0)). Two '1' diagonally adjacent are not considered from the same island.

written by fabrizio3 original link here

Answer 2

```
public class Solution {

private int n;
private int m;

public int numIslands(char[][] grid) {
    int count = 0;
    n = grid.length;
    if (n == 0) return 0;
    m = grid[0].length;
    for (int i = 0; i < n; i++){
        for (int j = 0; j < m; j++)
            if (grid[i][j] == '1') {
                DFSMarking(grid, i, j);
                ++count;
            }
    }
    return count;
}

private void DFSMarking(char[][] grid, int i, int j) {
    if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') return;
    grid[i][j] = '0';
    DFSMarking(grid, i + 1, j);
    DFSMarking(grid, i - 1, j);
    DFSMarking(grid, i, j + 1);
    DFSMarking(grid, i, j - 1);
}
```

}

written by wcyz666 original link here

Answer 3

Sink and count the islands.

Python Solution

```
def numIslands(self, grid):
    def sink(i, j):
        if 0 <= i < len(grid) and 0 <= j < len(grid[i]) and grid[i][j] == '1':
            grid[i][j] = '0'
            map(sink, (i+1, i-1, i, i), (j, j, j+1, j-1))
            return 1
        return 0
    return sum(sink(i, j) for i in range(len(grid)) for j in range(len(grid[i])))
```

Java Solution 1

```java
public class Solution {
    char[][] g;
    public int numIslands(char[][] grid) {
        int islands = 0;
        g = grid;
        for (int i=0; i<g.length; i++)
            for (int j=0; j<g[i].length; j++)
                islands += sink(i, j);
        return islands;
    }
    int sink(int i, int j) {
        if (i < 0 || i == g.length || j < 0 || j == g[i].length || g[i][j] == '0'
)
            return 0;
        g[i][j] = '0';
        sink(i+1, j); sink(i-1, j); sink(i, j+1); sink(i, j-1);
        return 1;
    }
}
```

Java Solution 2

```java
public class Solution {
    public int numIslands(char[][] grid) {
        int islands = 0;
        for (int i=0; i<grid.length; i++)
            for (int j=0; j<grid[i].length; j++)
                islands += sink(grid, i, j);
        return islands;
    }
    int sink(char[][] grid, int i, int j) {
        if (i < 0 || i == grid.length || j < 0 || j == grid[i].length || grid[i][
j] == '0')
            return 0;
        grid[i][j] = '0';
        for (int k=0; k<4; k++)
            sink(grid, i+d[k], j+d[k+1]);
        return 1;
    }
    int[] d = {0, 1, 0, -1, 0};
}
```

written by StefanPochmann original link here

Bitwise AND of Numbers Range(201)

Answer 1

The idea is very simple:

1. last bit of (odd number & even number) is 0.
2. when m != n, There is at least an odd number and an even number, so the last bit position result is 0.
3. Move m and n rigth a position.

Keep doing step 1,2,3 until m equal to n, use a factor to record the iteration time.

```
public class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        if(m == 0){
            return 0;
        }
        int moveFactor = 1;
        while(m != n){
            m >>= 1;
            n >>= 1;
            moveFactor <<= 1;
        }
        return m * moveFactor;
    }
}
```

written by zwangbo original link here

Answer 2

The idea is to use a mask to find the leftmost common digits of m and n. Example: m=1110001, n=1110111, and you just need to find 1110000 and it will be the answer.

```
public class Solution {
public int rangeBitwiseAnd(int m, int n) {
    int r=Integer.MAX_VALUE;
    while((m&r)!=(n&r))  r=r<<1;
    return n&r;
}
```

}

written by yu14 original link here

Answer 3

Consider the bits from low to high. if n > m, the lowest bit will be 0, and then we could transfer the problem to sub-problem: rangeBitwiseAnd(m>>1, n>>1).

```
int rangeBitwiseAnd(int m, int n) {
    return (n > m) ? (rangeBitwiseAnd(m/2, n/2) << 1) : m;
}
```

written by applewolf original link here

Happy Number(202)

Answer 1

I see the majority of those posts use hashset to record values. Actually, we can simply adapt the Floyd Cycle detection algorithm. I believe that many people have seen this in the Linked List Cycle detection problem. The following is my code:

```
int digitSquareSum(int n) {
    int sum = 0, tmp;
    while (n) {
        tmp = n % 10;
        sum += tmp * tmp;
        n /= 10;
    }
    return sum;
}

bool isHappy(int n) {
    int slow, fast;
    slow = fast = n;
    do {
        slow = digitSquareSum(slow);
        fast = digitSquareSum(fast);
        fast = digitSquareSum(fast);
    } while(slow != fast);
    if (slow == 1) return 1;
    else return 0;
}
```

written by Freezen original link here

Answer 2

```
public class Solution {
    public boolean isHappy(int n) {
        int x = n;
        int y = n;
        while(x>1){
            x = cal(x) ;
            if(x==1) return true ;
            y = cal(cal(y));
            if(y==1) return true ;

            if(x==y) return false;
        }
        return true ;
    }
    public int cal(int n){
        int x = n;
        int s = 0;
        while(x>0){
            s = s+(x%10)*(x%10);
            x = x/10;
        }
        return s ;
    }
}
```

written by ibmtp380 original link here

Answer 3

The idea is to use one hash set to record sum of every digit square of every number occurred. Once the current sum cannot be added to set, return false; once the current sum equals 1, return true;

```
public boolean isHappy(int n) {
    Set<Integer> inLoop = new HashSet<Integer>();
    int squareSum,remain;
    while (inLoop.add(n)) {
        squareSum = 0;
        while (n > 0) {
            remain = n%10;
            squareSum += remain*remain;
            n /= 10;
        }
        if (squareSum == 1)
            return true;
        else
            n = squareSum;

    }
    return false;

}
```

written by mo10 original link here

Remove Linked List Elements(203)

Answer 1

```
public ListNode removeElements(ListNode head, int val) {
        if (head == null) return null;
        head.next = removeElements(head.next, val);
        return head.val == val ? head.next : head;
}
```

written by renzid original link here

Answer 2

```
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode fakeHead = new ListNode(-1);
        fakeHead.next = head;
        ListNode curr = head, prev = fakeHead;
        while (curr != null) {
            if (curr.val == val) {
                prev.next = curr.next;
            } else {
                prev = prev.next;
            }
            curr = curr.next;
        }
        return fakeHead.next;
    }
}
```

written by lx223 original link here

Answer 3

Hi guys!

Here's an iterative solution without dummy head. First, we shift a head of a list while its' value equals to val. Then, we iterate through the nodes of the list checking if the next node's value equals to val and removing it if needed.

```
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        while (head != null && head.val == val) head = head.next;
        ListNode curr = head;
        while (curr != null && curr.next != null)
            if (curr.next.val == val) curr.next = curr.next.next;
            else curr = curr.next;
        return head;
    }
}
```

written by shpolsky original link here

Count Primes(204)

Answer 1

```
int countPrimes(int n) {
    if (n<=2) return 0;
    vector<bool> passed(n, false);
    int sum = 1;
    int upper = sqrt(n);
    for (int i=3; i<n; i+=2) {
        if (!passed[i]) {
            sum++;
            //avoid overflow
            if (i>upper) continue;
            for (int j=i*i; j<n; j+=i) {
                passed[j] = true;
            }
        }
    }
    return sum;
}
```

written by lester_zhang original link here

Answer 2

```
    /*1. trick1 is to use square root of n.
     2.  trick2 is not to use non-prime numbers as the step
     3. trick3 is to use i*i as the start.
     4. trick4 is to use count-- in every loop, avoiding another traversal. */
int countPrimes(int n) {
    if(n <= 2) return 0;
    if(n == 3) return 1;
    bool *prime= (bool*)malloc(sizeof(bool)*n);
    int i=0,j=0;
    int count = n-2;
    int rt = sqrt(n);//trick1
    for(j = 0; j < n; j++)
    {
        prime[j] = 1;
    }
    for(i = 2; i <= rt; i++)
    {
         if (prime[i])//trick2
        {
            for(j=i*i ; j<n ; j+=i)//trick3
            {
                if (prime[j])
                        {
                           prime[j]=0;
                           count--;//trick4
                        }
            }
        }
    }
    free(prime);
    return count;
}
```

written by scimagian original link here

Answer 3

```
class Solution {
public:
    int countPrimes(int n) {
        vector<bool> prime(n, true);
        prime[0] = false, prime[1] = false;
        for (int i = 0; i < sqrt(n); ++i) {
            if (prime[i]) {
                for (int j = i*i; j < n; j += i) {
                    prime[j] = false;
                }
            }
        }
        return count(prime.begin(), prime.end(), true);
    }
};
```

written by deck original link here

Isomorphic Strings(205)

Answer 1

```
class Solution {
public:
    bool isIsomorphic(string s, string t) {
        int m1[256] = {0}, m2[256] = {0}, n = s.size();
        for (int i = 0; i < n; ++i) {
            if (m1[s[i]] != m2[t[i]]) return false;
            m1[s[i]] = i + 1;
            m2[t[i]] = i + 1;
        }
        return true;
    }
};
```

written by grandyang original link here

Answer 2

Hi guys!

The main idea is to store the last seen positions of current (i-th) characters in both strings. If previously stored positions are different then we know that the fact they're occuring in the current i-th position simultaneously is a mistake. We could use a map for storing but as we deal with chars which are basically ints and can be used as indices we can do the whole thing with an array.

Check the code below. Happy coding!

```
public class Solution {
    public boolean isIsomorphic(String s1, String s2) {
        int[] m = new int[512];
        for (int i = 0; i < s1.length(); i++) {
            if (m[s1.charAt(i)] != m[s2.charAt(i)+256]) return false;
            m[s1.charAt(i)] = m[s2.charAt(i)+256] = i+1;
        }
        return true;
    }
}
```

written by shpolsky original link here

Answer 3

```
bool isIsomorphic(char* s, char* t) {
    char charArrS[256] = { 0 };
    char charArrT[256] = { 0 };
    int i = 0;
    while (s[i] !=0)
    {
        if (charArrS[s[i]] == 0 && charArrT[t[i]] == 0)
        {
            charArrS[s[i]] = t[i];
            charArrT[t[i]] = s[i];
        }
        else
        if (charArrS[s[i]] != t[i] || charArrT[t[i]] != s[i])
            return false;
        i++;
    }

    return true;
}
```

written by rxm24217 original link here

Reverse Linked List(206)

Answer 1

```
// iterative solution

public ListNode reverseList(ListNode head) {
    ListNode newHead = null;
    while(head != null){
        ListNode next = head.next;
        head.next = newHead;
        newHead = head;
        head = next;
    }
    return newHead;
}
```

// recursive solution

```
public ListNode reverseList(ListNode head) {
    return reverseListInt(head, null);
}

public ListNode reverseListInt(ListNode head, ListNode newHead) {
    if(head == null)
        return newHead;
    ListNode next = head.next;
    head.next = newHead;
    return reverseListInt(next, head);
}
```

written by braydenCN original link here

Answer 2

xWell, since the `head` pointer may also be modified, we create a `new_head` that points to it to facilitate the reverse process.

For the example list `1 -> 2 -> 3 -> 4 -> 5` in the problem statement, it will become `0 -> 1 -> 2 -> 3 -> 4 -> 5` (we init `new_head -> val` to be `0`). Then we set a pointer `pre` to `new_head` and another `cur` to `head`. Then we keep inserting `cur -> next` after `pre` until `cur` becomes the last node. The code is follows.

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* new_head = new ListNode(0);
        new_head -> next = head;
        ListNode* pre = new_head;
        ListNode* cur = head;
        while (cur && cur -> next) {
            ListNode* temp = pre -> next;
            pre -> next = cur -> next;
            cur -> next = cur -> next -> next;
            pre -> next -> next = temp;
        }
        return new_head -> next;
    }
};
```

This link provides a more concise solution without using the `new_head`. The idea is to reverse one node at a time for the beginning of the list. The rewritten code is as follows.

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = NULL;
        while (head) {
            ListNode* next = head -> next;
            head -> next = pre;
            pre = head;
            head = next;
        }
        return pre;
    }
};
```

Well, both of the above solutions are iterative. The hint has also suggested us to use recursion. In fact, the above link has a nice recursive solution, whose rewritten code is as follows.

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (!head || !(head -> next)) return head;
        ListNode* node = reverseList(head -> next);
        head -> next -> next = head;
        head -> next = NULL;
        return node;
    }
};
```

The basic idea of this recursive solution is to reverse all the following nodes after `head`. Then we need to set `head` to be the final node in the reversed list. We simply

set its next node in the original list (`head -> next`) to point to it and sets its `next` to be `NULL` .

written by jianchao.li.fighter original link here

Answer 3

```
struct ListNode* reverseList(struct ListNode* head) {
    if(NULL==head) return head;

    struct ListNode *p=head;
    p=head->next;
    head->next=NULL;
    while(NULL!=p){
        struct ListNode *ptmp=p->next;
        p->next=head;
        head=p;
        p=ptmp;
    }
    return head;
}
```

above is the iterative one. simple, nothing to explain.

```
struct ListNode* reverseListRe(struct ListNode* head) {
    if(NULL==head||NULL==head->next) return head;

    struct ListNode *p=head->next;
    head->next=NULL;
    struct ListNode *newhead=reverseListRe(p);
    p->next=head;

    return newhead;
}
```

above is the recursively one.Both are accepted.

written by redace85 original link here

Course Schedule(207)

Answer 1

1. BFS(Topological Sort)

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
{
    vector<unordered_set<int>> matrix(numCourses); // save this directed graph
    for(int i = 0; i < prerequisites.size(); ++ i)
        matrix[prerequisites[i][1]].insert(prerequisites[i][0]);

    vector<int> d(numCourses, 0); // in-degree
    for(int i = 0; i < numCourses; ++ i)
        for(auto it = matrix[i].begin(); it != matrix[i].end(); ++ it)
            ++ d[*it];

    for(int j = 0, i; j < numCourses; ++ j)
    {
        for(i = 0; i < numCourses && d[i] != 0; ++ i); // find a node whose in-de
gree is 0

        if(i == numCourses) // if not find
            return false;

        d[i] = -1;
        for(auto it = matrix[i].begin(); it != matrix[i].end(); ++ it)
            -- d[*it];
    }

    return true;
}
```

2. DFS(Finding cycle)

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
{
    vector<unordered_set<int>> matrix(numCourses); // save this directed graph
    for(int i = 0; i < prerequisites.size(); ++ i)
        matrix[prerequisites[i][1]].insert(prerequisites[i][0]);

    unordered_set<int> visited;
    vector<bool> flag(numCourses, false);
    for(int i = 0; i < numCourses; ++ i)
        if(!flag[i])
            if(DFS(matrix, visited, i, flag))
                return false;
    return true;
}
bool DFS(vector<unordered_set<int>> &matrix, unordered_set<int> &visited, int b,
vector<bool> &flag)
{
    flag[b] = true;
    visited.insert(b);
    for(auto it = matrix[b].begin(); it != matrix[b].end(); ++ it)
        if(visited.find(*it) != visited.end() || DFS(matrix, visited, *it, flag))
            return true;
    visited.erase(b);
    return false;
}
```

written by makuiyu original link here

Answer 2

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    int[][] matrix = new int[numCourses][numCourses]; // i -> j
    int[] indegree = new int[numCourses];

    for (int i=0; i<prerequisites.length; i++) {
        int ready = prerequisites[i][0];
        int pre = prerequisites[i][1];
        if (matrix[pre][ready] == 0)
            indegree[ready]++; //duplicate case
        matrix[pre][ready] = 1;
    }

    int count = 0;
    Queue<Integer> queue = new LinkedList();
    for (int i=0; i<indegree.length; i++) {
        if (indegree[i] == 0) queue.offer(i);
    }
    while (!queue.isEmpty()) {
        int course = queue.poll();
        count++;
        for (int i=0; i<numCourses; i++) {
            if (matrix[course][i] != 0) {
                if (--indegree[i] == 0)
                    queue.offer(i);
            }
        }
    }
    return count == numCourses;
}
```

written by jiayu.liu.961 original link here

Answer 3

As suggested by the hints, this problem is equivalent to detecting a cycle in the graph represented by `prerequisites`. Both BFS and DFS can be used to solve it using the idea of **topological sort**. If you find yourself unfamiliar with these concepts, you may refer to their wikipedia pages. Specifically, you may only need to refer to the link in the third hint to solve this problem.

Since `pair<int, int>` is inconvenient for the implementation of graph algorithms, we first transform it to a graph. If course `u` is a prerequisite of course `v`, we will add a directed edge from node `u` to node `v`.

BFS

BFS uses the indegrees of each node. We will first try to find a node with `0` indegree. If we fail to do so, there must be a cycle in the graph and we return `false`. Otherwise we have found one. We set its indegree to be `-1` to prevent from visiting it again and reduce the indegrees of all its neighbors by `1`. This process will be repeated for `n` (number of nodes) times. If we have not returned `false`, we will

return `true`.

```
class Solution {
public:
    bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> degrees = compute_indegree(graph);
        for (int i = 0; i < numCourses; i++) {
            int j = 0;
            for (; j < numCourses; j++)
                if (!degrees[j]) break;
            if (j == numCourses) return false;
            degrees[j] = -1;
            for (int neigh : graph[j])
                degrees[neigh]--;
        }
        return true;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    vector<int> compute_indegree(vector<unordered_set<int>>& graph) {
        vector<int> degrees(graph.size(), 0);
        for (auto neighbors : graph)
            for (int neigh : neighbors)
                degrees[neigh]++;
        return degrees;
    }
};
```

DFS

For DFS, it will first visit a node, then one neighbor of it, then one neighbor of this neighbor... and so on. If it meets a node which was visited in the current process of DFS visit, a cycle is detected and we will return `false`. Otherwise it will start from another unvisited node and repeat this process till all the nodes have been visited. Note that you should make two records: one is to record all the visited nodes and the other is to record the visited nodes in the current DFS visit.

The code is as follows. We use a `vector<bool> visited` to record all the visited nodes and another `vector<bool> onpath` to record the visited nodes of the current DFS visit. Once the current visit is finished, we reset the `onpath` value of the starting node to `false`.

```
class Solution {
public:
    bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<bool> onpath(numCourses, false), visited(numCourses, false);
        for (int i = 0; i < numCourses; i++)
            if (!visited[i] && dfs_cycle(graph, i, onpath, visited))
                return false;
        return true;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    bool dfs_cycle(vector<unordered_set<int>>& graph, int node, vector<bool>& onp
ath, vector<bool>& visited) {
        if (visited[node]) return false;
        onpath[node] = visited[node] = true;
        for (int neigh : graph[node])
            if (onpath[neigh] || dfs_cycle(graph, neigh, onpath, visited))
                return true;
        return onpath[node] = false;
    }
};
```

written by jianchao.li.fighter original link here

Implement Trie (Prefix Tree)(208)

Answer 1

```
class TrieNode
{
public:
    TrieNode *next[26];
    bool is_word;

    // Initialize your data structure here.
    TrieNode(bool b = false)
    {
        memset(next, 0, sizeof(next));
        is_word = b;
    }
};

class Trie
{
    TrieNode *root;
public:
    Trie()
    {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    void insert(string s)
    {
        TrieNode *p = root;
        for(int i = 0; i < s.size(); ++ i)
        {
            if(p -> next[s[i] - 'a'] == NULL)
                p -> next[s[i] - 'a'] = new TrieNode();
            p = p -> next[s[i] - 'a'];
        }
        p -> is_word = true;
    }

    // Returns if the word is in the trie.
    bool search(string key)
    {
        TrieNode *p = find(key);
        return p != NULL && p -> is_word;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    bool startsWith(string prefix)
    {
        return find(prefix) != NULL;
    }

private:
    TrieNode* find(string key)
```

```
    {
        TrieNode *p = root;
        for(int i = 0; i < key.size() && p != NULL; ++ i)
            p = p -> next[key[i] - 'a'];
        return p;
    }
};
```

written by makuiyu original link here

Answer 2

```
    /**
     ** author: cxq
     ** weibo: http://weibo.com/chenxq1992
     **/

    class TrieNode {
    public:
        char content;   // the character included
        bool isend;     // if the node is the end of a word
        int shared;     // the number of the node shared ,convenient to implement
delete(string key), not necessary in this problem
        vector<TrieNode*> children; // the children of the node
        // Initialize your data structure here.
        TrieNode():content(' '), isend(false), shared(0) {}
        TrieNode(char ch):content(ch), isend(false), shared(0) {}
        TrieNode* subNode(char ch) {
            if (!children.empty()) {
                for (auto child : children) {
                    if (child->content == ch)
                        return child;
                }
            }
            return nullptr;
        }
        ~TrieNode() {
            for (auto child : children)
                delete child;
        }
    };

    class Trie {
    public:
        Trie() {
            root = new TrieNode();
        }

        // Inserts a word into the trie.
        void insert(string s) {
            if (search(s)) return;
            TrieNode* curr = root;
            for (auto ch : s) {
                TrieNode* child = curr->subNode(ch);
                if (child != nullptr) {
                    curr = child;
```

```
                curr = child;
            } else {
                TrieNode *newNode = new TrieNode(ch);
                curr->children.push_back(newNode);
                curr = newNode;
            }
            ++curr->shared;
        }
        curr->isend = true;
    }

    // Returns if the word is in the trie.
    bool search(string key) {
        TrieNode* curr = root;
        for (auto ch : key) {
            curr = curr->subNode(ch);
            if (curr == nullptr)
                return false;
        }
        return curr->isend == true;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    bool startsWith(string prefix) {
        TrieNode* curr = root;
        for (auto ch : prefix) {
            curr = curr->subNode(ch);
            if (curr == nullptr)
                return false;
        }
        return true;
    }
    ~Trie() {
        delete root;
    }
private:
    TrieNode* root;
};
```

written by cxq1992 original link here

Answer 3

Detailed explanation after code!

```
class TrieNode {
    public char val;
    public boolean isWord;
    public TrieNode[] children = new TrieNode[26];
    public TrieNode() {}
    TrieNode(char c){
        TrieNode node = new TrieNode();
        node.val = c;
    }
}

public class Trie {
    private TrieNode root;
    public Trie() {
        root = new TrieNode();
        root.val = ' ';
    }

    public void insert(String word) {
        TrieNode ws = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a'] == null){
                ws.children[c - 'a'] = new TrieNode(c);
            }
            ws = ws.children[c - 'a'];
        }
        ws.isWord = true;
    }

    public boolean search(String word) {
        TrieNode ws = root;
        for(int i = 0; i < word.length(); i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a'] == null) return false;
            ws = ws.children[c - 'a'];
        }
        return ws.isWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode ws = root;
        for(int i = 0; i < prefix.length(); i++){
            char c = prefix.charAt(i);
            if(ws.children[c - 'a'] == null) return false;
            ws = ws.children[c - 'a'];
        }
        return true;
    }
}
```

With my solution I took the simple approach of giving each TrieNode a 26 element array of each possible child node it may have. I only gave 26 children nodes because we are only working with lowercase 'a' - 'z'. If you are uncertain why I made the root

of my Trie an empty character this is a standard/typical approach for building out a Trie it is somewhat arbitrary what the root node is.

For insert I used the following algorithm. Loop through each character in the word being inserted check if the character is a child node of the current TrieNode i.e. check if the array has a populated value in the index of this character. If the current character ISN'T a child node of my current node add this character representation to the corresponding index location then set current node equal to the child that was added. However if the current character IS a child of the current node only set current node equal to the child. After evaluating the entire String the Node we left off on is marked as a *word* this allows our Trie to know which words exist in our "dictionary"

For search I simply navigate through the Trie if I discover the current character isn't in the Trie I return false. After checking each Char in the String I check to see if the Node I left off on was marked as a word returning the result.

Starts with is identical to search except it doesn't matter if the Node I left off was marked as a word or not if entire string evaluated i always return true;

written by mlblount45 original link here

Minimum Size Subarray Sum(209)

Answer 1

```
public class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        return solveNLogN(s, nums);
    }

    private int solveN(int s, int[] nums) {
        int start = 0, end = 0, sum = 0, minLen = Integer.MAX_VALUE;
        while (end < nums.length) {
            while (end < nums.length && sum < s) sum += nums[end++];
            if (sum < s) break;
            while (start < end && sum >= s) sum -= nums[start++];
            if (end - start + 1 < minLen) minLen = end - start + 1;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int solveNLogN(int s, int[] nums) {
        int[] sums = new int[nums.length + 1];
        for (int i = 1; i < sums.length; i++) sums[i] = sums[i - 1] + nums[i - 1]
;
        int minLen = Integer.MAX_VALUE;
        for (int i = 0; i < sums.length; i++) {
            int end = binarySearch(i + 1, sums.length - 1, sums[i] + s, sums);
            if (end == sums.length) break;
            if (end - i < minLen) minLen = end - i;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int binarySearch(int lo, int hi, int key, int[] sums) {
        while (lo <= hi) {
           int mid = (lo + hi) / 2;
           if (sums[mid] >= key){
               hi = mid - 1;
           } else {
               lo = mid + 1;
           }
        }
        return lo;
    }
}
```

Since the given array contains only positive integers, the subarray sum can only increase by including more elements. Therefore, you don't have to include more elements once the current subarray already has a sum large enough. This gives the linear time complexity solution by maintaining a minimum window with a two indices.

As to NLogN solution, logN immediately reminds you of binary search. In this case, you cannot sort as the current order actually matters. How does one get an ordered

array then? Since all elements are positive, the cumulative sum must be strictly increasing. Then, a subarray sum can expressed as the difference between two cumulative sum. Hence, given a start index for the cumulative sum array, the other end index can be searched using binary search.

written by lx223 original link here

Answer 2

The problem statement has stated that there are both `O(n)` and `O(nlogn)` solutions to this problem. Let's see the `O(n)` solution first (taken from this link), which is pretty clever and short.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size(), start = 0, sum = 0, minlen = INT_MAX;
        for (int i = 0; i < n; i++) {
            sum += nums[i];
            while (sum >= s) {
                minlen = min(minlen, i - start + 1);
                sum -= nums[start++];
            }
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
};
```

Well, you may wonder how can it be `O(n)` since it contains an inner `while` loop. Well, the key is that the `while` loop executes at most once for each starting position `start`. Then `start` is increased by `1` and the `while` loop moves to the next element. Thus the inner `while` loop runs at most `O(n)` times during the whole `for` loop from `0` to `n - 1`. Thus both the `for` loop and `while` loop has `O(n)` time complexity in total and the overall running time is `O(n)`.

There is another `O(n)` solution in this link, which is easier to understand and prove it is `O(n)`. I have rewritten it below.

```cpp
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size(), left = 0, right = 0, sum = 0, minlen = INT_MAX;
        while (right < n) {
            do sum += nums[right++];
            while (right < n && sum < s);
            while (left < right && sum - nums[left] >= s)
                sum -= nums[left++];
            if (sum >= s) minlen = min(minlen, right - left);
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
};
```

Now let's move on to the `O(nlogn)` solution. Well, this less efficient solution is far more difficult to come up with. The idea is to first maintain an array of accumulated summations of elements in `nums`. Specifically, for `nums = [2, 3, 1, 2, 4, 3]` in the problem statement, `sums = [0, 2, 5, 6, 8, 12, 15]`. Then for each element in `sums`, if it is not less than `s`, we search for the first element that is greater than `sums[i] - s` (in fact, this is just what the `upper_bound` function does) in `sums` using binary search.

Let's do an example. Suppose we reach `12` in `sums`, which is greater than `s = 7`. We then search for the first element in `sums` that is greater than `sums[i] - s = 12 - 7 = 5` and we find `6`. Then we know that the elements in `nums` that correspond to `6, 8, 12` sum to a number `12 - 5 = 7` which is not less than `s = 7`. Let's check for that: `6` in `sums` corresponds to `1` in `nums`, `8` in `sums` corresponds to `2` in `nums`, `12` in `sums` corresponds to `4` in `nums`. `1, 2, 4` sum to `7`, which is `12` in `sums` minus `5` in `sums`.

We add a `0` in the first position of `sums` to account for cases like `nums = [3], s = 3`.

The code is as follows.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        vector<int> sums = accumulate(nums);
        int n = nums.size(), minlen = INT_MAX;
        for (int i = 1; i <= n; i++) {
            if (sums[i] >= s) {
                int p = upper_bound(sums, 0, i, sums[i] - s);
                if (p != -1) minlen = min(minlen, i - p + 1);
            }
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
private:
    vector<int> accumulate(vector<int>& nums) {
        int n = nums.size();
        vector<int> sums(n + 1, 0);
        for (int i = 1; i <= n; i++)
            sums[i] = nums[i - 1] + sums[i - 1];
        return sums;
    }
    int upper_bound(vector<int>& sums, int left, int right, int target) {
        int l = left, r = right;
        while (l < r) {
            int m = l + ((r - l) >> 1);
            if (sums[m] <= target) l = m + 1;
            else r = m;
        }
        return sums[r] > target ? r : -1;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```
class Solution {
public:
int minSubArrayLen(int s, vector<int>& nums) {
    int firstPos = 0, sum = 0, minLength = INT_MAX;
    for(int i = 0; i<nums.size(); i++) {
        sum += nums[i];
        while(sum >= s) {
            minLength = min(minLength, i - firstPos + 1);
            sum -= nums[firstPos++];
        }
    }

    return minLength == INT_MAX? 0 : minLength;
  }
};
```

written by jaewoo original link here

Course Schedule II(210)

Answer 1

This question asks for an order in which prerequisite courses must be taken first. This prerequisite relationship reminds one of directed graphs. Then, the problem reduces to find a topological sort order of the courses, which would be a DAG if it has a valid order.

```
public int[] findOrder(int numCourses, int[][] prerequisites) {
    int[] incLinkCounts = new int[numCourses];
    List<List<Integer>> adjs = new ArrayList<>(numCourses);
    initialiseGraph(incLinkCounts, adjs, prerequisites);
    //return solveByBFS(incLinkCounts, adjs);
    return solveByDFS(adjs);
}
```

The first step is to transform it into a directed graph. Since it is likely to be sparse,we use adjacency list graph data structure. 1 -> 2 means 1 must be taken before 2.

```
private void initialiseGraph(int[] incLinkCounts, List<List<Integer>> adjs, int[]
[] prerequisites){
    int n = incLinkCounts.length;
    while (n-- > 0) adjs.add(new ArrayList<>());
    for (int[] edge : prerequisites) {
        incLinkCounts[edge[0]]++;
        adjs.get(edge[1]).add(edge[0]);
    }
}
```

How can we obtain a topological sort order of a DAG?

We observe that if a node has incoming edges, it has prerequisites. Therefore, the first few in the order must be those with no prerequisites, i.e. no incoming edges. Any non-empty DAG must have at least one node without incoming links. You can draw a small graph to convince yourself. If we visit these few and remove all edges attached to them, we are left with a smaller DAG, which is the same problem. This will then give our BFS solution.

```
private int[] solveByBFS(int[] incLinkCounts, List<List<Integer>> adjs){
    int[] order = new int[incLinkCounts.length];
    Queue<Integer> toVisit = new ArrayDeque<>();
    for (int i = 0; i < incLinkCounts.length; i++) {
        if (incLinkCounts[i] == 0) toVisit.offer(i);
    }
    int visited = 0;
    while (!toVisit.isEmpty()) {
        int from = toVisit.poll();
        order[visited++] = from;
        for (int to : adjs.get(from)) {
            incLinkCounts[to]--;
            if (incLinkCounts[to] == 0) toVisit.offer(to);
        }
    }
    return visited == incLinkCounts.length ? order : new int[0];
}
```

Another way to think about it is the last few in the order must be those which are not prerequisites of other courses. Thinking it recursively means if one node has unvisited child node, you should visit them first before you put this node down in the final order array. This sounds like the post-order of a DFS. Since we are putting nodes down in the reverse order, we should reverse it back to correct ordering or use a stack.

```
private int[] solveByDFS(List<List<Integer>> adjs) {
    BitSet hasCycle = new BitSet(1);
    BitSet visited = new BitSet(adjs.size());
    BitSet onStack = new BitSet(adjs.size());
    Deque<Integer> order = new ArrayDeque<>();
    for (int i = adjs.size() - 1; i >= 0; i--) {
        if (visited.get(i) == false && hasOrder(i, adjs, visited, onStack, order)
== false) return new int[0];
    }
    int[] orderArray = new int[adjs.size()];
    for (int i = 0; !order.isEmpty(); i++) orderArray[i] = order.pop();
    return orderArray;
}

private boolean hasOrder(int from, List<List<Integer>> adjs, BitSet visited, BitS
et onStack, Deque<Integer> order) {
    visited.set(from);
    onStack.set(from);
    for (int to : adjs.get(from)) {
        if (visited.get(to) == false) {
            if (hasOrder(to, adjs, visited, onStack, order) == false) return fals
e;
        } else if (onStack.get(to) == true) {
            return false;
        }
    }
    onStack.clear(from);
    order.push(from);
    return true;
}
```

written by lx223 original link here

Answer 2

Well, this problem is spiritually similar to to Course Schedule. You only need to store the nodes in the order you visit into a vector during BFS or DFS. Well, for DFS, a final reversal is required.

BFS

```cpp
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> degrees = compute_indegree(graph);
        queue<int> zeros;
        for (int i = 0; i < numCourses; i++)
            if (!degrees[i]) zeros.push(i);
        vector<int> toposort;
        for (int i = 0; i < numCourses; i++) {
            if (zeros.empty()) return {};
            int zero = zeros.front();
            zeros.pop();
            toposort.push_back(zero);
            for (int neigh : graph[zero]) {
                if (!--degrees[neigh])
                    zeros.push(neigh);
            }
        }
        return toposort;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    vector<int> compute_indegree(vector<unordered_set<int>>& graph) {
        vector<int> degrees(graph.size(), 0);
        for (auto neighbors : graph)
            for (int neigh : neighbors)
                degrees[neigh]++;
        return degrees;
    }
};
```

DFS

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites)
{
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> toposort;
        vector<bool> onpath(numCourses, false), visited(numCourses, false);
        for (int i = 0; i < numCourses; i++)
            if (!visited[i] && dfs(graph, i, onpath, visited, toposort))
                return {};
        reverse(toposort.begin(), toposort.end());
        return toposort;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    bool dfs(vector<unordered_set<int>>& graph, int node, vector<bool>& onpath, v
ector<bool>& visited, vector<int>& toposort) {
        if (visited[node]) return false;
        onpath[node] = visited[node] = true;
        for (int neigh : graph[node])
            if (onpath[neigh] || dfs(graph, neigh, onpath, visited, toposort))
                return true;
        toposort.push_back(node);
        return onpath[node] = false;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```java
public class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adj = new ArrayList<>(numCourses);
        for (int i = 0; i < numCourses; i++) adj.add(i, new ArrayList<>());
        for (int i = 0; i < prerequisites.length; i++) adj.get(prerequisites[i][1
]).add(prerequisites[i][0]);
        boolean[] visited = new boolean[numCourses];
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < numCourses; i++) {
            if (!topologicalSort(adj, i, stack, visited, new boolean[numCourses])
) return new int[0];
        }
        int i = 0;
        int[] result = new int[numCourses];
        while (!stack.isEmpty()) {
            result[i++] = stack.pop();
        }
        return result;
    }

    private boolean topologicalSort(List<List<Integer>> adj, int v, Stack<Integer
> stack, boolean[] visited, boolean[] isLoop) {
        if (visited[v]) return true;
        if (isLoop[v]) return false;
        isLoop[v] = true;
        for (Integer u : adj.get(v)) {
            if (!topologicalSort(adj, u, stack, visited, isLoop)) return false;
        }
        visited[v] = true;
        stack.push(v);
        return true;
    }
}
```

written by baibing original link here

Add and Search Word - Data structure design(211)

Answer 1

This problem is an application of the Trie data structure. In the following, it is assumed that you have solved Implement Trie (Prefix Tree).

Now, let's first look at the `TrieNode` class. I define it as follows.

```
class TrieNode {
public:
    bool isKey;
    TrieNode* children[26];
    TrieNode(): isKey(false) {
        memset(children, NULL, sizeof(TrieNode*) * 26);
    }
};
```

The field `isKey` is to label whether the string comprised of characters starting from `root` to the current node is a key (word that has been added). In this problem, only lower-case letters `a - z` need to be considered, so each `TrieNode` has at most `26` children. I store it in an array of `TrieNode*`: `children[i]` corresponds to letter `'a' + i`. The remaining code defines the constructor of the `TrieNode` class.

Adding a word can be done in the same way as inImplement Trie (Prefix Tree). The basic idea is to create a `TrieNode` corresponding to each letter in the word. When we are done, label the last node to be a key (set `isKey = true`). The code is as follows.

```
void addWord(string word) {
    TrieNode* run = root;
    for (char c : word) {
        if (!(run -> children[c - 'a']))
            run -> children[c - 'a'] = new TrieNode();
        run = run -> children[c - 'a'];
    }
    run -> isKey = true;
}
```

By the way, `root` is defined as private data of `WordDictionary`:

```
private:
    TrieNode* root;
```

And the `WordDictionary` class has a constructor to initialize `root`:

```
WordDictionary() {
    root = new TrieNode();
}
```

Now we are left only with `search`. Let's do it. The basic idea is still the same as typical search operations in a Trie. The critical part is how to deal with the dots `.`. Well, my solution is very naive in this place. Each time when we reach a `.`, just traverse all the children of the current node and recursively search the remaining substring in `word` starting from that children. So I define a helper function `query` for `search` that takes in a string and a starting node. And the initial call to `query` is like `query(word, root)`.

By the way, I pass a `char*` instead of `string` to `query` and it greatly speeds up the code. So the initial call to `query` is actually `query(word.c_str(), root)`.

Now I put all the codes together below. Hope it to be useful!

```
class TrieNode {
public:
    bool isKey;
    TrieNode* children[26];
    TrieNode(): isKey(false) {
        memset(children, NULL, sizeof(TrieNode*) * 26);
    }
};

class WordDictionary {
public:
    WordDictionary() {
        root = new TrieNode();
    }

    // Adds a word into the data structure.
    void addWord(string word) {
        TrieNode* run = root;
        for (char c : word) {
            if (!(run -> children[c - 'a']))
                run -> children[c - 'a'] = new TrieNode();
            run = run -> children[c - 'a'];
        }
        run -> isKey = true;
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    bool search(string word) {
        return query(word.c_str(), root);
    }

private:
    TrieNode* root;

    bool query(const char* word, TrieNode* node) {
        TrieNode* run = node;
        for (int i = 0; word[i]; i++) {
            if (run && word[i] != '.')
                run = run -> children[word[i] - 'a'];
            else if (run && word[i] == '.') {
                TrieNode* tmp = run;
```

```
                TrieNode* tmp = run;
                for (int j = 0; j < 26; j++) {
                    run = tmp -> children[j];
                    if (query(word + i + 1, run))
                        return true;
                }
            }
            else break;
        }
        return run && run -> isKey;
    }
};

// Your WordDictionary object will be instantiated and called as such:
// WordDictionary wordDictionary;
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

written by jianchao.li.fighter original link here

Answer 2

```
public class WordDictionary {
    WordNode root = new WordNode();
    public void addWord(String word) {
        char chars[] = word.toCharArray();
        addWord(chars, 0, root);
    }

    private void addWord(char[] chars, int index, WordNode parent) {
        char c = chars[index];
        int idx = c-'a';
        WordNode node = parent.children[idx];
        if (node == null){
            node = new WordNode();
            parent.children[idx]=node;
        }
        if (chars.length == index+1){
            node.isLeaf=true;
            return;
        }
        addWord(chars, ++index, node);
    }


    public boolean search(String word) {
        return search(word.toCharArray(), 0, root);
    }

    private boolean search(char[] chars, int index, WordNode parent){
        if (index == chars.length){
            if (parent.isLeaf){
                return true;
            }
            return false;
        }
```

```
        WordNode[] childNodes = parent.children;
        char c = chars[index];
        if (c == '.'){
            for (int i=0;i<childNodes.length;i++){
                WordNode n = childNodes[i];
                if (n !=null){
                    boolean b = search(chars, index+1, n);
                    if (b){
                        return true;
                    }
                }
            }
            return false;
        }
        WordNode node = childNodes[c-'a'];
        if (node == null){
            return false;
        }
        return search(chars, ++index, node);
    }



    private class WordNode{
        boolean isLeaf;
        WordNode[] children = new WordNode[26];
    }
}
```

written by qgambit2 original link here

Answer 3

```
class WordDictionary:

    def __init__(self):
        self.root = {}

    def addWord(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
        node[None] = None

    def search(self, word):
        def find(word, node):
            if not word:
                return None in node
            char, word = word[0], word[1:]
            if char != '.':
                return char in node and find(word, node[char])
            return any(find(word, kid) for kid in node.values() if kid)
        return find(word, self.root)
```

An iterative alternative for the `search` method:

```
    def search(self, word):
        nodes = [self.root]
        for char in word:
            nodes = [kid
                     for node in nodes
                     for key, kid in node.items()
                     if char in (key, '.') and kid]
        return any(None in node for node in nodes)
```

And one that's a bit longer but faster:

```
    def search(self, word):
        nodes = [self.root]
        for char in word:
            nodes = [kid for node in nodes for kid in
                     ([node[char]] if char in node else
                      filter(None, node.values()) if char == '.' else [])]
        return any(None in node for node in nodes)
```

And a neat version where I append my end-marker to the word to simplify the final check:

```
class WordDictionary:

    def __init__(self):
        self.root = {}

    def addWord(self, word):
        node = self.root
        for char in word:
            node = node.setdefault(char, {})
        node['$'] = None

    def search(self, word):
        nodes = [self.root]
        for char in word + '$':
            nodes = [kid for node in nodes for kid in
                     ([node[char]] if char in node else
                      filter(None, node.values()) if char == '.' else [])]
        return bool(nodes)
```

written by StefanPochmann original link here

Word Search II(212)

Answer 1

Compared with Word Search, I make my DFS with a tire but a word. The Trie is formed by all the words in given *words*. Then during the DFS, for each current formed word, I check if it is in the Trie.

```
public class Solution {
    Set<String> res = new HashSet<String>();

    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }

        int m = board.length;
        int n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, visited, "", i, j, trie);
            }
        }

        return new ArrayList<String>(res);
    }

    public void dfs(char[][] board, boolean[][] visited, String str, int x, int y
, Trie trie) {
        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length) return;
        if (visited[x][y]) return;

        str += board[x][y];
        if (!trie.startsWith(str)) return;

        if (trie.search(str)) {
            res.add(str);
        }

        visited[x][y] = true;
        dfs(board, visited, str, x - 1, y, trie);
        dfs(board, visited, str, x + 1, y, trie);
        dfs(board, visited, str, x, y - 1, trie);
        dfs(board, visited, str, x, y + 1, trie);
        visited[x][y] = false;
    }
}
```

written by Lnic original link here

Answer 2

The idea is to use a Trie to build a prefix tree for words to simplify the search and do

DFS to search all the possible strings. For Trie, 26 pointers to point the sub-strings and a bool leaf to indicate whether the current node is a leaf (i.e. a string in words) and also idx is used to save the index of words for the current node. For DFS, just check if the current position is visited before (board[i][j]=='X'), if so, return, check if there is a string with such prefix (nullptr == root->children[words[idx][pos]-'a']), if not, return; otherwise, check if the current searched string is a leaf of the trie (a string in words), if so, save it to res and set leaf of the trie node to false to indicate such string is already found. At last, move to its neighbors to continue the search. Remember to recover the char [i][j] at the end.

```
    class Solution {
        class Trie{
        public:
            Trie *children[26]; // pointers to its substrings starting with 'a' t
o 'z'
            bool leaf; // if the node is a leaf, or if there is a word stopping a
t here
            int idx; // if it is a leaf, the string index of the array words
            Trie()
            {
                this->leaf = false;
                this->idx = 0;
                fill_n(this->children, 26, nullptr);
            }
        };

    public:
        void insertWords(Trie *root, vector<string>& words, int idx)
        {
            int pos = 0, len = words[idx].size();
            while(pos<len)
            {
                if(nullptr == root->children[words[idx][pos]-'a']) root->children
[words[idx][pos]-'a'] = new Trie();
                root = root->children[words[idx][pos++]-'a'];
            }
            root->leaf = true;
            root->idx = idx;
        }

        Trie *buildTrie(vector<string>& words)
        {
            Trie *root = new Trie();
            int i;
            for(i=0; i<words.size();i++) insertWords(root, words, i);
            return root;
        }

        void checkWords(vector<vector<char>>& board, int i, int j, int row, int co
l, Trie *root, vector<string> &res, vector<string>& words)
        {
            char temp;
            if(board[i][j]=='X') return; // visited before;
            if(nullptr == root->children[board[i][j]-'a']) return ; // no string
with such prefix
```

```
with such prefix
            else
            {
                temp = board[i][j];
                if(root->children[temp-'a']->leaf)  // if it is a leaf
                {
                    res.push_back(words[root->children[temp-'a']->idx]);
                    root->children[temp-'a']->leaf = false; // set to false to in
dicate that we found it already
                }
                board[i][j]='X'; //mark the current position as visited
// check all the possible neighbors
                if(i>0) checkWords(board, i-1, j, row, col, root->children[temp-'
a'], res, words);
                if((i+1)<row) checkWords(board, i+1, j, row, col,  root->children
[temp-'a'], res, words);
                if(j>0) checkWords(board, i, j-1,  row, col, root->children[temp-
'a'], res, words);
                if((j+1)<col)  checkWords(board, i, j+1,  row, col, root->childre
n[temp-'a'], res, words);
                board[i][j] = temp; // recover the current position
            }
        }

        vector<string> findWords(vector<vector<char>>& board, vector<string>& wor
ds) {
           vector<string> res;
           int row = board.size();
           if(0==row) return res;
           int col = board[0].size();
           if(0==col) return res;
           int wordCount = words.size();
           if(0==wordCount) return res;

           Trie *root = buildTrie(words);

           int i,j;
           for(i =0 ; i<row; i++)
           {
               for(j=0; j<col && wordCount > res.size(); j++)
               {
                   checkWords(board, i, j, row, col, root, res, words);
               }
           }
           return res;
        }
 };
```

Based on the comments received. I created another version with Trie node counter (thanks, zhiqing_xiao and gxyeecspku). However, for the current test set, it doesn't help too much. Anyway, my version with Trie node counter.

```
class Solution {
private:
class Trie
{
```

```
public:
    Trie * children[26];
    bool isLeaf;
    int  wordIdx;
    int prefixCount;

    Trie()
    {
        isLeaf = false;
        wordIdx = 0;
        prefixCount = 0;
        fill_n(children, 26, nullptr);
    }

    ~Trie()
    {
        for(auto i=0; i<26; ++i) delete children[i];
    }
};
    void insertWord(Trie *root,  const vector<string>& words, int idx)
    {
        int i, childID, len = words[idx].size();
        for(i=0, root->prefixCount++ ; i<len; ++i)
        {
            childID = words[idx][i]-'a';
            if(!root->children[childID]) root->children[childID] = new Trie();
            root = root->children[childID];
            ++root->prefixCount;
        }
        root->isLeaf = true;
        root->wordIdx = idx;
    }

    Trie *buildTrie(const vector<string> &words)
    {
        Trie *root = new Trie();
        for(int i=0; i < words.size(); ++i) insertWord(root, words, i);
        return root;
    }

    int dfs_Trie(vector<string> &res, Trie *root, vector<vector<char>>& board, ve
ctor<string>& words, int row, int col)
    {
        int detected = 0;

        if(root->isLeaf)
        {
            ++detected;
            root->isLeaf = false;
            res.push_back(words[root->wordIdx]);
        }

        if( row<0 || row>=board.size() || col<0 || col>=board[0].size() || board[
row][col]=='*' || !root->children[ board[row][col]-'a'] || root->children[ board[r
ow][col]-'a']->prefixCount <= 0 ) return detected;
        int curC = board[row][col] - 'a';
        board[row][col] = '*';
```

```
        board[row][col] = '*';
        detected += dfs_Trie(res, root->children[curC], board, words, row-1, col)
+
               dfs_Trie(res, root->children[curC], board, words, row+1, col) +

               dfs_Trie(res, root->children[curC], board, words, row, col - 1) +

               dfs_Trie(res, root->children[curC], board, words, row, col + 1) ;
        root->prefixCount -=detected;
        board[row][col] = curC+'a';
        return detected;
    }

public:
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words)
{
        int M, N, wordNum = words.size();
        vector<string> res;
        if( !(M = board.size()) || !(N = board[0].size()) || !wordNum) return res
;
        Trie *root = buildTrie(words);
        for(auto i=0; i<M && root->prefixCount; ++i)
            for(auto j=0; j<N; ++j)
                dfs_Trie(res, root, board, words, i, j);
        delete root;
        return res;
    }
};
```

written by dong.wang.1694 original link here

Answer 3

```
public class Solution {
    public class TrieNode{
        public boolean isWord = false;
        public TrieNode[] child = new TrieNode[26];
        public TrieNode(){

        }
    }

    TrieNode root = new TrieNode();
    boolean[][] flag;
    public List<String> findWords(char[][] board, String[] words) {
        Set<String> result = new HashSet<>();
        flag = new boolean[board.length][board[0].length];

        addToTrie(words);

        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(root.child[board[i][j] - 'a'] != null){
                    search(board, i, j, root, "", result);
                }
            }
```

```
        }

        return new LinkedList<>(result);
    }

    private void addToTrie(String[] words){
        for(String word: words){
            TrieNode node = root;
            for(int i = 0; i < word.length(); i++){
                char ch = word.charAt(i);
                if(node.child[ch - 'a'] == null){
                    node.child[ch - 'a'] = new TrieNode();
                }
                node = node.child[ch - 'a'];
            }
            node.isWord = true;
        }
    }

    private void search(char[][] board, int i, int j, TrieNode node, String word,
Set<String> result){
        if(i >= board.length || i < 0 || j >= board[i].length || j < 0 || flag[i]
[j]){
            return;
        }

        if(node.child[board[i][j] - 'a'] == null){
            return;
        }

        flag[i][j] = true;
        node = node.child[board[i][j] - 'a'];
        if(node.isWord){
            result.add(word + board[i][j]);
        }

        search(board, i-1, j, node, word + board[i][j], result);
        search(board, i+1, j, node, word + board[i][j], result);
        search(board, i, j-1, node, word + board[i][j], result);
        search(board, i, j+1, node, word + board[i][j], result);

        flag[i][j] = false;
    }
}
```

written by harish-v original link here

House Robber II(213)

Answer 1

Since this question is a follow-up to House Robber, we can assume we already have a way to solve the simpler question, i.e. given a 1 row of house, we know how to rob them. So we already have such a helper function. We modify it a bit to rob a given range of houses.

```
private int rob(int[] num, int lo, int hi) {
    int include = 0, exclude = 0;
    for (int j = lo; j <= hi; j++) {
        int i = include, e = exclude;
        include = e + num[j];
        exclude = Math.max(e, i);
    }
    return Math.max(include, exclude);
}
```

Now the question is how to rob a circular row of houses. It is a bit complicated to solve like the simpler question. It is because in the simpler question whether to rob *num[lo]* is entirely our choice. But, it is now constrained by whether*num[hi]* is robbed.

However, since we already have a nice solution to the simpler problem. We do not want to throw it away. Then, it becomes how can we reduce this problem to the simpler one. Actually, extending from the logic that if house i is not robbed, then you are free to choose whether to rob house i + 1, you can break the circle by assuming a house is not robbed.

For example, 1 -> 2 -> 3 -> 1 becomes 2 -> 3 if 1 is not robbed.

Since every house is either robbed or not robbed and at least half of the houses are not robbed, the solution is simply the larger of two cases with consecutive houses, i.e. house i not robbed, break the circle, solve it, or house i + 1 not robbed. Hence, the following solution. I chose i = n and i + 1 = 0 for simpler coding. But, you can choose whichever two consecutive ones.

```
public int rob(int[] nums) {
    if (nums.length == 1) return nums[0];
    return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1))
;
}
```

written by lx223 original link here

Answer 2

This problem is a little tricky at first glance. However, if you have finished the **House Robber** problem, this problem can simply be **decomposed into two House Robber problems**. Suppose there are `n` houses, since house `0` and `n -`

`1` are now neighbors, we cannot rob them together and thus the solution is now the maximum of

1. Rob houses `0` to `n - 2`;
2. Rob houses `1` to `n - 1`.

The code is as follows. Some edge cases (`n < 2`) are handled explicitly.

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n < 2) return n ? nums[0] : 0;
        return max(robber(nums, 0, n - 2), robber(nums, 1, n - 1));
    }
private:
    int robber(vector<int>& nums, int l, int r) {
        int pre = 0, cur = 0;
        for (int i = l; i <= r; i++) {
            int temp = max(pre + nums[i], cur);
            pre = cur;
            cur = temp;
        }
        return cur;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

Twice pass:

1. not rob nums[n-1]
2. not rob nums[0]

and the other is same as House Robber.

```
int rob(vector<int>& nums)
{
    if(nums.size() == 0)
        return 0;
    if(nums.size() == 1)
        return nums[0];

    int pre1 = 0, cur1 = 0;
    for(int i = 0; i < nums.size() - 1; ++ i)
    {
        int temp = pre1;
        pre1 = cur1;
        cur1 = max(temp + nums[i], pre1);
    }

    int pre2 = 0, cur2 = 0;
    for(int i = 1; i < nums.size(); ++ i)
    {
        int temp = pre2;
        pre2 = cur2;
        cur2 = max(temp + nums[i], pre2);
    }

    return max(cur1, cur2);
}
```

written by makuiyu original link here

Shortest Palindrome(214)

Answer 1

We can construct the following string and run KMP algorithm on it: (s) + (some symbol not present in s) + (reversed string)

After running KMP on that string as result we get a vector**p** with values of a prefix function for each character (for definition of a prefix function see KMP algorithm description). We are only interested in the last value because it shows us the largest suffix of the reversed string that matches the prefix of the original string. So basically all we left to do is to add the first k characters of the reversed string to the original string, where k is a difference between original string size and the prefix function for the last character of a constructed string.

```
class Solution {
public:
    string shortestPalindrome(string s) {
        string rev_s = s;
        reverse(rev_s.begin(), rev_s.end());
        string l = s + "#" + rev_s;

        vector<int> p(l.size(), 0);
        for (int i = 1; i < l.size(); i++) {
            int j = p[i - 1];
            while (j > 0 && l[i] != l[j])
                j = p[j - 1];
            p[i] = (j += l[i] == l[j]);
        }

        return rev_s.substr(0, s.size() - p[l.size() - 1]) + s;
    }
};
```

written by Sammax original link here

Answer 2

The idea is to use two anchors `j` and `i` to compare the String from beginning and end. If `j` can reach the end, the String itself is Palindrome. Otherwise, we divide the String by `j`, and get `mid = s.substring(0, j)` and `suffix`.

We reverse `suffix` as beginning of result and recursively call `shortestPalindrome` to get result of `mid` then appedn `suffix` to get result.

```
    int j = 0;
    for (int i = s.length() - 1; i >= 0; i--) {
        if (s.charAt(i) == s.charAt(j)) { j += 1; }
    }
    if (j == s.length()) { return s; }
    String suffix = s.substring(j);
    return new StringBuffer(suffix).reverse().toString() + shortestPalindrome(s.s
ubstring(0, j)) + suffix;
```

written by xcv58 original link here

Answer 3

The key idea is to first reverse the string, then check the max length from n to 0

```
class Solution {
public:
    string shortestPalindrome(string s) {
        string s2=s;
        reverse(s2.begin(),s2.end());
        int n=s.size(),l;
        for(l=n;l>=0;l--)
        {
            if(s.substr(0,l)==s2.substr(n-l))
                break;
        }
        return s2.substr(0,n-l)+s;
    }
};
```

written by pwh1 original link here

Kth Largest Element in an Array(215)

Answer 1

This problem is well known and quite often can be found in various text books.

You can take a couple of approaches to actually solve it:

- O(N lg N) running time + O(1) memory

The simplest approach is to sort the entire input array and then access the element by it's index (which is O(1)) operation:

```
public int findKthLargest(int[] nums, int k) {
        final int N = nums.length;
        Arrays.sort(nums);
        return nums[N - k];
}
```

- O(N lg K) running time + O(K) memory

Other possibility is to use a min oriented priority queue that will store the K-th largest values. The algorithm iterates over the whole input and maintains the size of priority queue.

```
public int findKthLargest(int[] nums, int k) {

    final PriorityQueue<Integer> pq = new PriorityQueue<>();
    for(int val : nums) {
        pq.offer(val);

        if(pq.size() > k) {
            pq.poll();
        }
    }
    return pq.peek();
}
```

- O(N) best case / O(N^2) worst case running time + O(1) memory

The smart approach for this problem is to use the selection algorithm (based on the partion method - the same one as used in quicksort).

```
public int findKthLargest(int[] nums, int k) {

        k = nums.length - k;
        int lo = 0;
        int hi = nums.length - 1;
        while (lo < hi) {
            final int j = partition(nums, lo, hi);
            if(j < k) {
                lo = j + 1;
            } else if (j > k) {
                hi = j - 1;
            } else {
                break;
            }
        }
        return nums[k];
    }

    private int partition(int[] a, int lo, int hi) {

        int i = lo;
        int j = hi + 1;
        while(true) {
            while(i < hi && less(a[++i], a[lo]));
            while(j > lo && less(a[lo], a[--j]));
            if(i >= j) {
                break;
            }
            exch(a, i, j);
        }
        exch(a, lo, j);
        return j;
    }

    private void exch(int[] a, int i, int j) {
        final int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }

    private boolean less(int v, int w) {
        return v < w;
    }
```

O(N) guaranteed running time + O(1) space

So how can we improve the above solution and make it O(N) guaranteed? The answer is quite simple, we can randomize the input, so that even when the worst case input would be provided the algorithm wouldn't be affected. So all what it is needed to be done is to shuffle the input.

```
public int findKthLargest(int[] nums, int k) {

        shuffle(nums);
        k = nums.length - k;
        int lo = 0;
        int hi = nums.length - 1;
        while (lo < hi) {
            final int j = partition(nums, lo, hi);
            if(j < k) {
                lo = j + 1;
            } else if (j > k) {
                hi = j - 1;
            } else {
                break;
            }
        }
        return nums[k];
    }

private void shuffle(int a[]) {

        final Random random = new Random();
        for(int ind = 1; ind < a.length; ind++) {
            final int r = random.nextInt(ind + 1);
            exch(a, ind, r);
        }
    }
```

There is also worth mentioning the Blum-Floyd-Pratt-Rivest-Tarjan algorithm that has a guaranteed O(N) running time.

written by jmnarloch original link here

Answer 2

Well, this problem has a naive solution, which is to sort the array in descending order and return the `k-1`-th element. However, sorting algorithm gives `O(nlogn)` complexity. Suppose `n = 10000` and `k = 2`, then we are doing a lot of unnecessary operations. In fact, this problem has at least two simple and faster solutions.

Well, the faster solution has no mystery. It is also closely related to sorting. I will give two algorithms for this problem below, one using quicksort(specifically, the partition subroutine) and the other using heapsort.

Quicksort

In quicksort, in each iteration, we need to select a pivot and then partition the array into three parts:

1. Elements smaller than the pivot;
2. Elements equal to the pivot;

3. Elements larger than the pivot.

Now, let's do an example with the array `[3, 2, 1, 5, 4, 6]` in the problem statement. Let's assume in each time we select the leftmost element to be the pivot, in this case, `3`. We then use it to partition the array into the above 3 parts, which results in `[1, 2, 3, 5, 4, 6]`. Now `3` is in the third position and we know that it is the third smallest element. Now, do you recognize that this subroutine can be used to solve this problem?

In fact, the above partition puts elements smaller than the pivot before the pivot and thus the pivot will then be the `k`-th smallest element if it is at the `k-1`-th position. Since the problem requires us to find the `k`-th largest element, we can simply modify the partition to put elements larger than the pivot before the pivot. That is, after partition, the array becomes `[5, 6, 4, 3, 1, 2]`. Now we know that `3` is the `4`-th largest element. If we are asked to find the `2`-th largest element, then we know it is left to `3`. If we are asked to find the `5`-th largest element, then we know it is right to `3`. So, in the **average** sense, the problem is reduced to approximately half of its original size, giving the recursion `T(n) = T(n/2) + O(n)` in which `O(n)` is the time for partition. This recursion, once solved, gives `T(n) = O(n)` and thus we have a linear time solution. Note that since we only need to consider one half of the array, the time complexity is `O(n)`. If we need to consider both the two halves of the array, like quicksort, then the recursion will be `T(n) = 2T(n/2) + O(n)` and the complexity will be `O(nlogn)`.

Of course, `O(n)` is the average time complexity. In the worst case, the recursion may become `T(n) = T(n - 1) + O(n)` and the complexity will be `O(n^2)`.

Now let's briefly write down the algorithm before writing our codes.

1. Initialize `left` to be 0 and `right` to be `nums.size() - 1`;
2. Partition the array, if the pivot is at the `k-1`-th position, return it (we are done);
3. If the pivot is right to the `k-1`-th position, update `right` to be the left neighbor of the pivot;
4. Else update `left` to be the right neighbor of the pivot.
5. Repeat 2.

Now let's turn it into code.

```
class Solution {
public:
    int partition(vector<int>& nums, int left, int right) {
        int pivot = nums[left];
        int l = left + 1, r = right;
        while (l <= r) {
            if (nums[l] < pivot && nums[r] > pivot)
                swap(nums[l++], nums[r--]);
            if (nums[l] >= pivot) l++;
            if (nums[r] <= pivot) r--;
        }
        swap(nums[left], nums[r]);
        return r;
    }

    int findKthLargest(vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pos = partition(nums, left, right);
            if (pos == k - 1) return nums[pos];
            if (pos > k - 1) right = pos - 1;
            else left = pos + 1;
        }
    }
};
```

Heapsort

Well, this problem still has a tag "heap". If you are familiar with heapsort, you can solve this problem using the following idea:

1. Build a max-heap for `nums`, set `heap_size` to be `nums.size()`;
2. Swap `nums[0]` (after buding the max-heap, it will be the largest element) with `nums[heap_size - 1]` (currently the last element). Then decrease `heap_size` by 1 and max-heapify `nums` (recovering its max-heap property) at index `0`;
3. Repeat 2 for `k` times and the `k`-th largest element will be stored finally at `nums[heap_size]`.

Now I paste my code below. If you find it tricky, I suggest you to read the Heapsort chapter of Introduction to Algorithms, which has a nice explanation of the algorithm. My code simply translates the pseudo code in that book :-)

```
class Solution {
public:
    inline int left(int idx) {
        return (idx << 1) + 1;
    }
    inline int right(int idx) {
        return (idx << 1) + 2;
    }
    void max_heapify(vector<int>& nums, int idx) {
        int largest = idx;
        int l = left(idx), r = right(idx);
        if (l < heap_size && nums[l] > nums[largest]) largest = l;
        if (r < heap_size && nums[r] > nums[largest]) largest = r;
        if (largest != idx) {
            swap(nums[idx], nums[largest]);
            max_heapify(nums, largest);
        }
    }
    void build_max_heap(vector<int>& nums) {
        heap_size = nums.size();
        for (int i = (heap_size >> 1) - 1; i >= 0; i--)
            max_heapify(nums, i);
    }
    int findKthLargest(vector<int>& nums, int k) {
        build_max_heap(nums);
        for (int i = 0; i < k; i++) {
            swap(nums[0], nums[heap_size - 1]);
            heap_size--;
            max_heapify(nums, 0);
        }
        return nums[heap_size];
    }
private:
    int heap_size;
}
```

If we are allowed to use the built-in `priority_queue`, the code will be much more shorter :-)

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int> pq(nums.begin(), nums.end());
        for (int i = 0; i < k - 1; i++)
            pq.pop();
        return pq.top();
    }
};
```

Well, the `priority_queue` can also be replaced by `multiset` :-)

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        multiset<int> mset;
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            mset.insert(nums[i]);
            if (mset.size() > k)
                mset.erase(mset.begin());
        }
        return *mset.begin();
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

https://en.wikipedia.org/wiki/Quickselect

```java
public class Solution {

  public int findKthLargest(int[] a, int k) {
    int n = a.length;
    int p = quickSelect(a, 0, n - 1, n - k + 1);
    return a[p];
  }

  // return the index of the kth smallest number
  int quickSelect(int[] a, int lo, int hi, int k) {
    // use quick sort's idea
    // put nums that are <= pivot to the left
    // put nums that are  > pivot to the right
    int i = lo, j = hi, pivot = a[hi];
    while (i < j) {
      if (a[i++] > pivot) swap(a, --i, --j);
    }
    swap(a, i, hi);

    // count the nums that are <= pivot from lo
    int m = i - lo + 1;

    // pivot is the one!
    if (m == k)     return i;
    // pivot is too big, so it must be on the left
    else if (m > k) return quickSelect(a, lo, i - 1, k);
    // pivot is too small, so it must be on the right
    else            return quickSelect(a, i + 1, hi, k - m);
  }

  void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }

}
```

written by jeantimex original link here

Combination Sum III(216)

Answer 1

```
class Solution {
public:
  void combination(vector<vector<int>>& result, vector<int> sol, int k, int n) {
    if (sol.size() == k && n == 0) { result.push_back(sol); return ; }
    if (sol.size() < k) {
      for (int i = sol.empty() ? 1 : sol.back() + 1; i <= 9; ++i) {
        if (n - i < 0) break;
        sol.push_back(i);
        combination(result, sol, k, n - i);
        sol.pop_back();
      }
    }
  }

  vector<vector<int>> combinationSum3(int k, int n) {
    vector<vector<int>> result;
    vector<int> sol;
    combination(result, sol, k, n);
    return result;
  }
};
```

written by yushu original link here

Answer 2

```
vector<vector<int>> combinationSum3(int k, int n) {
    vector<vector<int>> result;
    vector<int> path;
    backtrack(result, path, 1, k, n);
    return result;
}

void backtrack(vector<vector<int>> &result, vector<int> &path, int start, int k,
int target){
    if(target==0&&k==0){
        result.push_back(path);
        return;
    }
    for(int i=start;i<=10-k&&i<=target;i++){
        path.push_back(i);
        backtrack(result,path,i+1,k-1,target-i);
        path.pop_back();
    }
}
```

written by zephyr3 original link here

Answer 3

The idea is to choose proper number for 1,2..kth position in ascending order, and for each position, we only iterate through (prev_num, n/k]. Time comlexity O(k)

```
public class Solution {
    private List<List<Integer>> res = new ArrayList<List<Integer>>();
    public List<List<Integer>> combinationSum3(int k, int n) {
        findCombo( k, n, 1, new LinkedList<Integer>() );
        return res;
    }
    public void findCombo(int k, int n, int start, List<Integer> list){
        if( k == 1 ){
            if( n < start || n > 9 ) return;
            list.add( n );
            res.add( list );
            return;
        }
        for( int i = start; i <= n / k && i < 10; i++ ){
            List<Integer> subList = new LinkedList<Integer>( list );
            sub.add( i );
            findCombo( k - 1, n - i, i + 1, subList );
        }
    }
}
```

written by ericwanghai original link here

Contains Duplicate(217)

Answer 1

This problem seems trivial, so lets try different approaches to solve it:

Starting from worst time complexity to the best one:

Time complexity: O(N^2), memory: O(1)

The naive approach would be to run a iteration for each element and see whether a duplicate value can be found: this results in O(N^2) time complexity.

```
public boolean containsDuplicate(int[] nums) {

        for(int i = 0; i < nums.length; i++) {
            for(int j = i + 1; j < nums.length; j++) {
                if(nums[i] == nums[j]) {
                    return true;
                }
            }
        }
        return false;
    }
```

Time complexity: O(N lg N), memory: O(1) - not counting the memory used by sort

Since it is trivial task to find duplicates in sorted array, we can sort it as the first step of the algorithm and then search for consecutive duplicates.

```
    public boolean containsDuplicate(int[] nums) {

        Arrays.sort(nums);
        for(int ind = 1; ind < nums.length; ind++) {
            if(nums[ind] == nums[ind - 1]) {
                return true;
            }
        }
        return false;
    }
```

Time complexity: O(N), memory: O(N)

Finally we can used a well known data structure hash table that will help us to identify whether an element has been previously encountered in the array.

```
public boolean containsDuplicate(int[] nums) {

    final Set<Integer> distinct = new HashSet<Integer>();
    for(int num : nums) {
        if(distinct.contains(num)) {
            return true;
        }
        distinct.add(num);
    }
    return false;
}
```

This is trivial but quite nice example of space-time tradeoff.

written by jmnarloch original link here

Answer 2

Using anonymous set<>.
Not the most efficient as many already pointed out... but if you like one-liners ;)
akin to the solution possible with python.

```
#include <set>
using namespace std;

class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        return nums.size() > set<int>(nums.begin(), nums.end()).size();
    }
};
```

written by chammika original link here

Answer 3

```
class Solution {
  public:
    bool containsDuplicate(vector<int>& nums) {
        return set<int>(nums.begin(), nums.end()).size() < nums.size();
    }
};
```

written by kaikai2 original link here

The Skyline Problem(218)

Answer 1

Detailed explanation: http://www.geeksforgeeks.org/divide-and-conquer-set-7-the-skyline-problem/

```java
public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
        if (buildings.length == 0)
            return new LinkedList<int[]>();
        return recurSkyline(buildings, 0, buildings.length - 1);
    }

    private LinkedList<int[]> recurSkyline(int[][] buildings, int p, int q) {
        if (p < q) {
            int mid = p + (q - p) / 2;
            return merge(recurSkyline(buildings, p, mid),
                    recurSkyline(buildings, mid + 1, q));
        } else {
            LinkedList<int[]> rs = new LinkedList<int[]>();
            rs.add(new int[] { buildings[p][0], buildings[p][2] });
            rs.add(new int[] { buildings[p][1], 0 });
            return rs;
        }
    }

    private LinkedList<int[]> merge(LinkedList<int[]> l1, LinkedList<int[]> l2) {
        LinkedList<int[]> rs = new LinkedList<int[]>();
        int h1 = 0, h2 = 0;
        while (l1.size() > 0 && l2.size() > 0) {
            int x = 0, h = 0;
            if (l1.getFirst()[0] < l2.getFirst()[0]) {
                x = l1.getFirst()[0];
                h1 = l1.getFirst()[1];
                h = Math.max(h1, h2);
                l1.removeFirst();
            } else if (l1.getFirst()[0] > l2.getFirst()[0]) {
                x = l2.getFirst()[0];
                h2 = l2.getFirst()[1];
                h = Math.max(h1, h2);
                l2.removeFirst();
            } else {
                x = l1.getFirst()[0];
                h1 = l1.getFirst()[1];
                h2 = l2.getFirst()[1];
                h = Math.max(h1, h2);
                l1.removeFirst();
                l2.removeFirst();
            }
            if (rs.size() == 0 || h != rs.getLast()[1]) {
                rs.add(new int[] { x, h });
            }
        }
        rs.addAll(l1);
        rs.addAll(l2);
        return rs;
    }
}
```

written by hscaizh original link here

Answer 2

The idea is to do line sweep and just process the buildings only at the start and end points. The key is to use a priority queue to save all the buildings that are still "alive". The queue is sorted by its height and end time (the larger height first and if equal height, the one with a bigger end time first). For each iteration, we first find the current process time, which is either the next new building start time or the end time of the top entry of the live queue. If the new building start time is larger than the top one end time, then process the one in the queue first (pop them until it is empty or find the first one that ends after the new building); otherswise, if the new building starts before the top one ends, then process the new building (just put them in the queue). After processing, output it to the resulting vector if the height changes. Complexity is the worst case O(NlogN)

Not sure why my algorithm is so slow considering others' Python solution can achieve 160ms, any commments?

```
class Solution {
public:
    vector<pair<int, int>> getSkyline(vector<vector<int>>& buildings) {
        vector<pair<int, int>> res;
        int cur=0, cur_X, cur_H =-1,  len = buildings.size();
        priority_queue< pair<int, int>> liveBlg; // first: height, second, end ti
me
        while(cur<len || !liveBlg.empty())
        { // if either some new building is not processed or live building queue
is not empty
            cur_X = liveBlg.empty()? buildings[cur][0]:liveBlg.top().second; // n
ext timing point to process

            if(cur>=len || buildings[cur][0] > cur_X)
            { //first check if the current tallest building will end before the n
ext timing point
                  // pop up the processed buildings, i.e. those  have height no l
arger than cur_H and end before the top one
                while(!liveBlg.empty() && ( liveBlg.top().second <= cur_X) ) live
Blg.pop();
            }
            else
            { // if the next new building starts before the top one ends, process
the new building in the vector
                cur_X = buildings[cur][0];
                while(cur<len && buildings[cur][0]== cur_X)  // go through all th
e new buildings that starts at the same point
                {  // just push them in the queue
                    liveBlg.push(make_pair(buildings[cur][2], buildings[cur][1]))
;

                    cur++;
                }
            }
            cur_H = liveBlg.empty()?0:liveBlg.top().first; // outut the top one
            if(res.empty() || (res.back().second != cur_H) ) res.push_back(make_p
air(cur_X, cur_H));
        }
        return res;
    }
};
```

written by dong.wang.1694 original link here

Answer 3

```
    public List<int[]> getSkyline(int[][] buildings) {
    List<int[]> result = new ArrayList<>();
    List<int[]> height = new ArrayList<>();
    for(int[] b:buildings) {
        height.add(new int[]{b[0], -b[2]});
        height.add(new int[]{b[1], b[2]});
    }
    Collections.sort(height, (a, b) -> {
            if(a[0] != b[0])
                return a[0] - b[0];
            return a[1] - b[1];
    });
    Queue<Integer> pq = new PriorityQueue<>((a, b) -> (b - a));
    pq.offer(0);
    int prev = 0;
    for(int[] h:height) {
        if(h[1] < 0) {
            pq.offer(-h[1]);
        } else {
            pq.remove(h[1]);
        }
        int cur = pq.peek();
        if(prev != cur) {
            result.add(new int[]{h[0], cur});
            prev = cur;
        }
    }
    return result;
}
```

written by jinwu original link here

Contains Duplicate II(219)

Answer 1

```
public boolean containsNearbyDuplicate(int[] nums, int k) {
        Set<Integer> set = new HashSet<Integer>();
        for(int i = 0; i < nums.length; i++){
            if(i > k) set.remove(nums[i-k-1]);
            if(!set.add(nums[i])) return true;
        }
        return false;
 }
```

written by southpenguin original link here

Answer 2

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k)
    {
       unordered_set<int> s;

       if (k <= 0) return false;
       if (k >= nums.size()) k = nums.size() - 1;

       for (int i = 0; i < nums.size(); i++)
       {
           if (i > k) s.erase(nums[i - k - 1]);
           if (s.find(nums[i]) != s.end()) return true;
           s.insert(nums[i]);
       }

       return false;
    }
};
```

The basic idea is to maintain a set s which contain unique values from nums[i - k] to nums[i - 1], if nums[i] is in set s then return true else update the set.

written by luo_seu original link here

Answer 3

```
bool containsNearbyDuplicate(vector<int>& nums, int k) {
    set<int> cand;
    for (int i = 0; i < nums.size(); i++) {
        if (i > k) cand.erase(nums[i-k-1]);
        if (!cand.insert(nums[i]).second) return true;
    }
    return false;
}
```

written by lchen77 original link here

Contains Duplicate III(220)

Answer 1

As a followup question, it naturally also requires maintaining a window of size k. When t == 0, it reduces to the previous question so we just reuse the solution.

Since there is now a constraint on the range of the values of the elements to be considered duplicates, it reminds us of doing a range check which is implemented in tree data structure and would take O(LogN) if a balanced tree structure is used, or doing a bucket check which is constant time. We shall just discuss the idea using bucket here.

Bucketing means we map a range of values to the a bucket. For example, if the bucket size is 3, we consider 0, 1, 2 all map to the same bucket. However, if t == 3, (0, 3) is a considered duplicates but does not map to the same bucket. This is fine since we are checking the buckets immediately before and after as well. So, as a rule of thumb, just make sure the size of the bucket is reasonable such that elements having the same bucket is immediately considered duplicates or duplicates must lie within adjacent buckets. So this actually gives us a range of possible bucket size, i.e. t and t + 1. We just choose it to be t and a bucket mapping to be *num / t*.

Another complication is that negative ints are allowed. A simple*num / t* just shrinks everything towards 0. Therefore, we can just reposition every element to start from Integer.MIN_VALUE.

```
 public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (k < 1 || t < 0) return false;
        Map<Long, Long> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            long remappedNum = (long) nums[i] - Integer.MIN_VALUE;
            long bucket = remappedNum / ((long) t + 1);
            if (map.containsKey(bucket)
                    || (map.containsKey(bucket - 1) && remappedNum - map.get(buck
et - 1) <= t)
                        || (map.containsKey(bucket + 1) && map.get(bucket + 1) -
remappedNum <= t))
                            return true;
            if (map.entrySet().size() >= k) {
                long lastBucket = ((long) nums[i - k] - Integer.MIN_VALUE) / ((lo
ng) t + 1);
                map.remove(lastBucket);
            }
            map.put(bucket, remappedNum);
        }
        return false;
    }
}
```

Edits:

Actually, we can use t + 1 as the bucket size to get rid of the case when t == 0. It

simplifies the code. The above code is therefore the updated version.

written by lx223 original link here

Answer 2

This problem requires to maintain a window of size k of the previous values that can be queried for value ranges. The best data structure to do that is Binary Search Tree. As a result maintaining the tree of size k will result in time complexity O(N lg K). In order to check if there exists any value of range abs(nums[i] - nums[j]) to simple queries can be executed both of time complexity O(lg K)

Here is the whole solution using TreeMap.

```
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (nums == null || nums.length == 0 || k <= 0) {
            return false;
        }

        final TreeSet<Integer> values = new TreeSet<>();
        for (int ind = 0; ind < nums.length; ind++) {

            final Integer floor = values.floor(nums[ind] + t);
            final Integer ceil = values.ceiling(nums[ind] - t);
            if ((floor != null && floor >= nums[ind])
                    || (ceil != null && ceil <= nums[ind])) {
                return true;
            }

            values.add(nums[ind]);
            if (ind >= k) {
                values.remove(nums[ind - k]);
            }
        }

        return false;
    }
}
```

written by jmnarloch original link here

Answer 3

```
 bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
    set<int> window; // set is ordered automatically
    for (int i = 0; i < nums.size(); i++) {
        if (i > k) window.erase(nums[i-k-1]); // keep the set contains nums i j a
t most k
        // -t <= x - nums[i] <= t;
        auto pos = window.lower_bound(nums[i] - t); // x >= nums[i] - t
        if (pos != window.end() && *pos - nums[i] <= t) // x <= nums[i] + t
               return true;
        window.insert(nums[i]);
    }
    return false;
}
```

written by lchen77 original link here

Maximal Square(221)

Answer 1

Well, this problem desires for the use of dynamic programming. They key to any DP problem is to come up with the state equation. In this problem, we define the state to be **the maximal size of the square that can be achieved at point** `(i, j)`, denoted as `P[i][j]`. Remember that we use **size** instead of square as the state (`square = size^2`).

Now let's try to come up with the formula for `P[i][j]`.

First, it is obvious that for the topmost row (`i = 0`) and the leftmost column (`j = 0`), `P[i][j] = matrix[i][j]`. This is easily understood. Let's suppose that the topmost row of `matrix` is like `[1, 0, 0, 1]`. Then we can immediately know that the first and last point can be a square of size `1` while the two middle points cannot make any square, giving a size of `0`. Thus, `P = [1, 0, 0, 1]`, which is the same as `matrix`. The case is similar for the leftmost column. Till now, the boundary conditions of this DP problem are solved.

Let's move to the more general case for `P[i][j]` in which `i > 0` and `j > 0`. First of all, let's see another simple case in which `matrix[i][j] = 0`. It is obvious that `P[i][j] = 0` too. Why? Well, since `matrix[i][j] = 0`, no square will contain `matrix[i][j]`. According to our definition of `P[i][j]`, `P[i][j]` is also `0`.

Now we are almost done. The only unsolved case is `matrix[i][j] = 1`. Let's see an example.

Suppose `matrix = [[0, 1], [1, 1]]`, it is obvious that `P[0][0] = 0, P[0][1] = P[1][0] = 1`, what about `P[1][1]`? Well, to give a square of size larger than `1` in `P[1][1]`, all of its three neighbors (left, up, left-up) should be non-zero, right? In this case, the left-up neighbor `P[0][0] = 0`, so `P[1][1]` can only be 1, which means that it contains the square of itself.

Now you are near the solution. In fact, `P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1` in this case.

Taking all these together, we have the following state equations.

1. `P[0][j] = matrix[0][j]` (topmost row);
2. `P[i][0] = matrix[i][0]` (leftmost column);
3. For `i > 0` and `j > 0`: if `matrix[i][j] = 0`, `P[i][j] = 0`; if `matrix[i][j] = 1`, `P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1`.

Putting them into codes, and maintain a variable `maxsize` to record the maximum size of the square we have seen, we have the following (unoptimized) solution.

```
int maximalSquare(vector<vector<char>>& matrix) {
    int m = matrix.size();
    if (!m) return 0;
    int n = matrix[0].size();
    vector<vector<int> > size(m, vector<int>(n, 0));
    int maxsize = 0;
    for (int j = 0; j < n; j++) {
        size[0][j] = matrix[0][j] - '0';
        maxsize = max(maxsize, size[0][j]);
    }
    for (int i = 1; i < m; i++) {
        size[i][0] = matrix[i][0] - '0';
        maxsize = max(maxsize, size[i][0]);
    }
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][j] == '1') {
                size[i][j] = min(size[i - 1][j - 1], min(size[i - 1][j], size[i][
j - 1])) + 1;
                maxsize = max(maxsize, size[i][j]);
            }
        }
    }
    return maxsize * maxsize;
}
```

Now let's try to optimize the above solution. As can be seen, each time when we update `size[i][j]`, we only need `size[i][j - 1], size[i - 1][j - 1]` (at the previous left column) and `size[i - 1][j]` (at the current column). So we do not need to maintain the full `m*n` matrix. In fact, keeping two columns is enough. Now we have the following optimized solution.

```
int maximalSquare(vector<vector<char>>& matrix) {
    int m = matrix.size();
    if (!m) return 0;
    int n = matrix[0].size();
    vector<int> pre(m, 0);
    vector<int> cur(m, 0);
    int maxsize = 0;
    for (int i = 0; i < m; i++) {
        pre[i] = matrix[i][0] - '0';
        maxsize = max(maxsize, pre[i]);
    }
    for (int j = 1; j < n; j++) {
        cur[0] = matrix[0][j] - '0';
        maxsize = max(maxsize, cur[0]);
        for (int i = 1; i < m; i++) {
            if (matrix[i][j] == '1') {
                cur[i] = min(cur[i - 1], min(pre[i - 1], pre[i])) + 1;
                maxsize = max(maxsize, cur[i]);
            }
        }
        swap(pre, cur);
        fill(cur.begin(), cur.end(), 0);
    }
    return maxsize * maxsize;
}
```

Now you see the solution is finished? In fact, it can still be optimized! In fact, we need not maintain two vectors and one is enough. If you want to explore this idea, please refer to the answers provided by @stellari below. Moreover, in the code above, we distinguish between the `0`-th row and other rows since the `0`-th row has no row above it. In fact, we can make all the `m` rows the same by padding a `0` row on the top (in the following code, we pad a `0` on top of `dp`). Finally, we will have the following short code :) If you find it hard to understand, try to run it using your pen and paper and notice how it realizes what the two-vector solution does using only one vector.

```
int maximalSquare(vector<vector<char>>& matrix) {
    if (matrix.empty()) return 0;
    int m = matrix.size(), n = matrix[0].size();
    vector<int> dp(m + 1, 0);
    int maxsize = 0, pre = 0;
    for (int j = 0; j < n; j++) {
        for (int i = 1; i <= m; i++) {
            int temp = dp[i];
            if (matrix[i - 1][j] == '1') {
                dp[i] = min(dp[i], min(dp[i - 1], pre)) + 1;
                maxsize = max(maxsize, dp[i]);
            }
            else dp[i] = 0;
            pre = temp;
        }
    }
    return maxsize * maxsize;
}
```

This solution, since posted, has been suggested various improvements by kind people. For a more comprehensive collection of the solutions, please visit my technical blog.

written by jianchao.li.fighter original link here

Answer 2

```
public int maximalSquare(char[][] a) {
    if(a.length == 0) return 0;
    int m = a.length, n = a[0].length, result = 0;
    int[][] b = new int[m+1][n+1];
    for (int i = 1 ; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if(a[i-1][j-1] == '1') {
                b[i][j] = Math.min(Math.min(b[i][j-1] , b[i-1][j-1]), b[i-1][j])
+ 1;
                result = Math.max(b[i][j], result); // update result
            }
        }
    }
    return result*result;
}
```

written by andywhite original link here

Answer 3

```java
public int maximalSquare(char[][] a) {
  if (a == null || a.length == 0 || a[0].length == 0)
    return 0;

  int max = 0, n = a.length, m = a[0].length;

  // dp(i, j) represents the length of the square
  // whose lower-right corner is located at (i, j)
  // dp(i, j) = min{ dp(i-1, j-1), dp(i-1, j), dp(i, j-1) }
  int[][] dp = new int[n + 1][m + 1];

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (a[i - 1][j - 1] == '1') {
        dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]
)) + 1;
        max = Math.max(max, dp[i][j]);
      }
    }
  }

  // return the area
  return max * max;
}
```

written by jeantimex original link here

Count Complete Tree Nodes(222)

Answer 1

Main Solution - 572 ms

```
class Solution {
    int height(TreeNode root) {
        return root == null ? -1 : 1 + height(root.left);
    }
    public int countNodes(TreeNode root) {
        int h = height(root);
        return h < 0 ? 0 :
               height(root.right) == h-1 ? (1 << h) + countNodes(root.right)
                                         : (1 << h-1) + countNodes(root.left);
    }
}
```

Explanation

The height of a tree can be found by just going left. Let a single node tree have height 0. Find the height `h` of the whole tree. If the whole tree is empty, i.e., has height -1, there are 0 nodes.

Otherwise check whether the height of the right subtree is just one less than that of the whole tree, meaning left and right subtree have the same height.

- If yes, then the last node on the last tree row is in the right subtree and the left subtree is a full tree of height h-1. So we take the 2^h-1 nodes of the left subtree plus the 1 root node plus recursively the number of nodes in the right subtree.
- If no, then the last node on the last tree row is in the left subtree and the right subtree is a full tree of height h-2. So we take the 2^(h-1)-1 nodes of the right subtree plus the 1 root node plus recursively the number of nodes in the left subtree.

Since I halve the tree in every recursive step, I have O(log(n)) steps. Finding a height costs O(log(n)). So overall O(log(n)^2).

Iterative Version - 508 ms

Here's an iterative version as well, with the benefit that I don't recompute `h` in every step.

```
class Solution {
    int height(TreeNode root) {
        return root == null ? -1 : 1 + height(root.left);
    }
    public int countNodes(TreeNode root) {
        int nodes = 0, h = height(root);
        while (root != null) {
            if (height(root.right) == h - 1) {
                nodes += 1 << h;
                root = root.right;
            } else {
                nodes += 1 << h-1;
                root = root.left;
            }
            h--;
        }
        return nodes;
    }
}
```

A Different Solution - 544 ms

Here's one based on victorlee's C++ solution.

```
class Solution {
    public int countNodes(TreeNode root) {
        if (root == null)
            return 0;
        TreeNode left = root, right = root;
        int height = 0;
        while (right != null) {
            left = left.left;
            right = right.right;
            height++;
        }
        if (left == null)
            return (1 << height) - 1;
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
}
```

Note that that's basically this:

```
public int countNodes(TreeNode root) {
    if (root == null)
        return 0;
    return 1 + countNodes(root.left) + countNodes(root.right)
```

That would be O(n). But... the actual solution has a gigantic optimization. It first walks all the way left and right to determine the height and whether it's a full tree, meaning the last row is full. If so, then the answer is just 2^height-1. And since

always at least one of the two recursive calls is such a full tree, at least one of the two calls immediately stops. Again we have runtime O(log(n)^2).

written by StefanPochmann original link here

Answer 2

```
class Solution {

public:

    int countNodes(TreeNode* root) {

        if(!root) return 0;

        int hl=0, hr=0;

        TreeNode *l=root, *r=root;

        while(l) {hl++;l=l->left;}

        while(r) {hr++;r=r->right;}

        if(hl==hr) return pow(2,hl)-1;

        return 1+countNodes(root->left)+countNodes(root->right);

    }

};
```

written by victorlee original link here

Answer 3

public class Solution {

```java
public int countNodes(TreeNode root) {

    int leftDepth = leftDepth(root);
    int rightDepth = rightDepth(root);

    if (leftDepth == rightDepth)
        return (1 << leftDepth) - 1;
    else
        return 1+countNodes(root.left) + countNodes(root.right);

}

private int rightDepth(TreeNode root) {
    // TODO Auto-generated method stub
    int dep = 0;
    while (root != null) {
        root = root.right;
        dep++;
    }
    return dep;
}

private int leftDepth(TreeNode root) {
    // TODO Auto-generated method stub
    int dep = 0;
    while (root != null) {
        root = root.left;
        dep++;
    }
    return dep;
}
```

}

written by m010 original link here

Rectangle Area(223)

Answer 1

Instead of checking whether the rectangles overlap, I max `right` with `left` (and `top` with `bottom`). Haven't seen that in other solutions.

```
int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
    int left = max(A,E), right = max(min(C,G), left);
    int bottom = max(B,F), top = max(min(D,H), bottom);
    return (C-A)*(D-B) - (right-left)*(top-bottom) + (G-E)*(H-F);
}
```

written by ManuelP original link here

Answer 2

```
public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {

        int areaOfSqrA = (C-A) * (D-B);
         int areaOfSqrB = (G-E) * (H-F);

        int left = Math.max(A, E);
        int right = Math.min(G, C);
        int bottom = Math.max(F, B);
        int top = Math.min(D, H);

        //If overlap
        int overlap = 0;
        if(right > left && top > bottom)
             overlap = (right - left) * (top - bottom);

        return areaOfSqrA + areaOfSqrB - overlap;
    }
```

Hello! So, the code should be fairly straightforward. I first calculate the area of each rectangle and then calculate the overlapping area between the two rectangles (if there is one!). At the end, we sum up the individual areas and subtract the overlapping area/o !

Feel free to ask should you have any queries for me OR if my solution can be improved upon! :)

written by waisuan original link here

Answer 3

In the statement, it says *"Each rectangle is defined by its bottom left corner and top right corner"*

However, there are test cases like

Input: -2, -2, 2, 2, -3, 3, -4, 4

Which is wrong, because there are no rectangles with bottom left corner (-3, 3) and top right corner (-4, 4).

written by madokamylove original link here

Basic Calculator(224)

Answer 1

Simple iterative solution by identifying characters one by one. One important thing is that the input is valid, which means the parentheses are always paired and in order. Only 5 possible input we need to pay attention:

1. digit: it should be one digit from the current number
2. '+': number is over, we can add the previous number and start a new number
3. '-': same as above
4. '(': push the previous result and the sign into the stack, set result to 0, just calculate the new result within the parenthesis.
5. ')': pop out the top two numbers from stack, first one is the sign before this pair of parenthesis, second is the temporary result before this pair of parenthesis. We add them together.

Finally if there is only one number, from the above solution, we haven't add the number to the result, so we do a check see if the number is zero.

```java
public int calculate(String s) {
    Stack<Integer> stack = new Stack<Integer>();
    int result = 0;
    int number = 0;
    int sign = 1;
    for(int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if(Character.isDigit(c)){
            number = 10 * number + (int)(c - '0');
        }else if(c == '+'){
            result += sign * number;
            number = 0;
            sign = 1;
        }else if(c == '-'){
            result += sign * number;
            number = 0;
            sign = -1;
        }else if(c == '('){
            //we push the result first, then sign;
            stack.push(result);
            stack.push(sign);
            //reset the sign and result for the value in the parenthesis
            sign = 1;
            result = 0;
        }else if(c == ')'){
            result += sign * number;
            number = 0;
            result *= stack.pop();    //stack.pop() is the sign before the parent
hesis
            result += stack.pop();   //stack.pop() now is the result calculated b
efore the parenthesis

        }
    }
    if(number != 0) result += sign * number;
    return result;
}
```

written by southpenguin original link here

Answer 2

```
class Solution {
public:
    int calculate(string s) {
        // the given expression is always valid!!!
        // only + and - !!!
        // every + and - can be flipped base on it's depth in ().
        stack<int> signs;
        int sign = 1;
        int num = 0;
        int ans = 0;

        // always transform s into ( s )
        signs.push(1);

        for (auto c : s) {
            if (c >= '0' && c <= '9') {
                num = 10 * num + c - '0';
            } else if (c == '+' || c == '-') {
                ans = ans + signs.top() * sign * num;
                num = 0;
                sign = (c == '+' ? 1 : -1);
            } else if (c == '(') {
                signs.push(sign * signs.top());
                sign = 1;
            } else if (c == ')') {
                ans = ans + signs.top() * sign * num;
                num = 0;
                signs.pop();
                sign = 1;
            }
        }

        if (num) {
            ans = ans + signs.top() * sign * num;
        }

        return ans;
    }
};
```

written by ironhead.chuang original link here

Answer 3

My approach is based on the fact that the final arithmetic operation on each number is not only depend on the sign directly operating on it, but all signs associated with each unmatched '(' before that number.

e.g. 5 - (6 + (4 - 7)), if we remove all parentheses, the expression becomes 5 - 6 - 4 + 7.

sign:

6: (-1)(1) = -1

4: (-1)(1)(1) = -1

7: (-1)(1)(-1) = 1

The effect of associated signs are cumulative, stack is builded based on this. Any improvement is welcome.

```
public int calculate(String s) {
    Deque<Integer> stack = new LinkedList<>();
    int rs = 0;
    int sign = 1;
    stack.push(1);
    for (int i = 0; i < s.length(); i++){
        if (s.charAt(i) == ' ') continue;
        else if (s.charAt(i) == '('){
            stack.push(stack.peekFirst() * sign);
            sign = 1;
        }
        else if (s.charAt(i) == ')') stack.pop();
        else if (s.charAt(i) == '+') sign = 1;
        else if (s.charAt(i) == '-') sign = -1;
        else{
            int temp = s.charAt(i) - '0';
            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1)))
                temp = temp * 10 + s.charAt(++i) - '0';
            rs += sign * stack.peekFirst() * temp;
        }
    }
    return rs;
}
```

written by bidp original link here

Implement Stack using Queues(225)

Answer 1

```
class Stack {
public:
    queue<int> que;
    // Push element x onto stack.
    void push(int x) {
        que.push(x);
        for(int i=0;i<que.size()-1;++i){
            que.push(que.front());
            que.pop();
        }
    }

    // Removes the element on top of the stack.
    void pop() {
        que.pop();
    }

    // Get the top element.
    int top() {
        return que.front();
    }

    // Return whether the stack is empty.
    bool empty() {
        return que.empty();
    }
};
```

written by sjtuldk original link here

Answer 2

```
class MyStack
{
    Queue<Integer> queue;

    public MyStack()
    {
        this.queue=new LinkedList<Integer>();
    }

    // Push element x onto stack.
    public void push(int x)
    {
       queue.add(x);
       for(int i=0;i<queue.size()-1;i++)
       {
           queue.add(queue.poll());
       }
    }

    // Removes the element on top of the stack.
    public void pop()
    {
        queue.poll();
    }

    // Get the top element.
    public int top()
    {
        return queue.peek();
    }

    // Return whether the stack is empty.
    public boolean empty()
    {
        return queue.isEmpty();
    }
}
```

written by YYZ90 original link here

Answer 3

```java
class MyStack {
    Queue<Integer> q = new LinkedList<Integer>();

    // Push element x onto stack.
    public void push(int x) {
        q.add(x);
    }

    // Removes the element on top of the stack.
    public void pop() {
        int size = q.size();
        for(int i = 1; i < size; i++)
            q.add(q.remove());
        q.remove();
    }

    // Get the top element.
    public int top() {
        int size = q.size();
        for(int i = 1; i < size; i++)
            q.add(q.remove());
        int ret = q.remove();
        q.add(ret);
        return ret;
    }

    // Return whether the stack is empty.
    public boolean empty() {
        return q.isEmpty();
    }
}
```

written by leo_aly7 original link here

Invert Binary Tree(226)

Answer 1

As in many other cases this problem has more than one possible solutions:

Lets start with straightforward - recursive DFS - it's easy to write and pretty much concise.

```
public class Solution {
    public TreeNode invertTree(TreeNode root) {

        if (root == null) {
            return null;
        }

        final TreeNode left = root.left,
                right = root.right;
        root.left = invertTree(right);
        root.right = invertTree(left);
        return root;
    }
}
```

The above solution is correct, but it is also bound to the application stack, which means that it's no so much scalable - (you can find the problem size that will overflow the stack and crash your application), so more robust solution would be to use stack data structure.

```java
public class Solution {
    public TreeNode invertTree(TreeNode root) {

        if (root == null) {
            return null;
        }

        final Deque<TreeNode> stack = new LinkedList<>();
        stack.push(root);

        while(!stack.isEmpty()) {
            final TreeNode node = stack.pop();
            final TreeNode left = node.left;
            node.left = node.right;
            node.right = left;

            if(node.left != null) {
                stack.push(node.left);
            }
            if(node.right != null) {
                stack.push(node.right);
            }
        }
        return root;
    }
}
```

Finally we can easly convert the above solution to BFS - or so called level order traversal.

```java
public class Solution {
    public TreeNode invertTree(TreeNode root) {

        if (root == null) {
            return null;
        }

        final Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while(!queue.isEmpty()) {
            final TreeNode node = queue.poll();
            final TreeNode left = node.left;
            node.left = node.right;
            node.right = left;

            if(node.left != null) {
                queue.offer(node.left);
            }
            if(node.right != null) {
                queue.offer(node.right);
            }
        }
        return root;
    }
}
```

If I can write this code, does it mean I can get job at Google? ;)

written by jmnarloch original link here

Answer 2

Recursive

```cpp
TreeNode* invertTree(TreeNode* root) {
    if (root) {
        invertTree(root->left);
        invertTree(root->right);
        std::swap(root->left, root->right);
    }
    return root;
}
```

Non-Recursive

```
TreeNode* invertTree(TreeNode* root) {
    std::stack<TreeNode*> stk;
    stk.push(root);

    while (!stk.empty()) {
        TreeNode* p = stk.top();
        stk.pop();
        if (p) {
            stk.push(p->left);
            stk.push(p->right);
            std::swap(p->left, p->right);
        }
    }
    return root;
}
```

written by chammika original link here

Answer 3

```
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        if(root == null) return null;
        TreeNode tmp = root.left;
        root.left = invertTree(root.right);
        root.right = invertTree(tmp);
        return root;
    }
}
```

written by SOY original link here

Basic Calculator II(227)

Answer 1

```
public class Solution {
public int calculate(String s) {
    int len;
    if(s==null || (len = s.length())==0) return 0;
    Stack<Integer> stack = new Stack<Integer>();
    int num = 0;
    char sign = '+';
    for(int i=0;i<len;i++){
        if(Character.isDigit(s.charAt(i))){
            num = num*10+s.charAt(i)-'0';
        }
        if((!Character.isDigit(s.charAt(i)) &&' '!=s.charAt(i)) || i==len-1){
            if(sign=='-'){
                stack.push(-num);
            }
            if(sign=='+'){
                stack.push(num);
            }
            if(sign=='*'){
                stack.push(stack.pop()*num);
            }
            if(sign=='/'){
                stack.push(stack.pop()/num);
            }
            sign = s.charAt(i);
            num = 0;
        }
    }

    int re = 0;
    for(int i:stack){
        re += i;
    }
    return re;
}
```

}

written by abner original link here

Answer 2

If you don't like the `44 - op` ASCII trick, you can use `op == '+' ? 1 : -1` instead. And wow, I didn't know C++ has `or`. I'm a Python guy and wrote that out of habit and only realized it after getting this accepted :-)

```
int calculate(string s) {
    istringstream in('+' + s + '+');
    long long total = 0, term = 0, n;
    char op;
    while (in >> op) {
        if (op == '+' or op == '-') {
            total += term;
            in >> term;
            term *= 44 - op;
        } else {
            in >> n;
            if (op == '*')
                term *= n;
            else
                term /= n;
        }
    }
    return total;
}
```

written by StefanPochmann original link here

Answer 3

```
class Solution {
public:
    int calculate(string s) {
        int result = 0, cur_res = 0;
        char op = '+';
        for(int pos = s.find_first_not_of(' '); pos < s.size(); pos = s.find_firs
t_not_of(' ', pos)) {
            if(isdigit(s[pos])) {
                int tmp = s[pos] - '0';
                while(++pos < s.size() && isdigit(s[pos]))
                    tmp = tmp*10 + (s[pos] - '0');
                switch(op) {
                    case '+' : cur_res += tmp; break;
                    case '-' : cur_res -= tmp; break;
                    case '*' : cur_res *= tmp; break;
                    case '/' : cur_res /= tmp; break;
                }
            }
            else {
                if(s[pos] == '+' || s[pos] == '-') {
                    result += cur_res;
                    cur_res = 0;
                }
                op = s[pos++];
            }
        }
        return result + cur_res;
    }
};
```

written by fastcode original link here

Summary Ranges(228)

Answer 1

```
List<String> list=new ArrayList();
    if(nums.length==1){
        list.add(nums[0]+"");
        return list;
    }
    for(int i=0;i<nums.length;i++){
        int a=nums[i];
        while(i+1<nums.length&&(nums[i+1]-nums[i])==1){
            i++;
        }
        if(a!=nums[i]){
            list.add(a+"->"+nums[i]);
        }else{
            list.add(a+"");
        }
    }
    return list;
```

written by tanchiyuxx original link here

Answer 2

Three versions of the same algorithm, all take O(n) time.

Solution 1

Just collect the ranges, then format and return them.

```
def summaryRanges(self, nums):
    ranges = []
    for n in nums:
        if not ranges or n > ranges[-1][-1] + 1:
            ranges += [],
        ranges[-1][1:] = n,
    return ['->'.join(map(str, r)) for r in ranges]
```

Solution 2

A variation of solution 1, holding the current range in an extra variable `r` to make things easier. Note that `r` contains at most two elements, so the `in`-check takes constant time.

```
def summaryRanges(self, nums):
    ranges, r = [], []
    for n in nums:
        if n-1 not in r:
            r = []
            ranges += r,
        r[1:] = n,
    return ['->'.join(map(str, r)) for r in ranges]
```

Solution 3

A tricky short version.

```
def summaryRanges(self, nums):
    ranges = r = []
    for n in nums:
        if `n-1` not in r:
            r = []
            ranges += r,
        r[1:] = `n`,
    return map('->'.join, ranges)
```

About the commas :-)

Three people asked about them in the comments, so I'll also explain it here as well. I have these two basic cases:

```
ranges += [],
r[1:] = n,
```

Why the trailing commas? Because it turns the right hand side into a tuple and I get the same effects as these more common alternatives:

```
ranges += [[]]
or
ranges.append([])

r[1:] = [n]
```

Without the comma, ...

- `ranges += []` wouldn't add `[]` itself but only its elements, i.e., nothing.
- `r[1:] = n` wouldn't work, because my `n` is not an iterable.

Why do it this way instead of the more common alternatives I showed above? Because it's shorter and faster (according to tests I did a while back).

written by StefanPochmann original link here

Answer 3

```
   vector<string> summaryRanges(vector<int>& nums) {
    const int size_n = nums.size();
    vector<string> res;
    if ( 0 == size_n) return res;
    for (int i = 0; i < size_n;) {
        int start = i, end = i;
        while (end + 1 < size_n && nums[end+1] == nums[end] + 1) end++;
        if (end > start) res.push_back(to_string(nums[start]) + "->" + to_string(
nums[end]));
        else res.push_back(to_string(nums[start]));
        i = end+1;
    }
    return res;
}
```

written by lchen77 original link here

Majority Element II(229)

Answer 1

For those who aren't familiar with Boyer-Moore Majority Vote algorithm, I found a great article (http://goo.gl/64Nams) that helps me to understand this fantastic algorithm!! Please check it out!

The essential concepts is you keep a counter for the majority number**X**. If you find a number **Y** that is not **X**, the current counter should deduce 1. The reason is that if there is 5 **X** and 4 **Y**, there would be one (5-4) more **X** than **Y**. This could be explained as "4 **X** being paired out by 4 **Y**".

And since the requirement is finding the majority for more than ceiling of [n/3], the answer would be less than or equal to two numbers. So we can modify the algorithm to maintain two counters for two majorities.

Followings are my sample Python code:

```
class Solution:
# @param {integer[]} nums
# @return {integer[]}
def majorityElement(self, nums):
    if not nums:
        return []
    count1, count2, candidate1, candidate2 = 0, 0, 0, 1
    for n in nums:
        if n == candidate1:
            count1 += 1
        elif n == candidate2:
            count2 += 1
        elif count1 == 0:
            candidate1, count1 = n, 1
        elif count2 == 0:
            candidate2, count2 = n, 1
        else:
            count1, count2 = count1 - 1, count2 - 1
    return [n for n in (candidate1, candidate2)
                    if nums.count(n) > len(nums) // 3]
```

written by chungyushao original link here

Answer 2

```
vector<int> majorityElement(vector<int>& nums) {
    int cnt1=0, cnt2=0;
    int a,b;

    for(int n: nums){
        if (cnt1 == 0 || n == a){
            cnt1++;
            a = n;
        }
        else if (cnt2 == 0 || n==b){
            cnt2++;
            b = n;
        }
        else{
            cnt1--;
            cnt2--;
        }
    }

    cnt1=cnt2=0;
    for(int n: nums){
        if (n==a)   cnt1++;
        else if (n==b) cnt2++;
    }

    vector<int> result;
    if (cnt1 > nums.size()/3)   result.push_back(a);
    if (cnt2 > nums.size()/3)   result.push_back(b);
    return result;
}
```

written by longren original link here

Answer 3

Boyer-Moore Majority Vote algorithm generalization to elements appear more than floor(n/k) times

```cpp
class Solution {
public:
  vector<int> majorityElement(vector<int> &a) {
    int y = 0, z = 1, cy = 0, cz = 0;
    for (auto x: a) {
      if (x == y) cy++;
      else if (x == z) cz++;
      else if (! cy) y = x, cy = 1;
      else if (! cz) z = x, cz = 1;
      else cy--, cz--;
    }
    cy = cz = 0;
    for (auto x: a)
      if (x == y) cy++;
      else if (x == z) cz++;
    vector<int> r;
    if (cy > a.size()/3) r.push_back(y);
    if (cz > a.size()/3) r.push_back(z);
    return r;
  }
};
```

written by MaskRay original link here

Kth Smallest Element in a BST(230)

Answer 1

Binary Search (dfs): most preferable

```
  public int kthSmallest(TreeNode root, int k) {
        int count = countNodes(root.left);
        if (k <= count) {
            return kthSmallest(root.left, k);
        } else if (k > count + 1) {
            return kthSmallest(root.right, k-1-count); // 1 is counted as current
node
        }

        return root.val;
    }

    public int countNodes(TreeNode n) {
        if (n == null) return 0;

        return 1 + countNodes(n.left) + countNodes(n.right);
    }
```

DFS in-order recursive:

```
    // better keep these two variables in a wrapper class
    private static int number = 0;
    private static int count = 0;

    public int kthSmallest(TreeNode root, int k) {
        count = k;
        helper(root);
        return number;
    }

    public void helper(TreeNode n) {
        if (n.left != null) helper(n.left);
        count--;
        if (count == 0) {
            number = n.val;
            return;
        }
        if (n.right != null) helper(n.right);
    }
```

DFS in-order iterative:

```
  public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> st = new Stack<>();

        while (root != null) {
            st.push(root);
            root = root.left;
        }

        while (k != 0) {
            TreeNode n = st.pop();
            k--;
            if (k == 0) return n.val;
            TreeNode right = n.right;
            while (right != null) {
                st.push(right);
                right = right.left;
            }
        }

        return -1; // never hit if k is valid
  }
```

written by angelvivienne original link here

Answer 2

Go inorder and decrease `k` at each node. Stop the whole search as soon as `k` is zero, and then the k-th element is immediately returned all the way to the recursion top and to the original caller.

Try the left subtree first. If that made `k` zero, then its answer is the overall answer and we return it right away. Otherwise, decrease `k` for the current node, and if that made `k` zero, then we return the current node's value right away. Otherwise try the right subtree and return whatever comes back from there.

```
int kthSmallest(TreeNode* root, int& k) {
    if (root) {
        int x = kthSmallest(root->left, k);
        return !k ? x : !--k ? root->val : kthSmallest(root->right, k);
    }
}
```

You might notice that I changed `k` from `int` to `int&` because I didn't feel like adding a helper just for that and the OJ doesn't mind. Oh well, here is that now:

```
int kthSmallest(TreeNode* root, int k) {
    return find(root, k);
}
int find(TreeNode* root, int& k) {
    if (root) {
        int x = find(root->left, k);
        return !k ? x : !--k ? root->val : find(root->right, k);
    }
}
```

written by StefanPochmann original link here

Answer 3

If we could add a count field in the BST node class, it will take O(n) time when we calculate the count value for the whole tree, but after that, it will take O(logn) time when insert/delete a node or calculate the kth smallest element.

```java
   public class Solution {
        public int kthSmallest(TreeNode root, int k) {
            TreeNodeWithCount rootWithCount = buildTreeWithCount(root);
            return kthSmallest(rootWithCount, k);
        }

        private TreeNodeWithCount buildTreeWithCount(TreeNode root) {
            if (root == null) return null;
            TreeNodeWithCount rootWithCount = new TreeNodeWithCount(root.val);
            rootWithCount.left = buildTreeWithCount(root.left);
            rootWithCount.right = buildTreeWithCount(root.right);
            if (rootWithCount.left != null) rootWithCount.count += rootWithCount.
left.count;
            if (rootWithCount.right != null) rootWithCount.count += rootWithCount
.right.count;
            return rootWithCount;
        }

        private int kthSmallest(TreeNodeWithCount rootWithCount, int k) {
            if (k <= 0 || k > rootWithCount.count) return -1;
            if (rootWithCount.left != null) {
                if (rootWithCount.left.count >= k) return kthSmallest(rootWithCou
nt.left, k);
                if (rootWithCount.left.count == k-1) return rootWithCount.val;
                return kthSmallest(rootWithCount.right, k-1-rootWithCount.left.co
unt);
            } else {
                if (k == 1) return rootWithCount.val;
                return kthSmallest(rootWithCount.right, k-1);
            }
        }

        class TreeNodeWithCount {
            int val;
            int count;
            TreeNodeWithCount left;
            TreeNodeWithCount right;
            TreeNodeWithCount(int x) {val = x; count = 1;};
        }
    }
```

written by WHJ425 original link here

Power of Two(231)

Answer 1

Power of 2 means only one bit of n is '1', so use the trick n&(n-1)==0 to judge whether that is the case

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        if(n<=0) return false;
        return !(n&(n-1));
    }
};
```

written by dong.wang.1694 original link here

Answer 2

This is kind of cheating, but the idea is that a power of two in binary form has and only has one "1".

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        return n>0 && Integer.bitCount(n) == 1;
    }
}
```

written by qxx original link here

Answer 3

I used n&(n-1), but, attention the 0.

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        return n > 0 && !(n&(n-1));
    }
};
```

written by iamshanshanpt original link here

Implement Queue using Stacks(232)

Answer 1

I have one input stack, onto which I push the incoming elements, and one output stack, from which I peek/pop. I move elements from input stack to output stack when needed, i.e., when I need to peek/pop but the output stack is empty. When that happens, I move all elements from input to output stack, thereby reversing the order so it's the correct order for peek/pop.

The loop in `peek` does the moving from input to output stack. Each element only ever gets moved like that once, though, and only after we already spent time pushing it, so the overall amortized cost for each operation is O(1).

Ruby

```ruby
class Queue
    def initialize
        @in, @out = [], []
    end

    def push(x)
        @in << x
    end

    def pop
        peek
        @out.pop
    end

    def peek
        @out << @in.pop until @in.empty? if @out.empty?
        @out.last
    end

    def empty
        @in.empty? && @out.empty?
    end
end
```

Java

```
class MyQueue {

    Stack<Integer> input = new Stack();
    Stack<Integer> output = new Stack();

    public void push(int x) {
        input.push(x);
    }

    public void pop() {
        peek();
        output.pop();
    }

    public int peek() {
        if (output.empty())
            while (!input.empty())
                output.push(input.pop());
        return output.peek();
    }

    public boolean empty() {
        return input.empty() && output.empty();
    }
}
```

C++

```
class Queue {
    stack<int> input, output;
public:

    void push(int x) {
        input.push(x);
    }

    void pop(void) {
        peek();
        output.pop();
    }

    int peek(void) {
        if (output.empty())
            while (input.size())
                output.push(input.top()), input.pop();
        return output.top();
    }

    bool empty(void) {
        return input.empty() && output.empty();
    }
};
```

written by StefanPochmann original link here

Answer 2

```
class MyQueue {

  Stack<Integer> s1 = new Stack();
  Stack<Integer> s2 = new Stack();

  // Push element x to the back of queue.
  public void push(int x) {
    while (!s2.isEmpty())
      s1.push(s2.pop());

    s1.push(x);
  }

  // Removes the element from in front of queue.
  public void pop() {
    while (!s1.isEmpty())
      s2.push(s1.pop());

    s2.pop();
  }

  // Get the front element.
  public int peek() {
    while (!s1.isEmpty())
      s2.push(s1.pop());

    return s2.peek();
  }

  // Return whether the queue is empty.
  public boolean empty() {
    return s1.isEmpty() && s2.isEmpty();
  }

}
```

written by jeantimex original link here

Answer 3

```java
class MyQueue {
    Stack<Integer> pushStack = new Stack<>();
    Stack<Integer> popStack = new Stack<>();

    // Push element x to the back of queue.
    public void push(int x) {
        pushStack.push(x);
    }

    // Removes the element from in front of queue.
    public void pop() {
        if(popStack.isEmpty()) {
            while(!pushStack.isEmpty()) {
                popStack.push(pushStack.pop());
            }
        }
        popStack.pop();
    }

    // Get the front element.
    public int peek() {
        if(popStack.isEmpty()) {
            while(!pushStack.isEmpty()) {
                popStack.push(pushStack.pop());
            }
        }
        return popStack.peek();
    }

    // Return whether the queue is empty.
    public boolean empty() {
        return pushStack.isEmpty() && popStack.isEmpty();
    }
}
```

written by tuan.huu.minh.nguyen original link here

Number of Digit One(233)

Answer 1

Go through the digit positions one at a time, find out how often a "1" appears at each position, and sum those up.

C++ solution

```
int countDigitOne(int n) {
    int ones = 0;
    for (long long m = 1; m <= n; m *= 10)
        ones += (n/m + 8) / 10 * m + (n/m % 10 == 1) * (n%m + 1);
    return ones;
}
```

Explanation

Let me use variables `a` and `b` to make the explanation a bit nicer.

```
int countDigitOne(int n) {
    int ones = 0;
    for (long long m = 1; m <= n; m *= 10) {
        int a = n/m, b = n%m;
        ones += (a + 8) / 10 * m + (a % 10 == 1) * (b + 1);
    }
    return ones;
}
```

Go through the digit positions by using position multiplier `m` with values 1, 10, 100, 1000, etc.

For each position, split the decimal representation into two parts, for example split n=3141592 into a=31415 and b=92 when we're at m=100 for analyzing the hundreds-digit. And then we know that the hundreds-digit of n is 1 for prefixes "" to "3141", i.e., 3142 times. Each of those times is a streak, though. Because it's the hundreds-digit, each streak is 100 long. So `(a / 10 + 1) * 100` times, the hundreds-digit is 1.

Consider the thousands-digit, i.e., when m=1000. Then a=3141 and b=592. The thousands-digit is 1 for prefixes "" to "314", so 315 times. And each time is a streak of 1000 numbers. However, since the thousands-digit is a 1, the very last streak isn't 1000 numbers but only 593 numbers, for the suffixes "000" to "592". So `(a / 10 * 1000) + (b + 1)` times, the thousands-digit is 1.

The case distincton between the current digit/position being 0, 1 and >=2 can easily be done in one expression. With `(a + 8) / 10` you get the number of full streaks, and `a % 10 == 1` tells you whether to add a partial streak.

Java version

```java
public int countDigitOne(int n) {
    int ones = 0;
    for (long m = 1; m <= n; m *= 10)
        ones += (n/m + 8) / 10 * m + (n/m % 10 == 1 ? n%m + 1 : 0);
    return ones;
}
```

Python version

```python
def countDigitOne(self, n):
    ones, m = 0, 1
    while m <= n:
        ones += (n/m + 8) / 10 * m + (n/m % 10 == 1) * (n%m + 1)
        m *= 10
    return ones
```

Using `sum` or recursion it can also be a one-liner.

Old solution

Go through the digit positions from back to front. I found it ugly to explain, so I made up that above new solution instead. The `n` here is the new solution's `a`, and the `r` here is the new solution's `b+1`.

Python

```python
def countDigitOne(self, n):
    ones = 0
    m = r = 1
    while n > 0:
        ones += (n + 8) / 10 * m + (n % 10 == 1) * r
        r += n % 10 * m
        m *= 10
        n /= 10
    return ones
```

Java

```java
public int countDigitOne(int n) {
    int ones = 0, m = 1, r = 1;
    while (n > 0) {
        ones += (n + 8) / 10 * m + (n % 10 == 1 ? r : 0);
        r += n % 10 * m;
        m *= 10;
        n /= 10;
    }
    return ones;
}
```

C++

```
int countDigitOne(int n) {
    int ones = 0, m = 1, r = 1;
    while (n > 0) {
        ones += (n + 8) / 10 * m + (n % 10 == 1) * r;
        r += n % 10 * m;
        m *= 10;
        n /= 10;
    }
    return ones;
}
```

written by StefanPochmann original link here

Answer 2

```
public int countDigitOne(int n) {
  int count = 0;

  for (long k = 1; k <= n; k *= 10) {
    long r = n / k, m = n % k;
    // sum up the count of ones on every place k
    count += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
  }

  return count;
}
```

Solution explanation:

Let's start by counting the ones for every 10 numbers...

0, 1, 2, 3 ... 9 (1)

10, 11, 12, 13 ... 19 (1) + **10**

20, 21, 22, 23 ... 29 (1)

...

90, 91, 92, 93 ... 99 (1)

-

100, 101, 102, 103 ... 109 (10 + 1)

110, 111, 112, 113 ... 119 (10 + 1) + **10**

120, 121, 122, 123 ... 129 (10 + 1)

...

190, 191, 192, 193 ... 199 (10 + 1)

-

1). If we don't look at those special rows (start with 10, 110 etc), we know that there's a one at ones' place in every 10 numbers, there are 10 ones at tens' place in

every 100 numbers, and 100 ones at hundreds' place in every 1000 numbers, so on and so forth.

Ok, let's start with ones' place and count how many ones at this place, set k = 1, as mentioned above, there's a one at ones' place in every 10 numbers, so how many 10 numbers do we have?

The answer is (n / k) / 10.

Now let's count the ones in tens' place, set k = 10, as mentioned above, there are 10 ones at tens' place in every 100 numbers, so how many 100 numbers do we have?

The answer is (n / k) / 10, and the number of ones at ten's place is (n / k) / 10 * k.

Let r = n / k, now we have a formula to count the ones at k's place**r / 10 * k**

-

2). So far, everything looks good, but we need to fix those special rows, how?

We can use the mod. Take 10, 11, and 12 for example, if n is 10, we get (n / 1) / 10 * 1 = 1 ones at ones's place, perfect, but for tens' place, we get (n / 10) / 10 * 10 = 0, that's not right, there should be a one at tens' place! Calm down, from 10 to 19, we always have a one at tens's place, let m = n % k, the number of ones at this special place is m + 1, so let's fix the formula to be:

r / 10 * k + (r % 10 == 1 ? m + 1 : 0)

-

3). Wait, how about 20, 21 and 22?

Let's say 20, use the above formula we get 0 ones at tens' place, but it should be 10! How to fix it? We know that once the digit is larger than 2, we should add 10 more ones to the tens' place, a clever way to fix is to add 8 to r, so our final formula is:

(r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0)

As you can see, it's all about how we fix the formula. Really hope that makes sense to you.

written by jeantimex original link here

Answer 3

For every digit in n (Suppose n = 240315, the digits are 2, 4, 0, 3, 1, 5)ï¼ŒI respectively count the number of digit 1 assuming the position of current digit is 1 and other digits of n is arbitrary.

For example, I select 3 in n as the current digit, and I suppose the position of 3 is 1.

The highn is the number composed with the digits before the current digit. In the example, highn = 240;

The lown is the number composed with the digits after the current digit. In the example, lown = 15.

The lowc = 10 ^ (the number of lower digits). In the example, lowc = 100;

As curn = 3 and curn > 1, (highn * 10 + 1) must be less than (highn * 10 + curn). Then the higher part can be 0 ~ highn, the lower part can be 0 ~ (lowc-1), and the current result = (highn + 1) * lowc.

```
int countDigitOne(int n) {
        long long int res(0);
        int highn(n), lowc(1), lown(0);
        while(highn > 0){
            int curn = highn % 10;
            highn = highn / 10;
            if(1 == curn){
                //higher: 0~(highn-1);  lower:  0 ~ (lowc-1)
                res += highn * lowc;
                //higher: highn ~ highn;     lower:0~lown
                res += lown + 1;
            }else if(0 == curn){
                //curn < 1
               //higher: 0~(highn-1);  lower:  0 ~ (lowc-1)
                res += highn * lowc;
            }else{
                //curn > 1
                res += (highn + 1) * lowc;
            }
            //update lown and lowc
            lown = curn * lowc + lown;
            lowc = lowc * 10;
        }
        return res;
    }
```

written by tju_xu original link here

Palindrome Linked List(234)

Answer 1

It is a common misunderstanding that the space complexity of a program is just how much the size of additional memory space being used besides input. An important prerequisite is neglected the above definition: the input has to be read-only. By definition, changing the input and change it back is not allowed (or the input size should be counted when doing so). Another way of determining the space complexity of a program is to simply look at how much space it has written to. Reversing a singly linked list requires writing to O(n) memory space, thus the space complexities for all "reverse-the-list"-based approaches are O(n), not O(1).

Solving this problem in O(1) space is theoretically impossible due to two simple facts: (1) a program using O(1) space is computationally equivalent to a finite automata, or a regular expression checker; (2) the pumping lemma states that the set of palindrome strings does not form a regular set.

Please change the incorrect problem statement.

written by wangmenghui original link here

Answer 2

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        if(head==NULL||head->next==NULL)
            return true;
        ListNode* slow=head;
        ListNode* fast=head;
        while(fast->next!=NULL&&fast->next->next!=NULL){
            slow=slow->next;
            fast=fast->next->next;
        }
        slow->next=reverseList(slow->next);
        slow=slow->next;
        while(slow!=NULL){
            if(head->val!=slow->val)
                return false;
            head=head->next;
            slow=slow->next;
        }
        return true;
    }
    ListNode* reverseList(ListNode* head) {
        ListNode* pre=NULL;
        ListNode* next=NULL;
        while(head!=NULL){
            next=head->next;
            head->next=pre;
            pre=head;
            head=next;
        }
        return pre;
    }
};
```

written by YCG09 original link here

Answer 3

O(n) time, O(1) space. The second solution restores the list after changing it.

Solution 1: *Reversed first half == Second half?*

Phase 1: Reverse the first half while finding the middle.
Phase 2: Compare the reversed first half with the second half.

```
def isPalindrome(self, head):
    rev = None
    slow = fast = head
    while fast and fast.next:
        fast = fast.next.next
        rev, rev.next, slow = slow, rev, slow.next
    if fast:
        slow = slow.next
    while rev and rev.val == slow.val:
        slow = slow.next
        rev = rev.next
    return not rev
```

Solution 2: *Play Nice*

Same as the above, but while comparing the two halves, restore the list to its original state by reversing the first half back. Not that the OJ or anyone else cares.

```
def isPalindrome(self, head):
    rev = None
    fast = head
    while fast and fast.next:
        fast = fast.next.next
        rev, rev.next, head = head, rev, head.next
    tail = head.next if fast else head
    isPali = True
    while rev:
        isPali = isPali and rev.val == tail.val
        head, head.next, rev = rev, head, rev.next
        tail = tail.next
    return isPali
```

written by StefanPochmann original link here

Lowest Common Ancestor of a Binary Search Tree(235)

Answer 1

Just walk down from the whole tree's root as long as both p and q are in the same subtree (meaning their values are both smaller or both larger than root's). This walks straight from the root to the LCA, not looking at the rest of the tree, so it's pretty much as fast as it gets. A few ways to do it:

Iterative, O(1) space

Python

```
def lowestCommonAncestor(self, root, p, q):
    while (root.val - p.val) * (root.val - q.val) > 0:
        root = (root.left, root.right)[p.val > root.val]
    return root
```

Java

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    while ((root.val - p.val) * (root.val - q.val) > 0)
        root = p.val < root.val ? root.left : root.right;
    return root;
}
```

(in case of overflow, I'd do `(root.val - (long)p.val) * (root.val - (long)q.val)`)

Different Python

```
def lowestCommonAncestor(self, root, p, q):
    a, b = sorted([p.val, q.val])
    while not a <= root.val <= b:
        root = (root.left, root.right)[a > root.val]
    return root
```

"Long" Python, maybe easiest to understand

```
def lowestCommonAncestor(self, root, p, q):
    while root:
        if p.val < root.val > q.val:
            root = root.left
        elif p.val > root.val < q.val:
            root = root.right
        else:
            return root
```

Recursive

Python

```
def lowestCommonAncestor(self, root, p, q):
    next = p.val < root.val > q.val and root.left or \
           p.val > root.val < q.val and root.right
    return self.lowestCommonAncestor(next, p, q) if next else root
```

Python One-Liner

```
def lowestCommonAncestor(self, root, p, q):
    return root if (root.val - p.val) * (root.val - q.val) < 1 else \
           self.lowestCommonAncestor((root.left, root.right)[p.val > root.val], p
, q)
```

Java One-Liner

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    return (root.val - p.val) * (root.val - q.val) < 1 ? root :
           lowestCommonAncestor(p.val < root.val ? root.left : root.right, p, q);
}
```

"Long" Python, maybe easiest to understand

```
def lowestCommonAncestor(self, root, p, q):
    if p.val < root.val > q.val:
        return self.lowestCommonAncestor(root.left, p, q)
    if p.val > root.val < q.val:
        return self.lowestCommonAncestor(root.right, p, q)
    return root
```

written by StefanPochmann original link here

Answer 2

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root.val > p.val && root.val > q.val){
            return lowestCommonAncestor(root.left, p, q);
        }else if(root.val < p.val && root.val < q.val){
            return lowestCommonAncestor(root.right, p, q);
        }else{
            return root;
        }
    }
}
```

written by jingzhetian original link here

Answer 3

Well, remember to take advantage of the property of binary search trees, which is, `node -> left -> val < node -> val < node -> right -> val`. Moreover,

both `p` and `q` will be the descendants of the `root` of the subtree that contains both of them. And the `root` with the largest depth is just the lowest common ancestor. This idea can be turned into the following simple recursive code.

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (p -> val < root -> val && q -> val < root -> val)
            return lowestCommonAncestor(root -> left, p, q);
        if (p -> val > root -> val && q -> val > root -> val)
            return lowestCommonAncestor(root -> right, p, q);
        return root;
    }
};
```

Of course, we can also solve it iteratively.

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        TreeNode* cur = root;
        while (true) {
            if (p -> val < cur -> val && q -> val < cur -> val)
                cur = cur -> left;
            else if (p -> val > cur -> val && q -> val > cur -> val)
                cur = cur -> right;
            else return cur;
        }
    }
};
```

written by jianchao.li.fighter original link here

Lowest Common Ancestor of a Binary Tree(236)

Answer 1

Same solution in several languages. It's recursive and expands the meaning of the function. If the current (sub)tree contains both p and q, then the function result is their LCA. If only one of them is in that subtree, then the result is that one of them. If neither are in that subtree, the result is null/None/nil.

Update: I also wrote two iterative solutions now, one of them being a version of the solution here. They're more complicated than this simple recursive solution, but I do find them interesting.

C++

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || root == p || root == q) return root;
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);
    return !left ? right : !right ? left : root;
}
```

Python

```
def lowestCommonAncestor(self, root, p, q):
    if root in (None, p, q): return root
    left, right = (self.lowestCommonAncestor(kid, p, q)
                   for kid in (root.left, root.right))
    return root if left and right else left or right
```

Or using that `None` is considered smaller than any node:

```
def lowestCommonAncestor(self, root, p, q):
    if root in (None, p, q): return root
    subs = [self.lowestCommonAncestor(kid, p, q)
            for kid in (root.left, root.right)]
    return root if all(subs) else max(subs)
```

Ruby

```
def lowest_common_ancestor(root, p, q)
    return root if [nil, p, q].index root
    left = lowest_common_ancestor(root.left, p, q)
    right = lowest_common_ancestor(root.right, p, q)
    left && right ? root : left || right
end
```

Java

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    return left == null ? right : right == null ? left : root;
}
```

written by StefanPochmann original link here

Answer 2

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null || root == p || root == q)  return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left != null && right != null)   return root;
        return left != null ? left : right;
    }
}
```

written by yuhangjiang original link here

Answer 3

```
class Solution {
public:
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *p, TreeNode *q) {
        if (root == p || root == q || root == NULL) return root;
        TreeNode *left = lowestCommonAncestor(root->left, p, q), *right = lowestC
ommonAncestor(root->right, p, q);
        return left && right ? root : left ? left : right;
    }
};
```

written by prime_tang original link here

Delete Node in a Linked List(237)

Answer 1

We can't really delete the node, but we can kinda achieve the same effect by instead removing the **next** node after copying its data into the node that we were asked to delete.

C++

```
void deleteNode(ListNode* node) {
    *node = *node->next;
}
```

But better properly delete the next node:

```
void deleteNode(ListNode* node) {
    auto next = node->next;
    *node = *next;
    delete next;
}
```

Java and C#

```
public void deleteNode(ListNode node) {
    node.val = node.next.val;
    node.next = node.next.next;
}
```

Python

```
def deleteNode(self, node):
    node.val = node.next.val
    node.next = node.next.next
```

C

```
void deleteNode(struct ListNode* node) {
    *node = *node->next;
}
```

But better properly free the next node's memory:

```
void deleteNode(struct ListNode* node) {
    struct ListNode* next = node->next;
    *node = *next;
    free(next);
}
```

JavaScript

```
var deleteNode = function(node) {
    node.val = node.next.val;
    node.next = node.next.next;
};
```

Ruby

```
def delete_node(node)
    node.val = node.next.val
    node.next = node.next.next
    nil
end
```

written by StefanPochmann original link here

Answer 2

This question is stupid and should be deleted intermediately.

written by smfwuxiao original link here

Answer 3

```
public class Solution {
    public void deleteNode(ListNode node) {
        if(node != null && node.next != null) {
            node.val = node.next.val;
            node.next = node.next.next;
        }
    }
}
```

written by zwangbo original link here

Product of Array Except Self(238)

Answer 1

```
public class Solution {
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] res = new int[n];
    res[0] = 1;
    for (int i = 1; i < n; i++) {
        res[i] = res[i - 1] * nums[i - 1];
    }
    int right = 1;
    for (int i = n - 1; i >= 0; i--) {
        res[i] *= right;
        right *= nums[i];
    }
    return res;
}
```

}

written by lycjava3 original link here

Answer 2

Use `tmp` to store temporary multiply result by two directions. Then fill it into `result`. Bingo!

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];
    for (int i = 0, tmp = 1; i < nums.length; i++) {
        result[i] = tmp;
        tmp *= nums[i];
    }
    for (int i = nums.length - 1, tmp = 1; i >= 0; i--) {
        result[i] *= tmp;
        tmp *= nums[i];
    }
    return result;
}
```

written by xcv58 original link here

Answer 3

First, consider O(n) time and O(n) space solution.

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n=nums.size();
        vector<int> fromBegin(n);
        fromBegin[0]=1;
        vector<int> fromLast(n);
        fromLast[0]=1;

        for(int i=1;i<n;i++){
            fromBegin[i]=fromBegin[i-1]*nums[i-1];
            fromLast[i]=fromLast[i-1]*nums[n-i];
        }

        vector<int> res(n);
        for(int i=0;i<n;i++){
            res[i]=fromBegin[i]*fromLast[n-1-i];
        }
        return res;
    }
};
```

We just need to change the two vectors to two integers and note that we should do multiplying operations for two related elements of the results vector in each loop.

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n=nums.size();
        int fromBegin=1;
        int fromLast=1;
        vector<int> res(n,1);

        for(int i=0;i<n;i++){
            res[i]*=fromBegin;
            fromBegin*=nums[i];
            res[n-1-i]*=fromLast;
            fromLast*=nums[n-1-i];
        }
        return res;
    }
};
```

written by zhaoqiang original link here

Sliding Window Maximum(239)

Answer 1

We scan the array from 0 to n-1, keep "promising" elements in the deque. The algorithm is amortized O(n) as each element is put and polled once.

At each i, we keep "promising" elements, which are potentially max number in window [i-(k-1),i] or any subsequent window. This means

1. If an element in the deque and it is out of i-(k-1), we discard them. We just need to poll from the head, as we are using a deque and elements are ordered as the sequence in the array
2. Now only those elements within [i-(k-1),i] are in the deque. We then discard elements smaller than a[i] from the tail. This is because if a[x] <a[i] and x<i, then a[x] has no chance to be the "max" in [i-(k-1),i], or any other subsequent window: a[i] would always be a better candidate.
3. As a result elements in the deque are ordered in both sequence in array and their value. At each step the head of the deque is the max element in [i-(k-1),i]

```
public int[] maxSlidingWindow(int[] a, int k) {
        if (a == null || k <= 0) {
            return new int[0];
        }
        int n = a.length;
        int[] r = new int[n-k+1];
        int ri = 0;
        // store index
        Deque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < a.length; i++) {
            // remove numbers out of range k
            while (!q.isEmpty() && q.peek() < i - k + 1) {
                q.poll();
            }
            // remove smaller numbers in k range as they are useless
            while (!q.isEmpty() && a[q.peekLast()] < a[i]) {
                q.pollLast();
            }
            // q contains index... r contains content
            q.offer(i);
            if (i >= k - 1) {
                r[ri++] = a[q.peek()];
            }
        }
        return r;
    }
```

written by zjm84812 original link here

Answer 2

The data structure used is know as Monotonic Queue. Clickhere for more information.

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        vector<int> ans;
        for (int i=0; i<nums.size(); i++) {
            if (!dq.empty() && dq.front() == i-k) dq.pop_front();
            while (!dq.empty() && nums[dq.back()] < nums[i])
                dq.pop_back();
            dq.push_back(i);
            if (i>=k-1) ans.push_back(nums[dq.front()]);
        }
        return ans;
    }
};
```

written by zhaotianzju original link here

Answer 3

Sliding window minimum/maximum = monotonic queue. I smelled the solution just when I read the title. This is essentially same idea as others' deque solution, but this is much more standardized and modulized. If you ever need to use it in your real product, this code is definitely more preferable.

What does Monoqueue do here:

It has three basic options:

push: push an element into the queue; O (1) (amortized)

pop: pop an element out of the queue; O(1) (pop = remove, it can't report this element)

max: report the max element in queue;O(1)

It takes only O(n) time to process a N-size sliding window minimum/maximum problem.

Note: different from a priority queue (which takes O(nlogk) to solve this problem), it doesn't pop the max element: It pops the first element (in original order) in queue.

```
class Monoqueue
{
    deque<pair<int, int>> m_deque; //pair.first: the actual value,
                                   //pair.second: how many elements were deleted
between it and the one before it.
    public:
        void push(int val)
        {
            int count = 0;
            while(!m_deque.empty() && m_deque.back().first < val)
            {
                count += m_deque.back().second + 1;
                m_deque.pop_back();
            }
            m_deque.emplace_back(val, count);
        };
        int max()
        {
            return m_deque.front().first;
        }
        void pop ()
        {
            if (m_deque.front().second > 0)
            {
                m_deque.front().second --;
                return;
            }
            m_deque.pop_front();
        }
};
struct Solution {
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> results;
        Monoqueue mq;
        k = min(k, (int)nums.size());
        int i = 0;
        for (;i < k - 1; ++i) //push first k - 1 numbers;
        {
            mq.push(nums[i]);
        }
        for (; i < nums.size(); ++i)
        {
            mq.push(nums[i]);            // push a new element to queue;
            results.push_back(mq.max()); // report the current max in queue;
            mq.pop();                    // pop first element in queue;
        }
        return results;
    }
};
```

written by fentoyal original link here

Search a 2D Matrix II(240)

Answer 1

We start search the matrix from top right corner, initialize the current position to top right corner, if the target is greater than the value in current position, then the target can not be in entire row of current position because the row is sorted, if the target is less than the value in current position, then the target can not in the entire column because the column is sorted too. We can rule out one row or one column each time, so the time complexity is O(m+n).

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length < 1 || matrix[0].length <1) {
            return false;
        }
        int col = matrix[0].length-1;
        int row = 0;
        while(col >= 0 && row <= matrix.length-1) {
            if(target == matrix[row][col]) {
                return true;
            } else if(target < matrix[row][col]) {
                col--;
            } else if(target > matrix[row][col]) {
                row++;
            }
        }
        return false;
    }
}
```

written by chicm original link here

Answer 2

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size();
    if (m == 0) return false;
    int n = matrix[0].size();

    int i = 0, j = n - 1;
    while (i < m && j >= 0) {
        if (matrix[i][j] == target)
            return true;
        else if (matrix[i][j] > target) {
            j--;
        } else
            i++;
    }
    return false;
}
```

written by loki2441 original link here

Answer 3

1. O(m+n) solution

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
int i = 0;
int j = matrix[0].size() - 1;

while(i < matrix.size() && j >= 0) {
    if(matrix[i][j] == target)
        return true;

    if(matrix[i][j] < target)
        i++;
    else
        j--;
}

return false;
}
```

2. O(mlogn) solution

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
return searchMatrix(matrix, target, 0, matrix.size() - 1);
}

bool searchMatrix(vector<vector<int>>& matrix, int target, int top, int bott
om) {
if (top > bottom)
    return false;

int mid = top + (bottom - top) / 2;
if (matrix[mid].front() <= target && target <= matrix[mid].back())
    if (searchVector(matrix[mid], target)) return true;

if (searchMatrix(matrix, target, top, mid - 1)) return true;
if (searchMatrix(matrix, target, mid + 1, bottom)) return true;

return false;
}

bool searchVector(vector<int>& v, int target) {
int left = 0, right = v.size() - 1;

while (left <= right) {
    int mid = left + (right - left) / 2;
    if (v[mid] == target)
        return true;
    if (v[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
}

return false;
}
```

written by jaewoo original link here

Different Ways to Add Parentheses(241)

Answer 1

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        List<Integer> ret = new LinkedList<Integer>();
        for (int i=0; i<input.length(); i++) {
            if (input.charAt(i) == '-' ||
                input.charAt(i) == '*' ||
                input.charAt(i) == '+' ) {
                String part1 = input.substring(0, i);
                String part2 = input.substring(i+1);
                List<Integer> part1Ret = diffWaysToCompute(part1);
                List<Integer> part2Ret = diffWaysToCompute(part2);
                for (Integer p1 :   part1Ret) {
                    for (Integer p2 :   part2Ret) {
                        int c = 0;
                        switch (input.charAt(i)) {
                            case '+': c = p1+p2;
                                break;
                            case '-': c = p1-p2;
                                break;
                            case '*': c = p1*p2;
                                break;
                        }
                        ret.add(c);
                    }
                }
            }
        }
        if (ret.size() == 0) {
            ret.add(Integer.valueOf(input));
        }
        return ret;
    }
}
```

written by 2guotou original link here

Answer 2

Here is the basic recursive solution

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
        vector<int> result;
        int size = input.size();
        for (int i = 0; i < size; i++) {
            char cur = input[i];
            if (cur == '+' || cur == '-' || cur == '*') {
                // Split input string into two parts and solve them recursively
                vector<int> result1 = diffWaysToCompute(input.substr(0, i));
                vector<int> result2 = diffWaysToCompute(input.substr(i+1));
                for (auto n1 : result1) {
                    for (auto n2 : result2) {
                        if (cur == '+')
                            result.push_back(n1 + n2);
                        else if (cur == '-')
                            result.push_back(n1 - n2);
                        else
                            result.push_back(n1 * n2);
                    }
                }
            }
        }
        // if the input string contains only number
        if (result.empty())
            result.push_back(atoi(input.c_str()));
        return result;
    }
};
```

There are many repeating subquestions in this recursive method, therefore, we could use dynamic programming to avoid this situation by saving the results for subquestions. Here is the DP solution.

```
class Solution {
public:
    vector<int> diffWaysToCompute(string input) {
        unordered_map<string, vector<int>> dpMap;
        return computeWithDP(input, dpMap);
    }

    vector<int> computeWithDP(string input, unordered_map<string, vector<int>> &d
pMap) {
        vector<int> result;
        int size = input.size();
        for (int i = 0; i < size; i++) {
            char cur = input[i];
            if (cur == '+' || cur == '-' || cur == '*') {
                // Split input string into two parts and solve them recursively
                vector<int> result1, result2;
                string substr = input.substr(0, i);
                // check if dpMap has the result for substr
                if (dpMap.find(substr) != dpMap.end())
                    result1 = dpMap[substr];
                else
                    result1 = computeWithDP(substr, dpMap);

                substr = input.substr(i + 1);
                if (dpMap.find(substr) != dpMap.end())
                    result2 = dpMap[substr];
                else
                    result2 = computeWithDP(substr, dpMap);

                for (auto n1 : result1) {
                    for (auto n2 : result2) {
                        if (cur == '+')
                            result.push_back(n1 + n2);
                        else if (cur == '-')
                            result.push_back(n1 - n2);
                        else
                            result.push_back(n1 * n2);
                    }
                }
            }
        }
        // if the input string contains only number
        if (result.empty())
            result.push_back(atoi(input.c_str()));
        // save to dpMap
        dpMap[input] = result;
        return result;
    }
};
```

written by Gcdofree original link here

Answer 3

Just doing it...

Solution 1 ... 48 ms

```
def diffWaysToCompute(self, input):
   tokens = re.split('(\D)', input)
   nums = map(int, tokens[::2])
   ops = map({'+': operator.add, '-': operator.sub, '*': operator.mul}.get, token
s[1::2])
   def build(lo, hi):
       if lo == hi:
           return [nums[lo]]
       return [ops[i](a, b)
               for i in xrange(lo, hi)
               for a in build(lo, i)
               for b in build(i + 1, hi)]
   return build(0, len(nums) - 1)
```

Solution 2 ... 168 ms

One-liner inspired by Soba.

```
def diffWaysToCompute(self, input):
    return [eval(`a`+c+`b`)
            for i, c in enumerate(input) if c in '+-*'
            for a in self.diffWaysToCompute(input[:i])
            for b in self.diffWaysToCompute(input[i+1:])] or [int(input)]
```

Solution 3 ... 64 ms

Faster version of solution 2.

```
def diffWaysToCompute(self, input):
    return [a+b if c == '+' else a-b if c == '-' else a*b
            for i, c in enumerate(input) if c in '+-*'
            for a in self.diffWaysToCompute(input[:i])
            for b in self.diffWaysToCompute(input[i+1:])] or [int(input)]
```

Solution 4 ... 188 ms

A code golf version of solution 2.

```
diffWaysToCompute=d=lambda s,t:[eval(`a`+c+`b`)for i,c in enumerate(t)if
c<'0'for a in s.d(t[:i])for b in s.d(t[i+1:])]or[int(t)]
```

C++ ... 8 ms

C++ version of solution 3.

```
vector<int> diffWaysToCompute(string input) {
    vector<int> output;
    for (int i=0; i<input.size(); i++) {
        char c = input[i];
        if (ispunct(c))
            for (int a : diffWaysToCompute(input.substr(0, i)))
                for (int b : diffWaysToCompute(input.substr(i+1)))
                    output.push_back(c=='+' ? a+b : c=='-' ? a-b : a*b);
    }
    return output.size() ? output : vector<int>{stoi(input)};
}
```

written by StefanPochmann original link here

Valid Anagram(242)

Answer 1

The idea is simple. It creates a size 26 int arrays as buckets for each letter in alphabet. It increments the bucket value with String s and decrement with string t. So if they are anagrams, all buckets should remain with initial value which is zero. So just checking that and return

```
public class Solution {
    public boolean isAnagram(String s, String t) {
        int[] alphabet = new int[26];
        for (int i = 0; i < s.length(); i++) alphabet[s.charAt(i) - 'a']++;
        for (int i = 0; i < t.length(); i++) alphabet[t.charAt(i) - 'a']--;
        for (int i : alphabet) if (i != 0) return false;
        return true;
    }
}
```

written by vimukthi original link here

Answer 2

```
    public class Solution {
    public boolean isAnagram(String s, String t) {
        if(s.length()!=t.length()){
            return false;
        }
        int[] count = new int[26];
        for(int i=0;i<s.length();i++){
            count[s.charAt(i)-'a']++;
            count[t.charAt(i)-'a']--;
        }
        for(int i:count){
            if(i!=0){
                return false;
            }
        }
        return true;
    }
}
```

written by abner original link here

Answer 3

```java
public boolean isAnagram(String s, String t) {
        if(s == null || t == null || s.length() != t.length()) return false;
        int[] count = new int[26];
        int len = t.length();
        for(int i = 0; i < len; i++) {
            count[t.charAt(i) - 'a']++;
        }
        for(int i = 0; i < len; i++) {
            char c = s.charAt(i);
            if(count[c - 'a'] > 0) {
                count[c - 'a']--;
            } else {
                return false;
            }
        }
        return true;
    }
```

written by yzxu1101 original link here

Shortest Word Distance(243)

Answer 1

```
public int shortestDistance(String[] words, String word1, String word2) {
    int p1 = -1, p2 = -1, min = Integer.MAX_VALUE;

    for (int i = 0; i < words.length; i++) {
        if (words[i].equals(word1))
            p1 = i;

        if (words[i].equals(word2))
            p2 = i;

        if (p1 != -1 && p2 != -1)
            min = Math.min(min, Math.abs(p1 - p2));
    }

    return min;
}
```

written by jeantimex original link here

Answer 2

Creating two lists storing indexes of each occurrence of the `word1` and `word2` accordingly. After that finding minimum difference between two elements from these lists.

```
public class Solution {
        public int shortestDistance(String[] words, String word1, String word2) {
            List<Integer> w1occ=new ArrayList<Integer>();
            List<Integer> w2occ=new ArrayList<Integer>();

            for (int i=0; i<words.length; ++i){
                if (words[i].equals(word1)){
                    w1occ.add(i);
                }
                if (words[i].equals(word2)){
                    w2occ.add(i);
                }
            }

            int min=words.length;
            int p1=0;
            int p2=0;
            while (p1<w1occ.size() && p2<w2occ.size()){
                min=Math.min(Math.abs(w1occ.get(p1)-w2occ.get(p2)), min);
                if (w1occ.get(p1)<w2occ.get(p2)){
                    p1++;
                } else
                    p2++;
            }
            return min;
        }
    }
```

written by ammv original link here

Answer 3

```
public int shortestDistance(String[] words, String word1, String word2) {
   int index = -1, minDistance = Integer.MAX_VALUE;
   for (int i = 0; i < words.length; i++) {
      if (words[i].equals(word1) || words[i].equals(word2)) {
         if (index != -1 && !words[index].equals(words[i])) {
            minDistance = Math.min(minDistance, i - index);
          }
          index = i;
      }
   }
   return minDistance;
}
```

written by BIO2CS original link here

Shortest Word Distance II(244)

Answer 1

```
public class WordDistance {

private Map<String, List<Integer>> map;

public WordDistance(String[] words) {
    map = new HashMap<String, List<Integer>>();
    for(int i = 0; i < words.length; i++) {
        String w = words[i];
        if(map.containsKey(w)) {
            map.get(w).add(i);
        } else {
            List<Integer> list = new ArrayList<Integer>();
            list.add(i);
            map.put(w, list);
        }
    }
}

public int shortest(String word1, String word2) {
    List<Integer> list1 = map.get(word1);
    List<Integer> list2 = map.get(word2);
    int ret = Integer.MAX_VALUE;
    for(int i = 0, j = 0; i < list1.size() && j < list2.size(); ) {
        int index1 = list1.get(i), index2 = list2.get(j);
        if(index1 < index2) {
            ret = Math.min(ret, index2 - index1);
            i++;
        } else {
            ret = Math.min(ret, index1 - index2);
            j++;
        }
    }
    return ret;
}
}
```

written by qianzhige original link here

Answer 2

```
class WordDistance {
public:
    WordDistance(vector<string>& words) {
        for(int i=0;i<words.size();i++)
            wordMap[words[i]].push_back(i);
    }
    int shortest(string word1, string word2) {
        int  i=0, j=0, dist = INT_MAX;
        while(i < wordMap[word1].size() && j <wordMap[word2].size()) {
            dist = min(dist, abs(wordMap[word1][i] - wordMap[word2][j]));
            wordMap[word1][i]<wordMap[word2][j]?i++:j++;
        }
        return dist;
    }
private:
    unordered_map<string, vector<int>> wordMap;
};
```

written by luming89 original link here

Answer 3

```
class WordDistance {
public:
    WordDistance(vector<string>& words) {
        for(int i=0; i<words.size(); ++i)
        {
            mp[words[i]].push_back(i);
        }
    }

    int shortest(string word1, string word2) {
        vector<int> w1 = mp[word1];
        vector<int> w2 = mp[word2];
        int res=INT_MAX;
        for(int i=0; i<w1.size(); ++i)
            for(int j=0; j<w2.size(); ++j)
                res=(abs(w1[i]-w2[j])<res)? abs(w1[i]-w2[j]) : res;
        return res;
    }
private:
    unordered_map<string, vector<int>> mp;
};
```

written by wkawhi original link here

Shortest Word Distance III(245)

Answer 1

Solution 1 ... Java "short"

`i1` and `i2` are the indexes where word1 and word2 were last seen. Except if they're the same word, then `i1` is the previous index.

```
public int shortestWordDistance(String[] words, String word1, String word2) {
    long dist = Integer.MAX_VALUE, i1 = dist, i2 = -dist;
    for (int i=0; i<words.length; i++) {
        if (words[i].equals(word1))
            i1 = i;
        if (words[i].equals(word2)) {
            if (word1.equals(word2))
                i1 = i2;
            i2 = i;
        }
        dist = Math.min(dist, Math.abs(i1 - i2));
    }
    return (int) dist;
}
```

Solution 2 ... Java "fast"

Same as solution 1, but minimizing the number of string comparisons.

```
public int shortestWordDistance(String[] words, String word1, String word2) {
    long dist = Integer.MAX_VALUE, i1 = dist, i2 = -dist;
    boolean same = word1.equals(word2);
    for (int i=0; i<words.length; i++) {
        if (words[i].equals(word1)) {
            if (same) {
                i1 = i2;
                i2 = i;
            } else {
                i1 = i;
            }
        } else if (words[i].equals(word2)) {
            i2 = i;
        }
        dist = Math.min(dist, Math.abs(i1 - i2));
    }
    return (int) dist;
}
```

Solution 3 ... C++ "short"

C++ version of solution 1.

```
int shortestWordDistance(vector<string>& words, string word1, string word2) {
    long long dist = INT_MAX, i1 = dist, i2 = -dist;
    for (int i=0; i<words.size(); i++) {
        if (words[i] == word1)
            i1 = i;
        if (words[i] == word2) {
            if (word1 == word2)
                i1 = i2;
            i2 = i;
        }
        dist = min(dist, abs(i1 - i2));
    }
    return dist;
}
```

Solution 4 ... C++ "fast"

C++ version of solution 2.

```
int shortestWordDistance(vector<string>& words, string word1, string word2) {
    long long dist = INT_MAX, i1 = dist, i2 = -dist;
    bool same = word1 == word2;
    for (int i=0; i<words.size(); i++) {
        if (words[i] == word1) {
            i1 = i;
            if (same)
                swap(i1, i2);
        } else if (words[i] == word2) {
            i2 = i;
        }
        dist = min(dist, abs(i1 - i2));
    }
    return dist;
}
```

written by StefanPochmann original link here

Answer 2

```java
public class Solution {
    public int shortestWordDistance(String[] words, String word1, String word2) {
        int p1 = -1;
        int p2 = -1;
        int min = Integer.MAX_VALUE;
        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1)) {
                if (word1.equals(word2)) {
                    if (p1 != -1 && i - p1 < min) {
                        min = i - p1;
                    }
                    p1 = i;
                } else {
                    p1 = i;
                    if (p2 != -1 && p1 - p2 < min) {
                        min = p1 - p2;
                    }
                }
            } else if (words[i].equals(word2)) {
                p2 = i;
                if (p1 != -1 && p2 - p1 < min) {
                    min = p2 - p1;
                }
            }
        }
        return min;
    }
}
```

written by stevenye original link here

Answer 3

```java
public int shortestWordDistance(String[] words, String word1, String word2) {
    if (words.length < 2) {
        return 0;
    }
    int index1 = -1;
    int index2 = -1;
    int min = Integer.MAX_VALUE;

    for (int i = 0; i < words.length; i++) {
        if (words[i].equals(word1)) {
            index1 = i;
        }
        if (index1 != -1 && index2 != -1 && index1 != index2) {
            min = Math.min(min, Math.abs(index1 - index2));
        }
        if (words[i].equals(word2)) {
            index2 = i;
        }
        if (index1 != -1 && index2 != -1 && index1 != index2) {
            min = Math.min(min, Math.abs(index1 - index2));
        }
    }

    return min;
}
```

written by rjr130 original link here

Strobogrammatic Number(246)

Answer 1

Just checking the pairs, going inwards from the ends.

```
public boolean isStrobogrammatic(String num) {
    for (int i=0, j=num.length()-1; i <= j; i++, j--)
        if (!"00 11 88 696".contains(num.charAt(i) + "" + num.charAt(j)))
            return false;
    return true;
}
```

written by StefanPochmann original link here

Answer 2

```
public boolean isStrobogrammatic(String num) {
    Map<Character, Character> map = new HashMap<Character, Character>();
    map.put('6', '9');
    map.put('9', '6');
    map.put('0', '0');
    map.put('1', '1');
    map.put('8', '8');

    int l = 0, r = num.length() - 1;
    while (l <= r) {
        if (!map.containsKey(num.charAt(l))) return false;
        if (map.get(num.charAt(l)) != num.charAt(r))
            return false;
        l++;
        r--;
    }

    return true;
}
```

written by szn1992 original link here

Answer 3

```java
public class Solution {
    public boolean isStrobogrammatic(String num) {
        int start = 0;
        int end = num.length() - 1;
        while (start <= end) {
            switch(num.charAt(start)) {
                case '0':
                case '1':
                case '8':
                    if (num.charAt(end) != num.charAt(start)) {
                        return false;
                    }
                    break;
                case '6':
                    if (num.charAt(end) != '9') {
                        return false;
                    }
                    break;
                case '9':
                    if (num.charAt(end) != '6') {
                        return false;
                    }
                    break;
                default:
                    return false;
            }
            start++;
            end--;
        }
        return true;
    }
}
```

written by stevenye original link here

Strobogrammatic Number II(247)

Answer 1

```
public List<String> findStrobogrammatic(int n) {
    return helper(n, n);
}

List<String> helper(int n, int m) {
    if (n == 0) return new ArrayList<String>(Arrays.asList(""));
    if (n == 1) return new ArrayList<String>(Arrays.asList("0", "1", "8"));

    List<String> list = helper(n - 2, m);

    List<String> res = new ArrayList<String>();

    for (int i = 0; i < list.size(); i++) {
        String s = list.get(i);

        if (n != m) res.add("0" + s + "0");

        res.add("1" + s + "1");
        res.add("6" + s + "9");
        res.add("8" + s + "8");
        res.add("9" + s + "6");
    }

    return res;
}
```

written by jeantimex original link here

Answer 2

```java
public List<String> findStrobogrammatic(int n) {
    findStrobogrammaticHelper(new char[n], 0, n - 1);
    return res;
}

List<String> res = new ArrayList<String>();

public void findStrobogrammaticHelper(char[] a, int l, int r) {
    if (l > r) {
        res.add(new String(a));
        return;
    }
    if (l == r) {
        a[l] = '0'; res.add(new String(a));
        a[l] = '1'; res.add(new String(a));
        a[l] = '8'; res.add(new String(a));
        return;
    }

    if (l != 0) {
        a[l] = '0'; a[r] = '0';
        findStrobogrammaticHelper(a, l+1, r-1);
    }
    a[l] = '1'; a[r] = '1';
    findStrobogrammaticHelper(a, l+1, r-1);
    a[l] = '8'; a[r] = '8';
    findStrobogrammaticHelper(a, l+1, r-1);
    a[l] = '6'; a[r] = '9';
    findStrobogrammaticHelper(a, l+1, r-1);
    a[l] = '9'; a[r] = '6';
    findStrobogrammaticHelper(a, l+1, r-1);
}
```

written by szn1992 original link here

Answer 3

```java
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        Map<Character, Character> map = new HashMap<Character, Character>();
        map.put('0', '0');
        map.put('1', '1');
        map.put('6', '9');
        map.put('8', '8');
        map.put('9', '6');
        List<String> result = new ArrayList<String>();
        char[] buffer = new char[n];
        dfs(n, 0, buffer, result, map);
        return result;
    }

    private void dfs(int n, int index, char[] buffer, List<String> result, Map<Ch
aracter, Character> map) {
        if (n == 0) {
            return;
        }
        if (index == (n + 1) / 2) {
            result.add(String.valueOf(buffer));
            return;
        }
        for (Character c : map.keySet()) {
            if (index == 0 && n > 1 && c == '0') {  // first digit cannot be '0'
when n > 1
                continue;
            }
            if (index == n / 2 && (c == '6' || c == '9')) {   // mid digit cannot
be '6' or '9' when n is odd
                continue;
            }
            buffer[index] = c;
            buffer[n - 1 - index] = map.get(c);
            dfs(n, index + 1, buffer, result, map);
        }
    }
}
```

written by stevenye original link here

Strobogrammatic Number III(248)

Answer 1

The basic idea is to find generate a list of strobogrammatic number with the length between the length of lower bound and the length of upper bound. Then we pass the list and ignore the numbers with the same length of lower bound or upper bound but not in the range.

I think it is not a a very optimized method and can any one provide a better one?

```
public class Solution{


    public int strobogrammaticInRange(String low, String high){
        int count = 0;
        List<String> rst = new ArrayList<String>();
        for(int n = low.length(); n <= high.length(); n++){
            rst.addAll(helper(n, n));
        }
        for(String num : rst){

            if((num.length() == low.length()&&num.compareTo(low) < 0 ) ||(num.len
gth() == high.length() && num.compareTo(high) > 0)) continue;
                count++;
        }
        return count;
    }

    private List<String> helper(int cur, int max){
        if(cur == 0) return new ArrayList<String>(Arrays.asList(""));
        if(cur == 1) return new ArrayList<String>(Arrays.asList("1", "8", "0"));

        List<String> rst = new ArrayList<String>();
        List<String> center = helper(cur - 2, max);

        for(int i = 0; i < center.size(); i++){
            String tmp = center.get(i);
            if(cur != max) rst.add("0" + tmp + "0");
            rst.add("1" + tmp + "1");
            rst.add("6" + tmp + "9");
            rst.add("8" + tmp + "8");
            rst.add("9" + tmp + "6");
        }
        return rst;
    }
}
```

written by czonzhu original link here

Answer 2

My clear Java solution. All comments are welcome.

```
    public class Solution {
Map<Character, Character> map = new HashMap<>();
{
    map.put('1', '1');
    map.put('8', '8');
    map.put('6', '9');
    map.put('9', '6');
    map.put('0', '0');
}
String low = "", high = "";
public int strobogrammaticInRange(String low, String high) {
    this.low = low;
    this.high = high;
    int result = 0;
    for(int n = low.length(); n <= high.length(); n++){
        int[] count = new int[1];
        strobogrammaticInRange(new char[n], count, 0, n-1);
        result += count[0];
    }
    return result;
}
private void strobogrammaticInRange(char[] arr, int[] count, int lo, int hi){
    if(lo > hi){
        String s = new String(arr);
        if((arr[0] != '0' || arr.length == 1) && compare(low, s) && compare(s, hi
gh)){
            count[0]++;
        }
        return;
    }
    for(Character c: map.keySet()){
        arr[lo] = c;
        arr[hi] = map.get(c);
        if((lo == hi && c == map.get(c)) || lo < hi)
            strobogrammaticInRange(arr, count, lo+1, hi-1);
    }
}
private boolean compare(String a, String b){
    if(a.length() != b.length())
        return a.length() < b.length();
    int i = 0;
    while(i < a.length() &&a.charAt(i) == b.charAt(i))
        i++;
    return i == a.length() ? true: a.charAt(i) <= b.charAt(i);
}
```

}

written by hyuna915 original link here

Answer 3

```
bool compare(string s1, string s2) {
    if (s1.length() != s2.length())
        return s1.length() <= s2.length();

    for (int i = 0; i < s1.length(); i++) {
        if (s1[i] < s2[i]) return true;
        else if (s1[i] > s2[i]) return false;
    }

    return true;
}

int strobogrammaticInRange(const vector<pair<char, char>>& nums, const string& lo
w, const string& high, string t, int cnt) {
    if (high.length() < t.length())
        return cnt;
    if (compare(low, t) && compare(t, high))
        if (t.length() == 1 || t.length() > 1 && t.front() != '0')
            cnt++;

    for (auto iter = nums.begin(); iter != nums.end(); ++iter)
        cnt = strobogrammaticInRange(nums, low, high, string(1, iter->first) + t
+ string(1, iter->second), cnt);

    return cnt;
}

int strobogrammaticInRange(string low, string high) {
    if (!compare(low, high)) return 0;

    vector<pair<char, char>> nums = { { '0', '0' }, { '1', '1' }, { '6', '9' }, {
'8', '8' }, { '9', '6' } };

    int cnt = strobogrammaticInRange(nums, low, high, "", 0);
    cnt = strobogrammaticInRange(nums, low, high, "0", cnt);
    cnt = strobogrammaticInRange(nums, low, high, "1", cnt);
    cnt = strobogrammaticInRange(nums, low, high, "8", cnt);

    return cnt;
}
```

written by jaewoo original link here

Group Shifted Strings(249)

Answer 1

```
public class Solution {
    public List<List<String>> groupStrings(String[] strings) {
        List<List<String>> result = new ArrayList<List<String>>();
        Map<String, List<String>> map = new HashMap<String, List<String>>();
        for (String str : strings) {
            int offset = str.charAt(0) - 'a';
            String key = "";
            for (int i = 0; i < str.length(); i++) {
                char c = (char) (str.charAt(i) - offset);
                if (c < 'a') {
                    c += 26;
                }
                key += c;
            }
            if (!map.containsKey(key)) {
                List<String> list = new ArrayList<String>();
                map.put(key, list);
            }
            map.get(key).add(str);
        }
        for (String key : map.keySet()) {
            List<String> list = map.get(key);
            Collections.sort(list);
            result.add(list);
        }
        return result;
    }
}
```

written by stevenye original link here

Answer 2

The key to this problem is how to identify strings that are in the same shifting sequence. There are different ways to encode this.

In the following code, this manner is adopted: for a string `s` of length `n`, we encode its shifting feature as `"s[1] - s[0], s[2] - s[1], ..., s[n - 1] - s[n - 2],"`.

Then we build an `unordered_map`, using the above shifting feature string as key and strings that share the shifting feature as value. We store all the strings that share the same shifting feature in a `vector`. Well, remember to `sort` the `vector` since the problem requires them to be in lexicographic order :-)

A final note, since the problem statement has given that `"az"` and `"ba"` belong to the same shifting sequence. So if `s[i] - s[i - 1]` is negative, we need to add `26` to it to make it positive and give the correct result. BTW, taking the suggestion of @StefanPochmann, we change the difference from numbers to lower-case alphabets

using `'a' + diff`.

The code is as follows.

```
class Solution {
public:
    vector<vector<string>> groupStrings(vector<string>& strings) {
        unordered_map<string, vector<string> > mp;
        for (string  s : strings)
            mp[shift(s)].push_back(s);
        vector<vector<string> > groups;
        for (auto m : mp) {
            vector<string> group = m.second;
            sort(group.begin(), group.end());
            groups.push_back(group);
        }
        return groups;
    }
private:
    string shift(string& s) {
        string t;
        int n = s.length();
        for (int i = 1; i < n; i++) {
            int diff = s[i] - s[i - 1];
            if (diff < 0) diff += 26;
            t += 'a' + diff + ',';
        }
        return t;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

To identify each group, compute the modulo 26 difference between each letter in a word with the first letter in it.

Solution 1: Ruby with `group_by`

```
def group_strings(strings)
  strings.sort.group_by { |s| s.bytes.map { |b| (b - s[0].ord) % 26 } }.values
end
```

Can be a bit faster to group first and sort (each group) afterwards:

```
def group_strings(strings)
  strings.group_by { |s| s.bytes.map { |b| (b - s[0].ord) % 26 } }.values.map &:s
ort
end
```

Solution 2: Python with `groupby`

```
def groupStrings(self, strings):
    key = lambda s: [(ord(c) - ord(s[0])) % 26 for c in s]
    return [sorted(g) for _, g in itertools.groupby(sorted(strings, key=key), key
)]
```

Solution 3: Python with `defaultdict`

```
def groupStrings(self, strings):
    groups = collections.defaultdict(list)
    for s in strings:
        groups[tuple((ord(c) - ord(s[0])) % 26 for c in s)] += s,
    return map(sorted, groups.values())
```

written by StefanPochmann original link here

Count Univalue Subtrees(250)

Answer 1

Helper `all` tells whether all nodes in the given tree have the given value. And while doing that, it also counts the uni-value subtrees.

```
public class Solution {
    int count = 0;
    boolean all(TreeNode root, int val) {
        if (root == null)
            return true;
        if (!all(root.left, root.val) | !all(root.right, root.val))
            return false;
        count++;
        return root.val == val;
    }
    public int countUnivalSubtrees(TreeNode root) {
        all(root, 0);
        return count;
    }
}
```

written by StefanPochmann original link here

Answer 2

```
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        int[] count = new int[1];
        helper(root, count);
        return count[0];
    }

    private boolean helper(TreeNode node, int[] count) {
        if (node == null) {
            return true;
        }
        boolean left = helper(node.left, count);
        boolean right = helper(node.right, count);
        if (left && right) {
            if (node.left != null && node.val != node.left.val) {
                return false;
            }
            if (node.right != null && node.val != node.right.val) {
                return false;
            }
            count[0]++;
            return true;
        }
        return false;
    }
}
```

written by stevenye original link here

Answer 3

```
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int[] counter = new int[1];
        count(root, counter, root.val);
        return counter[0];
    }

    private boolean count(TreeNode root, int[] counter, int val) {
        if (root == null) {
            return true;
        }
        boolean l = count(root.left, counter, root.val);
        boolean r = count(root.right, counter, root.val);

        if (l && r) {
            counter[0]++;
        }

        return l && r && root.val == val;
    }

}
```

written by AndyLiu0429 original link here

Flatten 2D Vector(251)

Answer 1

Since the OJ does `while (i.hasNext()) cout << i.next();`, i.e., always calls `hasNext` before `next`, I don't really have to call it myself so I could save that line in `next`. But I think that would be bad, we shouldn't rely on that.

C++

```
class Vector2D {
    vector<vector<int>>::iterator i, iEnd;
    int j = 0;
public:
    Vector2D(vector<vector<int>>& vec2d) {
        i = vec2d.begin();
        iEnd = vec2d.end();
    }

    int next() {
        hasNext();
        return (*i)[j++];
    }

    bool hasNext() {
        while (i != iEnd && j == (*i).size())
            i++, j = 0;
        return i != iEnd;
    }
};
```

Java

```
public class Vector2D {

    private Iterator<List<Integer>> i;
    private Iterator<Integer> j;

    public Vector2D(List<List<Integer>> vec2d) {
        i = vec2d.iterator();
    }

    public int next() {
        hasNext();
        return j.next();
    }

    public boolean hasNext() {
        while ((j == null || !j.hasNext()) && i.hasNext())
            j = i.next().iterator();
        return j != null && j.hasNext();
    }
}
```

written by StefanPochmann original link here

Answer 2

- Put all iterator in a queue
- Keep track of the current iterator
- Check hasNext() and next() of current

public class Vector2D {

```
Queue<Iterator<Integer>> queue;
Iterator<Integer> current = null;

public Vector2D(List<List<Integer>> vec2d) {
    queue = new LinkedList<Iterator<Integer>>();
    for (int i = 0; i < vec2d.size(); i++){
        queue.add(vec2d.get(i).iterator());
    }
    current = queue.poll(); // first
}

public int next() {
    if (!current.hasNext()) return -1;

    return current.next();
}

public boolean hasNext() {
    if (current == null) return false;

    while (!current.hasNext()) {
        if (!queue.isEmpty()) {
            current = queue.poll();
        } else return false;
    }

    return true;
}
```

}

written by angelvivienne original link here

Answer 3

//1. with positions of vectors

```
class Vector2D {
   int row;
   int col;
   vector<vector<int>> data;

public:
   Vector2D(vector<vector<int>>& vec2d) {
       data = vec2d;
       row = 0;
       col = 0;
   }

   int next() {
       return data[row][col++];
   }

   bool hasNext() {
       while(row < data.size() && data[row].size() == col)
           row++, col = 0;
       return row < data.size();
   }
};
```

//2. with Iterator

```
class Vector2D {
   vector<vector<int>> data;
   vector<vector<int>>::iterator rowIter;
   vector<int>::iterator colIter;

public:
   Vector2D(vector<vector<int>>& vec2d) {
       data = vec2d;
       rowIter = data.begin();
       if(rowIter != data.end())
           colIter = rowIter->begin();
   }

   int next() {
       int r = *colIter;
       colIter++;
       return r;
   }

   bool hasNext() {
       while(rowIter != data.end() && colIter == rowIter->end()) {
           rowIter++;
           if(rowIter != data.end())
                colIter = rowIter->begin();
       }

       return rowIter != data.end();
   }
 };
```

written by jaewoo original link here

Meeting Rooms(252)

Answer 1

```
public boolean canAttendMeetings(Interval[] intervals) {
  if (intervals == null)
    return false;

  // Sort the intervals by start time
  Arrays.sort(intervals, new Comparator<Interval>() {
    public int compare(Interval a, Interval b) { return a.start - b.start; }
  });

  for (int i = 1; i < intervals.length; i++)
    if (intervals[i].start < intervals[i - 1].end)
      return false;

  return true;
}
```

written by jeantimex original link here

Answer 2

The idea is pretty simple: first we sort the `intervals` in the ascending order of `start`; then we check for the overlapping of each pair of neighboring intervals. If they do, then return `false`; after we finish all the checks and have not returned `false`, just return `true`.

Sorting takes `O(nlogn)` time and the overlapping checks take `O(n)` time, so this idea is `O(nlogn)` time in total.

The code is as follows.

```
class Solution {
public:
    bool canAttendMeetings(vector<Interval>& intervals) {
        sort(intervals.begin(), intervals.end(), compare);
        int n = intervals.size();
        for (int i = 0; i < n - 1; i++)
            if (overlap(intervals[i], intervals[i + 1]))
                return false;
        return true;
    }
private:
    static bool compare(Interval& interval1, Interval& interval2) {
        return interval1.start < interval2.start;
    }
    bool overlap(Interval& interval1, Interval& interval2) {
        return interval1.end > interval2.start;
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```
class Solution {
public:
    bool canAttendMeetings(vector<Interval>& intervals) {
        sort(intervals.begin(), intervals.end(),
            [](const Interval& interval1, const Interval& interval2){
                return interval1.start < interval2.start;
            });

        for (int i = 1; i < intervals.size(); i++){
            if (intervals[i].start < intervals[i-1].end)
                return false;
        }
        return true;
    }
};
```

written by whnzinc original link here

Meeting Rooms II(253)

Answer 1

Just want to share another idea that uses min heap, average time complexity is O(nlogn).

```
public int minMeetingRooms(Interval[] intervals) {
    if (intervals == null || intervals.length == 0)
        return 0;

    // Sort the intervals by start time
    Arrays.sort(intervals, new Comparator<Interval>() {
        public int compare(Interval a, Interval b) { return a.start - b.start; }
    });

    // Use a min heap to track the minimum end time of merged intervals
    PriorityQueue<Interval> heap = new PriorityQueue<Interval>(intervals.length,
new Comparator<Interval>() {
        public int compare(Interval a, Interval b) { return a.end - b.end; }
    });

    // start with the first meeting, put it to a meeting room
    heap.offer(intervals[0]);

    for (int i = 1; i < intervals.length; i++) {
        // get the meeting room that finishes earliest
        Interval interval = heap.poll();

        if (intervals[i].start >= interval.end) {
            // if the current meeting starts right after
            // there's no need for a new room, merge the interval
            interval.end = intervals[i].end;
        } else {
            // otherwise, this meeting needs a new room
            heap.offer(intervals[i]);
        }

        // don't forget to put the meeting room back
        heap.offer(interval);
    }

    return heap.size();
}
```

written by jeantimex original link here

Answer 2

First collect the changes: at what times the number of meetings goes up or down and by how much. Then go through those changes in ascending order and keep track of the current and maximum number of rooms needed.

Solution 1: Using `map` ... 600 ms

```
int minMeetingRooms(vector<Interval>& intervals) {
    map<int, int> changes;
    for (auto i : intervals) {
        changes[i.start] += 1;
        changes[i.end] -= 1;
    }
    int rooms = 0, maxrooms = 0;
    for (auto change : changes)
        maxrooms = max(maxrooms, rooms += change.second);
    return maxrooms;
}
```

Solution 2: Using `vector` ... 588 ms

```
int minMeetingRooms(vector<Interval>& intervals) {
    vector<pair<int, int>> changes;
    for (auto i : intervals) {
        changes.push_back({i.start, 1});
        changes.push_back({i.end, -1});
    };
    sort(begin(changes), end(changes));
    int rooms = 0, maxrooms = 0;
    for (auto change : changes)
        maxrooms = max(maxrooms, rooms += change.second);
    return maxrooms;
}
```

Solution 3: Using two `vector`s ... 584 ms

Based on yinfeng.zhang.9's Python solution. Uses separate vectors for start and end times, which ends up consistently being fastest. I'm guessing it's mostly because working with ints is simpler than working with pairs of ints. The initial sorting also needs fewer steps, 2(nlogn) instead of (2n)log(2n), but I think the added merging in the later loop cancels that advantage out.

```
int minMeetingRooms(vector<Interval>& intervals) {
    vector<int> starts, ends;
    for (auto i : intervals) {
        starts.push_back(i.start);
        ends.push_back(i.end);
    }
    sort(begin(starts), end(starts));
    sort(begin(ends), end(ends));
    int e = 0, rooms = 0, available = 0;
    for (int start : starts) {
        while (ends[e] <= start) {
            ++e;
            ++available;
        }
        available ? --available : ++rooms;
    }
    return rooms;
}
```

written by StefanPochmann original link here

Answer 3

Simulate event queue procession. Create event for each `start` and `end` of intervals. Then for `start` event, open one more room; for `end` event, close one meeting room. At the same time, update the most rooms that is required.

Be careful of events like `[(end at 11), (start at 11)]`. Put `end` before `start` event when they share the same happening time, so that two events can share one meeting room.

```java
public class Solution {

    private static final int START = 1;

    private static final int END = 0;

    private class Event {
        int time;
        int type; // end event is 0; start event is 1

        public Event(int time, int type) {
            this.time = time;
            this.type = type;
        }
    }

    public int minMeetingRooms(Interval[] intervals) {
        int rooms = 0; // occupied meeting rooms
        int res = 0;

        // initialize an event queue based on event's happening time
        Queue<Event> events = new PriorityQueue<>(new Comparator<Event>() {
            @Override
            public int compare(Event e1, Event e2) {
                // for same time, let END event happens first to save rooms
                return e1.time != e2.time ?
                       e1.time - e2.time : e1.type - e2.type;
            }
        });

        // create event and push into event queue
        for (Interval interval : intervals) {
            events.offer(new Event(interval.start, START));
            events.offer(new Event(interval.end, END));
        }

        // process events
        while (!events.isEmpty()) {
            Event event = events.poll();
            if (event.type == START) {
                rooms++;
                res = Math.max(res, rooms);
            } else {
                rooms--;
            }
        }

        return res;
    }

}
```

written by StevenCooks original link here

Factor Combinations(254)

Answer 1

```
public List<List<Integer>> getFactors(int n) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    helper(result, new ArrayList<Integer>(), n, 2);
    return result;
}

public void helper(List<List<Integer>> result, List<Integer> item, int n, int star
t){
    if (n <= 1) {
        if (item.size() > 1) {
            result.add(new ArrayList<Integer>(item));
        }
        return;
    }

    for (int i = start; i <= n; ++i) {
        if (n % i == 0) {
            item.add(i);
            helper(result, item, n/i, i);
            item.remove(item.size()-1);
        }
    }
}
```

written by yinfeng.zhang.9 original link here

Answer 2

```
class Solution {
    public:
        void getResult(vector<vector<int>> &result,vector<int> &row,int n){
            int i=row.empty()?2:row.back();
            for(;i<=n/i;++i){
                if(n%i==0){
                    row.push_back(i);
                    row.push_back(n/i);
                    result.push_back(row);
                    row.pop_back();
                    getResult(result,row,n/i);
                    row.pop_back();
                }
            }
        }

        vector<vector<int>> getFactors(int n) {
            vector<vector<int>> result;
            vector<int>row;
            getResult(result,row,n);
            return result;
        }
    };
```

written by jiannan original link here

Answer 3

```
public List<List<Integer>> getFactors(int n) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    if (n <= 3) return result;
    helper(n, -1, result, new ArrayList<Integer>());
    return result;
}

public void helper(int n, int lower, List<List<Integer>> result, List<Integer> cur
) {
    if (lower != -1) {
        cur.add(n);
        result.add(new ArrayList<Integer>(cur));
        cur.remove(cur.size() - 1);
    }
    int upper = (int) Math.sqrt(n);
    for (int i = Math.max(2, lower); i <= upper; ++i) {
        if (n % i == 0) {
            cur.add(i);
            helper(n / i, i, result, cur);
            cur.remove(cur.size() - 1);
        }
    }
}
```

written by hahadaxiong original link here

Verify Preorder Sequence in Binary Search Tree(255)

Answer 1

Solution 1

Kinda simulate the traversal, keeping a stack of nodes (just their values) of which we're still in the left subtree. If the next number is smaller than the last stack value, then we're still in the left subtree of all stack nodes, so just push the new one onto the stack. But before that, pop all smaller ancestor values, as we must now be in their right subtrees (or even further, in the right subtree of an ancestor). Also, use the popped values as a lower bound, since being in their right subtree means we must never come across a smaller number anymore.

```
public boolean verifyPreorder(int[] preorder) {
    int low = Integer.MIN_VALUE;
    Stack<Integer> path = new Stack();
    for (int p : preorder) {
        if (p < low)
            return false;
        while (!path.empty() && p > path.peek())
            low = path.pop();
        path.push(p);
    }
    return true;
}
```

Solution 2 ... O(1) extra space

Same as above, but abusing the given array for the stack.

```
public boolean verifyPreorder(int[] preorder) {
    int low = Integer.MIN_VALUE, i = -1;
    for (int p : preorder) {
        if (p < low)
            return false;
        while (i >= 0 && p > preorder[i])
            low = preorder[i--];
        preorder[++i] = p;
    }
    return true;
}
```

Solution 3 ... Python

Same as solution 1, just in Python.

```
def verifyPreorder(self, preorder):
    stack = []
    low = float('-inf')
    for p in preorder:
        if p < low:
            return False
        while stack and p > stack[-1]:
            low = stack.pop()
        stack.append(p)
    return True
```

written by StefanPochmann original link here

Answer 2

The idea is traversing the preorder list and using a stack to store all predecessors. curr*p is a predecessor of current node and current node is in the right subtree of curr*p.

For example, for the following bst with preorder 6,3,1,2,5,4,7:

```
          6
        /  \
       3    7
      /  \
     1   5
     \   /
     2  4
```

We push to stack before we see 2. So at 2 the stack is 6,3,1. For 2, we pop stack until we see 3 which is greater than 2 and curr*p is 1. 2 is in left subtree of 3 and is right child of 1. Stack is 6,3,2 now. Then we see 5, and we pop stack until 6 and curr*p is 3. Stack now is 6,5. Then we see 4 and push to stack. At 7, we pop stack until empty and curr_p is 6.

```
bool verifyPreorder(vector<int>& preorder){
    // using stack
    int sz = preorder.size();
    if(sz < 2) return true;
    stack<int> s;
    s.push(preorder[0]);
    int curr_p = INT_MIN;
    for(int i=1; i<sz; i++){
        if(s.empty() || preorder[i]<s.top()){ // if current node is less than sta
ck top, then go to left subtree
            if(preorder[i]<curr_p) return false;
            s.push(preorder[i]);
        }
        else{
            while(!s.empty() && s.top()<preorder[i]){ //find curr_p such that cur
rent node is right child of curr_p
                curr_p = s.top();
                s.pop();
            }
            s.push(preorder[i]);
        }
    }
    return true;
}
```

written by yu23 original link here

Answer 3

I'm in bit of disbelief that my solution was accepted. I coded it up while I was really sleepy, and hit submit, fully expecting it to fail compilation or something...

The logic is as follows:

For any pre-order sequence, we have:

1. The root, followed by
2. Another preorder sequence for the LHS, where everything is less than root, followed by,
3. Another preorder sequence for the RHS, where everything is greater than root.

In order to find the boundary of the LHS preorder and the RHS preorder, you can run binary search to look for the boundary where the elements transition from being (< root) to being (> root).

Once you find the boundary, you can recursively call the verify function on the LHS preorder sequence and the RHS preorder sequence. The trick is to pass two elements along, that say what the relationship of the elements should be w.r.t. the ancestors that have gone by so far. e.g. I am passing two elements (lessThan and greaterThan) which say that all elements in the preorder sequence need to be less than lessThan, and greater than greaterThan. Look at the recursive calls for lhsverify and rhsverify bools to see what is being passed on to recursive calls.

```
class Solution {
    //boundary denotes LAST number that is less than or equal to num
    int binarySearchBoundary(vector<int>& preorder, int from, int to, int num) {
        if (from == to) return from;
        int mid = (from+to)/2;
        int midplus = mid + 1;
        if (preorder[mid] <= num && preorder[midplus] > num) return mid;
        if (preorder[mid] > num) return binarySearchBoundary(preorder, from, mid,
num);
        return binarySearchBoundary(preorder, midplus, to, num);
    }

    //verify the sequence from "from" to "to", in which every elements needs to be

    // less than "lessThan", and greater than "greaterThan"
    bool verify(vector<int>& preorder, int from, int to, int lessThan, int greate
rThan) {
        if (from > to) return true;
        int root = preorder[from];
        if (root > lessThan || root < greaterThan) return false;
        if (from == to) return true;

        int boundary = binarySearchBoundary(preorder, from, to, root);
        bool lhsverify = verify(preorder, from+1, boundary, min(lessThan, root),
greaterThan);
        bool rhsverify = verify(preorder, boundary+1, to, lessThan, max(root, gre
aterThan));
        return (lhsverify && rhsverify);
    }

public:
    bool verifyPreorder(vector<int>& preorder) {
        if (preorder.size() == 0) return true;
        return verify(preorder, 0, preorder.size()-1, std::numeric_limits<int>::m
ax(), std::numeric_limits<int>::min());
    }
};
```

written by cbrghostrider original link here

Paint House(256)

Answer 1

The 1st row is the prices for the 1st house, we can change the matrix to present sum of prices from the 2nd row. i.e, the costs[1][0] represent minimum price to paint the second house red plus the 1st house.

```
public class Solution {
public int minCost(int[][] costs) {
    if(costs==null||costs.length==0){
        return 0;
    }
    for(int i=1; i<costs.length; i++){
        costs[i][0] += Math.min(costs[i-1][1],costs[i-1][2]);
        costs[i][1] += Math.min(costs[i-1][0],costs[i-1][2]);
        costs[i][2] += Math.min(costs[i-1][1],costs[i-1][0]);
    }
    int n = costs.length-1;
    return Math.min(Math.min(costs[n][0], costs[n][1]), costs[n][2]);
}
```

}

written by yao9208 original link here

Answer 2

```
public class Solution {
    public int minCost(int[][] costs) {

        if(costs==null || costs.length==0) return 0;
        int[] prevRow = costs[0];
         for(int i=1;i<costs.length;i++)
        {
            int[] currRow = new int[3];
            for(int j=0;j<3;j++)
                currRow[j]=costs[i][j]+Math.min(prevRow[(j+1)%3],prevRow[(j+2)%3]
);
            prevRow = currRow;
        }
        return Math.min(prevRow[0],Math.min(prevRow[1],prevRow[2]));
    }
}
```

written by prachibhans original link here

Answer 3

the idea is to store current house's minimum cost in different colors.

```
def minCost(self, costs):
    size = len(costs)
    if size == 0:
        return 0

    pre = costs[0][:]
    now = [0]*3

    for i in xrange(size-1):
        now[0] = min(pre[1], pre[2]) + costs[i+1][0]
        now[1] = min(pre[0], pre[2]) + costs[i+1][1]
        now[2] = min(pre[0], pre[1]) + costs[i+1][2]
        pre[:] = now[:]

    return min(pre)
```

written by autekwing original link here

Binary Tree Paths(257)

Answer 1

```
public List<String> binaryTreePaths(TreeNode root) {
    List<String> answer = new ArrayList<String>();
    if (root != null) searchBT(root, "", answer);
    return answer;
}
private void searchBT(TreeNode root, String path, List<String> answer) {
    if (root.left == null && root.right == null) answer.add(path + root.val);
    if (root.left != null) searchBT(root.left, path + root.val + "->", answer);
    if (root.right != null) searchBT(root.right, path + root.val + "->", answer);
}
```

written by vimukthi original link here

Answer 2

```
void binaryTreePaths(vector<string>& result, TreeNode* root, string t) {
    if(!root->left && !root->right) {
        result.push_back(t);
        return;
    }

    if(root->left) binaryTreePaths(result, root->left, t + "->" + to_string(root-
>left->val));
    if(root->right) binaryTreePaths(result, root->right, t + "->" + to_string(roo
t->right->val));
}

vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> result;
    if(!root) return result;

    binaryTreePaths(result, root, to_string(root->val));
    return result;
}
```

written by jaewoo original link here

Answer 3

Lot of recursive solutions on this forum involves creating a helper recursive function with added parameters. The added parameter which usually is of the type List , carries the supplementary path information. However, the approach below doesn't use such a helper function.

```
public List<String> binaryTreePaths(TreeNode root) {

        List<String> paths = new LinkedList<>();

        if(root == null) return paths;

        if(root.left == null && root.right == null){
            paths.add(root.val+"");
            return paths;
        }

         for (String path : binaryTreePaths(root.left)) {
             paths.add(root.val + "->" + path);
         }

         for (String path : binaryTreePaths(root.right)) {
             paths.add(root.val + "->" + path);
         }

         return paths;

    }
```

written by sanket.purbey original link here

Add Digits(258)

Answer 1

The problem, widely known as *digit root* problem, has a congruence formula:

```
https://en.wikipedia.org/wiki/Digital_root#Congruence_formula
```

For base b (decimal case b = 10), the digit root of an integer is:

- dr(n) = 0 if n == 0
- dr(n) = (b-1) if n != 0 and n % (b-1) == 0
- dr(n) = n mod (b-1) if n % (b-1) != 0

or

- dr(n) = 1 + (n - 1) % 9

Note here, when n = 0, since (n - 1) % 9 = -1, the return value is zero (correct).

From the formula, we can find that the result of this problem is immanently periodic, with period (b-1).

Output sequence for decimals (b = 10):

~input: 0 1 2 3 4 ...
output: 0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 ....

Henceforth, we can write the following code, whose time and space complexities are both O(1).

```
class Solution {
public:
    int addDigits(int num) {
        return 1 + (num - 1) % 9;
    }
};
```

Thanks for reading. :)

written by zhiqing_xiao original link here

Answer 2

```
int addDigits(int num) {
    int res = num % 9;
    return (res != 0 || num == 0) ? res : 9;
}
```

The essence of this problem is that 10^n â‰¡ 1 (mod 9), and thus a*n*10^*n* + ... + *a*1*10 + a_0 â‰¡ a_n + ... + a_1 + a_0 (mod 9). This process can be continued until a number less than 9 is gotten, i.e. num % 9. For any digit n, n = n % 9 unless n = 9. The only confusing case is n % 9 = 0, but addDigits(num) = 0 if and only if num = 0,

otherwise it should be 9 in fact.

written by DeKoder original link here

Answer 3

If an integer is like 100a+10b+c, then (100a+10b+c)%9=(a+99a+b+9b+c)%9=(a+b+c)%9

```
class Solution:
    # @param {integer} num
    # @return {integer}
    def addDigits(self, num):
        if num==0:
            return 0
        return num%9 if num%9!=0 else 9
```

written by yawn.zheng original link here

3Sum Smaller(259)

Answer 1

After sorting, if `i, j, k` is a valid triple, then `i, j-1, k`, ..., `i, i+1, k` are also valid triples. No need to count them one by one.

```
def threeSumSmaller(self, nums, target):
    nums.sort()
    count = 0
    for k in range(len(nums)):
        i, j = 0, k - 1
        while i < j:
            if nums[i] + nums[j] + nums[k] < target:
                count += j - i
                i += 1
            else:
                j -= 1
    return count
```

written by StefanPochmann original link here

Answer 2

```
public class Solution {
    int count;

    public int threeSumSmaller(int[] nums, int target) {
        count = 0;
        Arrays.sort(nums);
        int len = nums.length;

        for(int i=0; i<len-2; i++) {
            int left = i+1, right = len-1;
            while(left < right) {
                if(nums[i] + nums[left] + nums[right] < target) {
                    count += right-left;
                    left++;
                } else {
                    right--;
                }
            }
        }

        return count;
    }
}
```

written by SOY original link here

Answer 3

```
class Solution {
public:
    int threeSumSmaller(vector<int>& nums, int target) {
        if(nums.size()<3)return 0;
        sort(nums.begin(),nums.end());
        int count=0;
        for(int i=0;i<nums.size()-2;++i){
            if(nums[i]+nums[i+1]+nums[i+2]>=target)break;
            int j=i+1,k=nums.size()-1;
            while(j<k){
                while(j<k && nums[i]+nums[j]+nums[k]>=target)k--;
                count+=k-j;
                j++;
            }
        }
        return count;
    }
};
```

written by jiannan original link here

Single Number III(260)

Answer 1

Once again, we need to use XOR to solve this problem. But this time, we need to do it in two passes:

- In the first pass, we XOR all elements in the array, and get the XOR of the two numbers we need to find. Note that since the two numbers are distinct, so there must be a set bit (that is, the bit with value '1') in the XOR result. Find out an arbitrary set bit (for example, the rightmost set bit).
- In the second pass, we divide all numbers into two groups, one with the aforementioned bit set, another with the aforementinoed bit unset. Two different numbers we need to find must fall into thte two distrinct groups. XOR numbers in each group, we can find a number in either group.

Complexity:

- Time: $O(n)$
- Space: $O(1)$

A Corner Case:

- When `diff == numeric_limits<int>::min()`, `-diff` is also `numeric_limits<int>::min()`. Therefore, the value of `diff` after executing `diff &= -diff` is still `numeric_limits<int>::min()`. The answer is still correct.

C++:

```cpp
class Solution
{
public:
    vector<int> singleNumber(vector<int>& nums)
    {
        // Pass 1 :
        // Get the XOR of the two numbers we need to find
        int diff = accumulate(nums.begin(), nums.end(), 0, bit_xor<int>());
        // Get its last set bit
        diff &= -diff;

        // Pass 2 :
        vector<int> rets = {0, 0}; // this vector stores the two numbers we will
return
        for (int num : nums)
        {
            if ((num & diff) == 0) // the bit is not set
            {
                rets[0] ^= num;
            }
            else // the bit is set
            {
                rets[1] ^= num;
            }
        }
        return rets;
    }
};
```

Java:

```java
public class Solution {
    public int[] singleNumber(int[] nums) {
        // Pass 1 :
        // Get the XOR of the two numbers we need to find
        int diff = 0;
        for (int num : nums) {
            diff ^= num;
        }
        // Get its last set bit
        diff &= -diff;

        // Pass 2 :
        int[] rets = {0, 0}; // this array stores the two numbers we will return
        for (int num : nums)
        {
            if ((num & diff) == 0) // the bit is not set
            {
                rets[0] ^= num;
            }
            else // the bit is set
            {
                rets[1] ^= num;
            }
        }
        return rets;
    }
}
```

Thanks for reading :)

Acknowledgements:

- Thank **@jianchao.li.fighter** for introducing this problem and for your encouragement.
- Thank **@StefanPochmann** for your valuable suggestions and comments. Your idea of `diff &= -diff` is very elegent! And yes, it does not need to XOR for both group in the second pass. XOR for one group suffices. I revise my code accordingly.
- Thank **@Nakagawa_Kanon** for posting this question and presenting the same idea in a previous thread (prior to this thread).
- Thank **@caijun** for providing an interesting test case.

written by zhiqing_xiao original link here

Answer 2

If you were stuck by this problem, it's easy to find a solution in the discussion. However, usually, the solution lacks some explanations.

I'm sharing my understanding here:

The two numbers that appear only once must differ at some bit, this is how we can distinguish between them. Otherwise, they will be one of the duplicate numbers.

Letâ€™s say the at the ith bit, the two desired numbers differ from each other. which means one number has bit i equaling: 0, the other number has bit i equaling 1.

Thus, all the numbers can be partitioned into two groups according to their bits at location i. the first group consists of all numbers whose bits at i is 0. the second group consists of all numbers whose bits at i is 1.

Notice that, if a duplicate number has bit i as 0, then, two copies of it will belong to the first group. Similarly, if a duplicate number has bit i as 1, then, two copies of it will belong to the second group.

by XoRing all numbers in the first group, we can get the first number. by XoRing all numbers in the second group, we can get the second number.

written by douglasleer original link here

Answer 3

```
vector<int> singleNumber(vector<int>& nums) {
    int aXorb = 0;  // the result of a xor b;
    for (auto item : nums) aXorb ^= item;
    int lastBit = (aXorb & (aXorb - 1)) ^ aXorb;  // the last bit that a diffs b
    int intA = 0, intB = 0;
    for (auto item : nums) {
        // based on the last bit, group the items into groupA(include a) and grou
pB
        if (item & lastBit) intA = intA ^ item;
        else intB = intB ^ item;
    }
    return vector<int>{intA, intB};
}
```

written by lchen77 original link here

Graph Valid Tree(261)

Answer 1

```java
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        // initialize n isolated islands
        int[] nums = new int[n];
        Arrays.fill(nums, -1);

        // perform union find
        for (int i = 0; i < edges.length; i++) {
            int x = find(nums, edges[i][0]);
            int y = find(nums, edges[i][1]);

            // if two vertices happen to be in the same set
            // then there's a cycle
            if (x == y) return false;

            // union
            nums[x] = y;
        }

        return edges.length == n - 1;
    }

    int find(int nums[], int i) {
        if (nums[i] == -1) return i;
        return find(nums, nums[i]);
    }
}
```

written by jeantimex original link here

Answer 2

There are so many different approaches and so many different ways to implement each. I find it hard to decide, so here are several :-)

In all of them, I check one of these tree characterizations:

- Has n-1 edges and is acyclic.
- Has n-1 edges and is connected.

Solution 1 ... Union-Find

The test cases are small and harmless, simple union-find suffices (runs in about 50~60 ms).

```
def validTree(self, n, edges):
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    def union(xy):
        x, y = map(find, xy)
        parent[x] = y
        return x != y
    return len(edges) == n-1 and all(map(union, edges))
```

A version without using `all(...)`, to be closer to other programming languages:

```
def validTree(self, n, edges):
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    for e in edges:
        x, y = map(find, e)
        if x == y:
            return False
        parent[x] = y
    return len(edges) == n - 1
```

A version checking `len(edges) != n - 1` first, as `parent = range(n)` could fail for huge `n`:

```
def validTree(self, n, edges):
    if len(edges) != n - 1:
        return False
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    def union(xy):
        x, y = map(find, xy)
        parent[x] = y
        return x != y
    return all(map(union, edges))
```

Solution 2 ... DFS

```
def validTree(self, n, edges):
    neighbors = {i: [] for i in range(n)}
    for v, w in edges:
        neighbors[v] += w,
        neighbors[w] += v,
    def visit(v):
        map(visit, neighbors.pop(v, []))
    visit(0)
    return len(edges) == n-1 and not neighbors
```

Or check the number of edges first, to be faster and to survive unreasonably huge `n`:

```
def validTree(self, n, edges):
    if len(edges) != n - 1:
        return False
    neighbors = {i: [] for i in range(n)}
    for v, w in edges:
        neighbors[v] += w,
        neighbors[w] += v,
    def visit(v):
        map(visit, neighbors.pop(v, []))
    visit(0)
    return not neighbors
```

For an iterative version, just replace the three "visit" lines with

```
    stack = [0]
    while stack:
        stack += neighbors.pop(stack.pop(), [])
```

Solution 3 ... BFS

Just like DFS above, but replace the three "visit" lines with

```
    queue = [0]
    for v in queue:
        queue += neighbors.pop(v, [])
```

or, since that is not guaranteed to work, the safer

```
    queue = collections.deque([0])
    while queue:
        queue.extend(neighbors.pop(queue.popleft(), []))
```

written by StefanPochmann original link here

Answer 3

```java
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        // initialize adjacency list
        List<List<Integer>> adjList = new ArrayList<List<Integer>>(n);

        // initialize vertices
        for (int i = 0; i < n; i++)
            adjList.add(i, new ArrayList<Integer>());

        // add edges
        for (int i = 0; i < edges.length; i++) {
            int u = edges[i][0], v = edges[i][1];
            adjList.get(u).add(v);
            adjList.get(v).add(u);
        }

        boolean[] visited = new boolean[n];

        // make sure there's no cycle
        if (hasCycle(adjList, 0, visited, -1))
            return false;

        // make sure all vertices are connected
        for (int i = 0; i < n; i++) {
            if (!visited[i])
                return false;
        }

        return true;
    }

    // check if an undirected graph has cycle started from vertex u
    boolean hasCycle(List<List<Integer>> adjList, int u, boolean[] visited, int pa
rent) {
        visited[u] = true;

        for (int i = 0; i < adjList.get(u).size(); i++) {
            int v = adjList.get(u).get(i);

            if ((visited[v] && parent != v) || (!visited[v] && hasCycle(adjList,
v, visited, u)))
                return true;
        }

        return false;
    }
}
```

written by jeantimex original link here

Ugly Number(263)

Answer 1

Just divide by 2, 3 and 5 as often as possible and then check whether we arrived at 1. Also try divisor 4 if that makes the code simpler / less repetitive.

C++ / C

```
for (int i=2; i<6 && num; i++)
    while (num % i == 0)
        num /= i;
return num == 1;
```

Ruby

```
(2..5).each { |i| num /= i while num % i == 0 } if num > 0
num == 1
```

Or:

```
require 'prime'
num > 0 && num.prime_division.all? { |p, _| p < 6 }
```

Python

```
for p in 2, 3, 5:
    while num % p == 0 < num:
        num /= p
return num == 1
```

Java / C#

```
for (int i=2; i<6 && num>0; i++)
    while (num % i == 0)
        num /= i;
return num == 1;
```

Javascript

```
for (var p of [2, 3, 5])
    while (num && num % p == 0)
        num /= p;
return num == 1;
```

General

Would be a bit cleaner if I did the zero-test outside, and discarding negative

numbers right away can speed things up a little, but meh... I don't want to add another line and indentation level :-)

```
if (num > 0)
    for (int i=2; i<6; i++)
        while (num % i == 0)
            num /= i;
return num == 1;
```

written by StefanPochmann original link here

Answer 2

```
public boolean isUgly(int num) {
    if(num==1) return true;
    if(num==0) return false;
    while(num%2==0) num=num>>1;
    while(num%3==0) num=num/3;
    while(num%5==0) num=num/5;
    return num==1;
}
```

written by kittyfeng original link here

Answer 3

```
public boolean isUgly(int num) {
    if (num <= 0) {return false;}
    if (num == 1) {return true;}
    if (num % 2 == 0) {
        return isUgly(num/2);
    }
    if (num % 3 == 0) {
        return isUgly(num/3);
    }
    if (num % 5 == 0) {
        return isUgly(num/5);
    }
    return false;
}
```

---

idea:

- (1) basic cases: <= 0 and == 1
- (2) other cases: since the number can contain the factors of 2, 3, 5, I just remove those factors. So now, I have a number without any factors of 2, 3, 5.
- (3) after the removing, the number (new number) can contain a) the factor that is prime and meanwhile it is >= 7, or b) the factor that is not the prime and the factor is not comprised of 2, 3 or 5. In both cases, it is false (not ugly number).

For example, new number can be 11, 23 --> not ugly number (case a)). new number also can be 49, 121 --> not ugly number (case b))

written by hello_today_ original link here

Ugly Number II(264)

Answer 1

We have an array *k* of first n ugly number. We only know, at the beginning, the first one, which is 1. Then

k[1] = min(k[0]x2, k[0]x3, k[0]x5). The answer is k[0]x2. So we move 2's pointer to 1. Then we test:

k[2] = min(k[1]x2, k[0]x3, k[0]x5). And so on. Be careful about the cases such as 6, in which we need to forward both pointers of 2 and 3.

x here is multiplication.

```
class Solution {
public:
    int nthUglyNumber(int n) {
        if(n <= 0) return false; // get rid of corner cases
        if(n == 1) return true; // base case
        int t2 = 0, t3 = 0, t5 = 0; //pointers for 2, 3, 5
        vector<int> k(n);
        k[0] = 1;
        for(int i  = 1; i < n ; i ++)
        {
            k[i] = min(k[t2]*2,min(k[t3]*3,k[t5]*5));
            if(k[i] == k[t2]*2) t2++;
            if(k[i] == k[t3]*3) t3++;
            if(k[i] == k[t5]*5) t5++;
        }
        return k[n-1];
    }
};
```

written by jmty0083 original link here

Answer 2

The idea of this solution is from this page:http://www.geeksforgeeks.org/ugly-numbers/

The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, â€¦ because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

```
(1) 1Ã—2, 2Ã—2, 3Ã—2, 4Ã—2, 5Ã×2, â€¦
(2) 1Ã—3, 2Ã×3, 3Ã×3, 4Ã×3, 5Ã×3, â€¦
(3) 1Ã×5, 2Ã×5, 3Ã×5, 4Ã×5, 5Ã×5, â€¦
```

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, â€¦) multiply 2, 3, 5.

Then we use similar merge method as merge sort, to get every ugly number from the three subsequence.

Every step we choose the smallest one, and move one step after,including nums with same value.

Thanks for this author about this brilliant idea. Here is my java solution

```
public class Solution {
    public int nthUglyNumber(int n) {
        int[] ugly = new int[n];
        ugly[0] = 1;
        int index2 = 0, index3 = 0, index5 = 0;
        int factor2 = 2, factor3 = 3, factor5 = 5;
        for(int i=1;i<n;i++){
            int min = Math.min(Math.min(factor2,factor3),factor5);
            ugly[i] = min;
            if(factor2 == min)
                factor2 = 2*ugly[++index2];
            if(factor3 == min)
                factor3 = 3*ugly[++index3];
            if(factor5 == min)
                factor5 = 5*ugly[++index5];
        }
        return ugly[n-1];
    }
}
```

written by syftalent original link here

Answer 3

```
struct Solution {
    int nthUglyNumber(int n) {
        vector <int> results (1,1);
        int i = 0, j = 0, k = 0;
        while (results.size() < n)
        {
            results.push_back(min(results[i] * 2, min(results[j] * 3, results[k]
* 5)));
            if (results.back() == results[i] * 2) ++i;
            if (results.back() == results[j] * 3) ++j;
            if (results.back() == results[k] * 5) ++k;
        }
        return results.back();
    }
};
```

Explanation:

The key is to realize each number can be and have to be generated by a former number multiplied by 2, 3 or 5 e.g. 1 2 3 4 5 6 8 9 10 12 15.. what is next? it must be x * 2 or y * 3 or z * 5, where x, y, z is an existing number.

How do we determine x, y, z then? apparently, you can just *traverse the sequence generated by far* from 1 ... 15, until you find such x, y, z that x * 2, y * 3, z * 5 is just bigger than 15. In this case x=8, y=6, z=4. Then you compare x * 2, y * 3, z * 5 so you

know next number will be x * 2 = 8 * 2 = 16. k, now you have 1,2,3,4,....,15, 16,

Then what is next? You wanna do the same process again to find the new x, y, z, but you realize, wait, do I have to *traverse the sequence generated by far* again?

NO! since you know last time, x=8, y=6, z=4 and x=8 was used to generate 16, so this time, you can immediately know the new*x = 9 (the next number after 8 is 9 in the generated sequence), y=6, z=4. Then you need to compare new*x * 2, y * 3, z * 5. You know next number is 9 * 2 = 18;

And you also know, the next x will be 10 since new_x = 9 was used this time. But what is next y? apparently, if y=6, 6*3 = 18, which is already generated in this round. So you also need to update next y from 6 to 8.

Based on the idea above, you can actually generated x,y,z from very beginning, and update x, y, z accordingly. It ends up with a O(n) solution.

written by fentoyal original link here

Paint House II(265)

Answer 1

The idea is similar to the problem Paint House I, for each house and each color, the minimum cost of painting the house with that color should be the minimum cost of painting previous houses, and make sure the previous house doesn't paint with the same color.

We can use min1 and min2 to track the indices of the 1st and 2nd smallest cost till previous house, if the current color's index is same as min1, then we have to go with min2, otherwise we can safely go with min1.

The code below modifies the value of costs[][] so we don't need extra space.

```
public int minCostII(int[][] costs) {
    if (costs == null || costs.length == 0) return 0;

    int n = costs.length, k = costs[0].length;
    // min1 is the index of the 1st-smallest cost till previous house
    // min2 is the index of the 2nd-smallest cost till previous house
    int min1 = -1, min2 = -1;

    for (int i = 0; i < n; i++) {
        int last1 = min1, last2 = min2;
        min1 = -1; min2 = -1;

        for (int j = 0; j < k; j++) {
            if (j != last1) {
                // current color j is different to last min1
                costs[i][j] += last1 < 0 ? 0 : costs[i - 1][last1];
            } else {
                costs[i][j] += last2 < 0 ? 0 : costs[i - 1][last2];
            }

            // find the indices of 1st and 2nd smallest cost of painting current
house i
            if (min1 < 0 || costs[i][j] < costs[i][min1]) {
                min2 = min1; min1 = j;
            } else if (min2 < 0 || costs[i][j] < costs[i][min2]) {
                min2 = j;
            }
        }
    }

    return costs[n - 1][min1];
}
```

written by jeantimex original link here

Answer 2

maintain the minimum two costs min1(smallest) and min2 (second to smallest) after painting i-th house.

```
int minCostII(vector<vector<int>>& costs) {
    int n = costs.size();
    if(n==0) return 0;
    int k = costs[0].size();
    if(k==1) return costs[0][0];

    vector<int> dp(k, 0);
    int min1, min2;

    for(int i=0; i<n; ++i){
        int min1_old = (i==0)?0:min1;
        int min2_old = (i==0)?0:min2;
        min1 = INT_MAX;
        min2 = INT_MAX;
        for(int j=0; j<k; ++j){
            if(dp[j]!=min1_old || min1_old==min2_old){
                dp[j] = min1_old + costs[i][j];
            }else{//min1_old occurred when painting house i-1 with color j, so it
cannot be added to dp[j]
                dp[j] = min2_old + costs[i][j];
            }

            if(min1<=dp[j]){
                min2 = min(min2, dp[j]);
            }else{
                min2 = min1;
                min1 = dp[j];
            }
        }
    }

    return min1;
}
```

written by kelly original link here

Answer 3

```java
public class Solution {
    public int minCostII(int[][] costs) {
        if (costs.length == 0 || costs[0].length == 0) {
            return 0;
        }
        int m = costs.length, n = costs[0].length, m1 = 0, m2 = 0;
        int[] dp = new int[n];
        for (int i = 0; i < m; i++) {
            int t1 = m1, t2 = m2;
            m1 = Integer.MAX_VALUE;
            m2 = Integer.MAX_VALUE;
            for (int j = 0; j < n; j++) {
                dp[j] = (dp[j] == t1 ? t2 : t1) + costs[i][j];
                if (m1 <= dp[j]) {
                    m2 = Math.min(dp[j], m2);
                }
                else {
                    m2 = m1;
                    m1 = dp[j];
                }
            }
        }

        return m1;
    }
}
```

written by rjr130 original link here

Palindrome Permutation(266)

Answer 1

The idea is to iterate over string, adding current character to `set` if `set` doesn't contain that character, or removing current character from `set` if `set` contains it. When the iteration is finished, just return `set.size()==0 || set.size()==1`.

`set.size()==0` corresponds to the situation when there are even number of any character in the string, and `set.size()==1` corresponsds to the fact that there are even number of any character except one.

```
public class Solution {
    public boolean canPermutePalindrome(String s) {
        Set<Character> set=new HashSet<Character>();
        for(int i=0; i<s.length(); ++i){
            if (!set.contains(s.charAt(i)))
                set.add(s.charAt(i));
            else
                set.remove(s.charAt(i));
        }
        return set.size()==0 || set.size()==1;
    }
}
```

written by ammv original link here

Answer 2

Just check that no more than one character appears an odd number of times. Because if there is one, then it must be in the middle of the palindrome. So we can't have two of them.

Python

First count all characters in a `Counter`, then count the odd ones.

```
def canPermutePalindrome(self, s):
    return sum(v % 2 for v in collections.Counter(s).values()) < 2
```

Ruby

Using an integer as a bitset (Ruby has arbitrarily large integers).

```
def can_permute_palindrome(s)
  x = s.chars.map { |c| 1 << c.ord }.reduce(0, :^)
  x & x-1 == 0
end
```

C++

Using a bitset.

```
bool canPermutePalindrome(string s) {
    bitset<256> b;
    for (char c : s)
        b.flip(c);
    return b.count() < 2;
}
```

C

Tricky one. Increase `odds` when the increased counter is odd, decrease it otherwise.

```
bool canPermutePalindrome(char* s) {
    int ctr[256] = {}, odds = 0;
    while (*s)
        odds += ++ctr[*s++] & 1 ? 1 : -1;
    return odds < 2;
}
```

Thanks to jianchao.li.fighter for pointing out a nicer way in the comments to which I switched now because it's clearer and faster. Some speed test results (see comments for details):

```
        odds += ++ctr[*s++] % 2 * 2 - 1;       // 1499 ms mean-of-five (my origin
al)
        odds += (ctr[*s++] ^= 1) * 2 - 1;      // 1196 ms mean-of-five
        odds += ++ctr[*s++] % 2 ? 1 : -1;      // 1108 ms mean-of-five
        odds += ((++ctr[*s++] & 1) << 1) - 1;  // 1217 ms mean-of-five
        odds += ++ctr[*s++] & 1 ? 1 : -1;      // 1132 ms mean-of-five
```

Java

Using a BitSet.

```
public boolean canPermutePalindrome(String s) {
    BitSet bs = new BitSet();
    for (byte b : s.getBytes())
        bs.flip(b);
    return bs.cardinality() < 2;
}
```

written by StefanPochmann original link here

Answer 3

```java
public class Solution {
    public boolean canPermutePalindrome(String s) {
        char[] A = new char[256];
        int count=0;
        for(int i=0; i<s.length(); i++){
            if(A[s.charAt(i)]>0)A[s.charAt(i)]--;
            else A[s.charAt(i)]++;
        }
        for(int i=0; i<256; i++){
            if(A[i]!=0)count++;
        }
        return count<=1;
    }
}
```

written by wzhou original link here

Palindrome Permutation II(267)

Answer 1

Basically, the idea is to perform permutation on half of the palindromic string and then form the full palindromic result.

```
public List<String> generatePalindromes(String s) {
    int odd = 0;
    String mid = "";
    List<String> res = new ArrayList<>();
    List<Character> list = new ArrayList<>();
    Map<Character, Integer> map = new HashMap<>();

    // step 1. build character count map and count odds
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);
        odd += map.get(c) % 2 != 0 ? 1 : -1;
    }

    // cannot form any palindromic string
    if (odd > 1) return res;

    // step 2. add half count of each character to list
    for (Map.Entry<Character, Integer> entry : map.entrySet()) {
        char key = entry.getKey();
        int val = entry.getValue();

        if (val % 2 != 0) mid += key;

        for (int i = 0; i < val / 2; i++) list.add(key);
    }

    // step 3. generate all the permutations
    getPerm(list, mid, new boolean[list.size()], new StringBuilder(), res);

    return res;
}

// generate all unique permutation from list
void getPerm(List<Character> list, String mid, boolean[] used, StringBuilder sb,
List<String> res) {
    if (sb.length() == list.size()) {
        // form the palindromic string
        res.add(sb.toString() + mid + sb.reverse().toString());
        sb.reverse();
        return;
    }

    for (int i = 0; i < list.size(); i++) {
        // avoid duplication
        if (i > 0 && list.get(i) == list.get(i - 1) && !used[i - 1]) continue;

        if (!used[i]) {
            used[i] = true; sb.append(list.get(i));
```

```
            // recursion
            getPerm(list, mid, used, sb, res);
            // backtracking
            used[i] = false; sb.deleteCharAt(sb.length() - 1);
        }
    }
}
```

written by jeantimex original link here

Answer 2

```
public List<String> generatePalindromes(String s) {
    int[] map = new int[256];
    for(int i=0;i<s.length();i++){
        map[s.charAt(i)]++;
    }
    int j=0,count=0;
    for(int i=0;i<256;i++){
        if(count== 0 && map[i] %2 == 1){
            j= i;
            count++;
        }else if(map[i] % 2==1){
            return new ArrayList<String>();
        }
    }
    String cur = "";
    if(j != 0){
        cur = ""+ (char)j;
        map[j]--;
    }
    List<String> res = new ArrayList<String>();
    DFS(res,cur,map,s.length());
    return res;
}
public void DFS(List<String> res,String cur,int[] map,int len){
    if(cur.length()== len){
        res.add(new String(cur));
    }else {
        for(int i=0;i<map.length;i++){
            if(map[i] <= 0) continue;
            map[i] = map[i] - 2;
            cur = (char)i + cur + (char)i;
            DFS(res,cur,map,len);
            cur = cur.substring(1,cur.length()-1);
            map[i] = map[i]+2;
        }
    }
}
```

written by zq670067 original link here

Answer 3

Use `collections.Counter` and `itertools.permutations`

```
class Solution(object):
    def generatePalindromes(self, s):
        d = collections.Counter(s)
        m = tuple(k for k, v in d.iteritems() if v % 2)
        p = ''.join(k*(v/2) for k, v in d.iteritems())
        return [''.join(i + m + i[::-1]) for i in set(itertools.permutations(p))]
if len(m) < 2 else []
```

written by xcv58 original link here

Missing Number(268)

Answer 1

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int result = nums.size();
        int i=0;

        for(int num:nums){
            result ^= num;
            result ^= i;
            i++;
        }

        return result;
    }
};
```

There are several similar problems in the problem list.

written by CodingKing original link here

Answer 2

The basic idea is to use XOR operation. We all know that a^b^b =a, which means two xor operations with the same number will eliminate the number and reveal the original number. In this solution, I apply XOR operation to both the index and value of the array. In a complete array with no missing numbers, the index and value should be perfectly corresponding(nums[index] = index), so in a missing array, what left finally is the missing number.

```
public int missingNumber(int[] nums) {

    int xor = 0, i = 0;
    for (i = 0; i < nums.length; i++) {
        xor = xor ^ i ^ nums[i];
    }

    return xor ^ i;
}
```

written by m010 original link here

Answer 3

1.XOR

```java
public int missingNumber(int[] nums) { //xor
    int res = nums.length;
    for(int i=0; i<nums.length; i++){
        res ^= i;
        res ^= nums[i];
    }
    return res;
}
```

2.SUM

```java
public int missingNumber(int[] nums) { //sum
    int len = nums.length;
    int sum = (0+len)*(len+1)/2;
    for(int i=0; i<len; i++)
        sum-=nums[i];
    return sum;
}
```

3.Binary Search

```java
public int missingNumber(int[] nums) { //binary search
    Arrays.sort(nums);
    int left = 0, right = nums.length, mid= (left + right)/2;
    while(left<right){
        mid = (left + right)/2;
        if(nums[mid]>mid) right = mid;
        else left = mid+1;
    }
    return left;
}
```

Summary:

If the array is in order, I prefer `Binary Search` method. Otherwise, the `XOR` method is better.

written by mingjun original link here

Alien Dictionary(269)

Answer 1

Two similar solutions. Both first go through the word list to find letter pairs `(a, b)` where `a` must come before `b` in the alien alphabet. The first solution just works with these pairs, the second is a bit smarter and uses successor lists and predecessor counters. Doesn't make a big difference here, though, I got both accepted in 48 ms.

Solution 1

```
def alienOrder(self, words):
    less = []
    for pair in zip(words, words[1:]):
        for a, b in zip(*pair):
            if a != b:
                less += a + b,
                break
    chars = set(''.join(words))
    order = []
    while less:
        free = chars - set(zip(*less)[1])
        if not free:
            return ''
        order += free
        less = filter(free.isdisjoint, less)
        chars -= free
    return ''.join(order + list(chars))
```

Solution 2

```
def alienOrder(self, words):
    pre, suc = collections.defaultdict(int), collections.defaultdict(set)
    for pair in zip(words, words[1:]):
        for a, b in zip(*pair):
            if a != b:
                suc[a].add(b)
                pre[b] += 1
                break
    chars = set(''.join(words))
    free = chars - set(pre)
    order = ''
    while free:
        a = free.pop()
        order += a
        for b in suc[a]:
            pre[b] -= 1
            if pre[b] == 0:
                free.add(b)
    return order * (set(order) == chars)
```

C++ version of solution 2

```
string alienOrder(vector<string>& words) {
    map<char, set<char>> suc;
    map<char, int> pre;
    set<char> chars;
    string s;
    for (string t : words) {
        chars.insert(t.begin(), t.end());
        for (int i=0; i<min(s.size(), t.size()); ++i) {
            char a = s[i], b = t[i];
            if (a != b) {
                suc[a].insert(b);
                ++pre[b];
                break;
            }
        }
        s = t;
    }
    set<char> free = chars;
    for (auto ci : pre)
        free.erase(ci.first);
    string order;
    while (free.size()) {
        char a = *begin(free);
        free.erase(a);
        order += a;
        for (char b : suc[a])
            if (--pre[b] == 0)
                free.insert(b);
    }
    return order.size() == chars.size() ? order : "";
}
```

written by StefanPochmann original link here

Answer 2

Why do I see this error?

Input: ["wrt","wrf","er","ett","rftt"] Output: "werft" Expected: Special judge: No expected output available.

written by FerociousCodingKitten original link here

Answer 3

The question says: if the input is `[ "wrt", "wrf", "er", "ett", "rftt" ]` The correct order is: `"wertf"`. but from `"rftt"`, f should be lexicographically smaller than t? How can the result be "wertf"? Correct me if I am wrong.

written by AndyLiu0429 original link here

Closest Binary Search Tree Value(270)

Answer 1

Same recursive/iterative solution in different languages.

Recursive

Closest is either the root's value (`a`) or the closest in the appropriate subtree (`b`).

Ruby

```ruby
def closest_value(root, target)
  a = root.val
  kid = target < a ? root.left : root.right or return a
  b = closest_value(kid, target)
  [b, a].min_by { |x| (x - target).abs }
end
```

C++

```cpp
int closestValue(TreeNode* root, double target) {
    int a = root->val;
    auto kid = target < a ? root->left : root->right;
    if (!kid) return a;
    int b = closestValue(kid, target);
    return abs(a - target) < abs(b - target) ? a : b;
}
```

Java

```java
public int closestValue(TreeNode root, double target) {
    int a = root.val;
    TreeNode kid = target < a ? root.left : root.right;
    if (kid == null) return a;
    int b = closestValue(kid, target);
    return Math.abs(a - target) < Math.abs(b - target) ? a : b;
}
```

Python

```python
def closestValue(self, root, target):
    a = root.val
    kid = root.left if target < a else root.right
    if not kid: return a
    b = self.closestValue(kid, target)
    return min((b, a), key=lambda x: abs(target - x))
```

Alternative endings:

```
    return (b, a)[abs(a - target) < abs(b - target)]
    return a if abs(a - target) < abs(b - target) else b
```

Iterative

Walk the path down the tree close to the target, return the closest value on the path. Inspired by yd, I wrote these after reading "while loop".

Ruby

```ruby
def closest_value(root, target)
  path = []
  while root
    path << root.val
    root = target < root.val ? root.left : root.right
  end
  path.reverse.min_by { |x| (x - target).abs }
end
```

The `.reverse` is only for handling targets much larger than 32-bit integer range, where different path values x have the same "distance" `(x - target).abs`. In such cases, the leaf value is the correct answer. If such large targets aren't asked, then it's unnecessary.

Or with O(1) space:

```ruby
def closest_value(root, target)
  closest = root.val
  while root
    closest = [root.val, closest].min_by { |x| (x - target).abs }
    root = target < root.val ? root.left : root.right
  end
  closest
end
```

C++

```cpp
int closestValue(TreeNode* root, double target) {
    int closest = root->val;
    while (root) {
        if (abs(closest - target) >= abs(root->val - target))
            closest = root->val;
        root = target < root->val ? root->left : root->right;
    }
    return closest;
}
```

Python

```
def closestValue(self, root, target):
    path = []
    while root:
        path += root.val,
        root = root.left if target < root.val else root.right
    return min(path[::-1], key=lambda x: abs(target - x))
```

The `[::-1]` is only for handling targets much larger than 32-bit integer range, where different path values x have the same "distance" `(x - target).abs`. In such cases, the leaf value is the correct answer. If such large targets aren't asked, then it's unnecessary.

Or with O(1) space:

```
def closestValue(self, root, target):
    closest = root.val
    while root:
        closest = min((root.val, closest), key=lambda x: abs(target - x))
        root = root.left if target < root.val else root.right
    return closest
```

written by StefanPochmann original link here

Answer 2

```
public int closestValue(TreeNode root, double target) {
        int closestVal = root.val;
        while(root != null){
            //update closestVal if the current value is closer to target
            closestVal = (Math.abs(target - root.val) < Math.abs(target - closest
Val))? root.val : closestVal;
            if(closestVal == target){   //already find the best result
                return closestVal;
            }
            root = (root.val > target)? root.left: root.right;   //binary search
        }
        return closestVal;
  }
```

written by ranylee2 original link here

Answer 3

```
public int closestValue(TreeNode root, double target) {
    int ret = root.val;
    while(root != null){
        if(Math.abs(target - root.val) < Math.abs(target - ret)){
            ret = root.val;
        }
        root = root.val > target? root.left: root.right;
    }
    return ret;
}
```

written by larrywang2014 original link here

Encode and Decode Strings(271)

Answer 1

```
public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for(String s : strs) {
            sb.append(s.length()).append('/').append(s);
        }
        return sb.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> ret = new ArrayList<String>();
        int i = 0;
        while(i < s.length()) {
            int slash = s.indexOf('/', i);
            int size = Integer.valueOf(s.substring(i, slash));
            ret.add(s.substring(slash + 1, slash + size + 1));
            i = slash + size + 1;
        }
        return ret;
    }
}
```

written by qianzhige original link here

Answer 2

The rule is, for each str in strs, encode it as + '@' + str

```
class Codec {
public:

    // Encodes a list of strings to a single string.
    string encode(vector<string>& strs) {
        string encoded = "";
        for (string &str: strs) {
            int len = str.size();
            encoded += to_string(len) + "@" + str;
        }

        return encoded;
    }

    // Decodes a single string to a list of strings.
    vector<string> decode(string s) {
        vector<string> r;
        int head = 0;
        while (head < s.size())    {
            int at_pos = s.find_first_of('@', head);
            int len = stoi(s.substr(head, at_pos - head));
            head = at_pos + 1;
            r.push_back(s.substr(head, len));
            head += len;
        }

        return r;
    }
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.decode(codec.encode(strs));
```

written by sculd original link here

Answer 3

Is anyone else getting this error? I get it for any input that doesn't have syntax errors, even if I just try to compile the blank problem skeleton.

written by omg_zozobra original link here

Closest Binary Search Tree Value II(272)

Answer 1

The idea is to compare the predecessors and successors of the closest node to the target, we can use two stacks to track the predecessors and successors, then like what we do in merge sort, we compare and pick the closest one to the target and put it to the result list.

As we know, inorder traversal gives us sorted predecessors, whereas reverse-inorder traversal gives us sorted successors.

We can use iterative inorder traversal rather than recursion, but to keep the code clean, here is the recursion version.

```
public List<Integer> closestKValues(TreeNode root, double target, int k) {
  List<Integer> res = new ArrayList<>();

  Stack<Integer> s1 = new Stack<>(); // predecessors
  Stack<Integer> s2 = new Stack<>(); // successors

  inorder(root, target, false, s1);
  inorder(root, target, true, s2);

  while (k-- > 0) {
    if (s1.isEmpty())
      res.add(s2.pop());
    else if (s2.isEmpty())
      res.add(s1.pop());
    else if (Math.abs(s1.peek() - target) < Math.abs(s2.peek() - target))
      res.add(s1.pop());
    else
      res.add(s2.pop());
  }

  return res;
}

// inorder traversal
void inorder(TreeNode root, double target, boolean reverse, Stack<Integer> stack)
{
  if (root == null) return;

  inorder(reverse ? root.right : root.left, target, reverse, stack);
  // early terminate, no need to traverse the whole tree
  if ((reverse && root.val <= target) || (!reverse && root.val > target)) return;
  // track the value of current node
  stack.push(root.val);
  inorder(reverse ? root.left : root.right, target, reverse, stack);
}
```

written by jeantimex original link here

Answer 2

```
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        List<Integer> ret = new LinkedList<>();
        Stack<TreeNode> succ = new Stack<>();
        Stack<TreeNode> pred = new Stack<>();
        initializePredecessorStack(root, target, pred);
        initializeSuccessorStack(root, target, succ);
        if(!succ.isEmpty() && !pred.isEmpty() && succ.peek().val == pred.peek().v
al) {
            getNextPredecessor(pred);
        }
        while(k-- > 0) {
            if(succ.isEmpty()) {
                ret.add(getNextPredecessor(pred));
            } else if(pred.isEmpty()) {
                ret.add(getNextSuccessor(succ));
            } else {
                double succ_diff = Math.abs((double)succ.peek().val - target);
                double pred_diff = Math.abs((double)pred.peek().val - target);
                if(succ_diff < pred_diff) {
                    ret.add(getNextSuccessor(succ));
                } else {
                    ret.add(getNextPredecessor(pred));
                }
            }
        }
        return ret;
    }

    private void initializeSuccessorStack(TreeNode root, double target, Stack<Tre
eNode> succ) {
        while(root != null) {
            if(root.val == target) {
                succ.push(root);
                break;
            } else if(root.val > target) {
                succ.push(root);
                root = root.left;
            } else {
                root = root.right;
            }
        }
    }

    private void initializePredecessorStack(TreeNode root, double target, Stack<T
reeNode> pred) {
        while(root != null){
            if(root.val == target){
                pred.push(root);
                break;
            } else if(root.val < target){
                pred.push(root);
                root = root.right;
            } else{
                root = root.left;
            }
```

```
        }
    }

    private int getNextSuccessor(Stack<TreeNode> succ) {
        TreeNode curr = succ.pop();
        int ret = curr.val;
        curr = curr.right;
        while(curr != null) {
            succ.push(curr);
            curr = curr.left;
        }
        return ret;
    }

    private int getNextPredecessor(Stack<TreeNode> pred) {
        TreeNode curr = pred.pop();
        int ret = curr.val;
        curr = curr.left;
        while(curr != null) {
            pred.push(curr);
            curr = curr.right;
        }
        return ret;
    }
}
```

written by qianzhige original link here

Answer 3

```java
public List<Integer> closestKValues(TreeNode root, double target, int k) {
    Deque<TreeNode> bigger = new ArrayDeque<TreeNode>();
    Deque<TreeNode> smaller = new ArrayDeque<TreeNode>();
    TreeNode node = root;
    // log(n)
    while(node != null)
    {
        if(node.val > target)
        {
            bigger.push(node);
            node = node.left;
        }
        else
        {
            smaller.push(node);
            node = node.right;
        }
    }

    // k
    List<Integer> ret = new ArrayList<Integer>();
    while(ret.size() < k)
    {
        if(bigger.isEmpty() ||
           !smaller.isEmpty() &&
            ((bigger.peek().val - target) > (target - smaller.peek().val)))
        {
            node = smaller.pop();
            ret.add(node.val);

            // Get next smaller
            node = node.left;
            while(node != null)
            {
                smaller.push(node);
                node = node.right;
            }
        }
        else
        {
            node = bigger.pop();
            ret.add(node.val);

            // get next bigger
            node = node.right;
            while(node != null)
            {
                bigger.push(node);
                node = node.left;
            }
        }
    }

    return ret;
}
```

written by lotustree86 original link here

Integer to English Words(273)

Answer 1

```
private final String[] lessThan20 = {"", "One", "Two", "Three", "Four", "Five", "
Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen"
, "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
private final String[] tens = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "
Sixty", "Seventy", "Eighty", "Ninety"};
private final String[] thousands = {"", "Thousand", "Million", "Billion"};

public String numberToWords(int num) {
    if (num == 0)
        return "Zero";
    int i = 0;
    String words = "";

    while (num > 0) {
        if (num % 1000 != 0)
            words = helper(num % 1000) + thousands[i] + " " + words;
        num /= 1000;
        i++;
    }

    return words.trim();
}

private String helper(int num) {
    if (num == 0)
        return "";
    else if (num < 20)
        return lessThan20[num] + " ";
    else if (num < 100)
        return tens[num / 10] + " " + helper(num % 10);
    else
        return lessThan20[num / 100] + " Hundred " + helper(num % 100);
}
```

written by hwy_2015 original link here

Answer 2

```
class Solution {
public:
    static string numberToWords(int n) {
        if(n == 0) return "Zero";
        else return int_string(n).substr(1);
    }
private:
    static const char * const below_20[];
    static const char * const below_100[];
    static string int_string(int n) {
        if(n >= 1000000000)   return int_string(n / 1000000000) + " Billion" + in
t_string(n - 1000000000 * (n / 1000000000));
        else if(n >= 1000000) return int_string(n / 1000000) + " Million" + int_s
tring(n - 1000000 * (n / 1000000));
        else if(n >= 1000)    return int_string(n / 1000) + " Thousand" + int_str
ing(n - 1000 * (n / 1000));
        else if(n >= 100)     return int_string(n / 100) + " Hundred" + int_strin
g(n - 100 * (n / 100));
        else if(n >= 20)      return string(" ") + below_100[n / 10 - 2] + int_s
tring(n - 10 * (n / 10));
        else if(n >= 1)       return string(" ") + below_20[n - 1];
        else return "";
        }
    }
};

const char * const Solution::below_20[] =  {"One", "Two", "Three", "Four","Five",
"Six","Seven","Eight","Nine","Ten", "Eleven","Twelve","Thirteen","Fourteen","Fift
een","Sixteen","Seventeen","Eighteen","Nineteen"};
const char * const Solution::below_100[] = {"Twenty", "Thirty", "Forty", "Fifty",
"Sixty", "Seventy", "Eighty", "Ninety"};
```

written by popffabrik original link here

Answer 3

```
def numberToWords(self, num):
    to19 = 'One Two Three Four Five Six Seven Eight Nine Ten Eleven Twelve ' \
           'Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen Nineteen'.split(
)
    tens = 'Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety'.split()
    def words(n):
        if n < 20:
            return to19[n-1:n]
        if n < 100:
            return [tens[n/10-2]] + words(n%10)
        if n < 1000:
            return [to19[n/100-1]] + ['Hundred'] + words(n%100)
        for p, w in enumerate(('Thousand', 'Million', 'Billion'), 1):
            if n < 1000**(p+1):
                return words(n/1000**p) + [w] + words(n%1000**p)
    return ' '.join(words(num)) or 'Zero'
```

written by StefanPochmann original link here

H-Index(274)

Answer 1

```
public class Solution {
    // 9.3 70 years diaoZhaTian China jiaYou
    public int hIndex(int[] citations) {
        int length = citations.length;
        if (length == 0) {
            return 0;
        }

        int[] array2 = new int[length + 1];
        for (int i = 0; i < length; i++) {
            if (citations[i] > length) {
                array2[length] += 1;
            } else {
                array2[citations[i]] += 1;
            }
        }
        int t = 0;
        int result = 0;

        for (int i = length; i >= 0; i--) {
            t = t + array2[i];
            if (t >= i) {
                return i;
            }
        }
        return 0;
    }
}
```

written by pythonicjava original link here

Answer 2

```java
public int hIndex(int[] citations) {
    int len = citations.length;
    int[] count = new int[len + 1];

    for (int c: citations)
        if (c > len)
            count[len]++;
        else
            count[c]++;


    int total = 0;
    for (int i = len; i >= 0; i--) {
        total += count[i];
        if (total >= i)
            return i;
    }

    return 0;
}
```

written by jinrf original link here

Answer 3

```java
 public int hIndex(int[] citations) {
   Arrays.sort(citations);
   int len=citations.length;
    for(int i=0;i<len;i++){
        if(citations[i]>=len-i) return len-i;

    }
    return 0;
}
```

written by dezhihe original link here

H-Index II(275)

Answer 1

Just binary search, each time check citations[mid] case 1: citations[mid] == len-mid, then it means there are citations[mid] papers that have at least citations[mid] citations. case 2: citations[mid] > len-mid, then it means there are citations[mid] papers that have moret than citations[mid] citations, so we should continue searching in the left half case 3: citations[mid] < len-mid, we should continue searching in the right side After iteration, it is guaranteed that right+1 is the one we need to find (i.e. len-(right+1) papars have at least len-(righ+1) citations)

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        int left=0, len = citations.size(), right= len-1,  mid;
        while(left<=right)
        {
            mid=(left+right)>>1;
            if(citations[mid]== (len-mid)) return citations[mid];
            else if(citations[mid] > (len-mid)) right = mid - 1;
            else left = mid + 1;
        }
        return len - (right+1);
    }
};
```

written by dong.wang.1694 original link here

Answer 2

The basic idea comes from the description**h of his/her N papers have at least h citations each**. Therefore, we know `if "mid + 1" is a valid h index, it means value of position "citationsSize - mid - 1" must exceed "mid"`. After we find a valid h index, we go on searching on the right part to see if we can find a larger h index. If it's not a valid h index, the h index can be found in the left part and we simply follow the standard binary search to solve this problem.

```
int hIndex(int* citations, int citationsSize) {
    int lo = 0, hi = citationsSize, mid, index = 0;
    while (lo <= hi) {
        mid = lo + ((hi - lo) >> 1);
        if (citations[citationsSize - mid - 1] > mid) {
            lo = mid + 1;
            index = lo;
        } else {
            hi = mid - 1;
        }
    }
    return index;
}
```

written by hdchen original link here

Answer 3

The basic idea of this solution is to use **binary search** to find the minimum `index` such that

```
citations[index] >= length(citations) - index
```

After finding this `index`, the answer is `length(citations) - index`.

This logic is very similar to the C++ function `lower_bound` or `upper_bound`.

Complexities:

- Time: O(log n)
- Space: O(1)

C++:

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        int size = citations.size();

        int first = 0;
        int mid;
        int count = size;
        int step;

        while (count > 0) {
            step = count / 2;
            mid = first + step;
            if (citations[mid] < size - mid) {
                first = mid + 1;
                count -= (step + 1);
            }
            else {
                count = step;
            }
        }

        return size - first;
    }
};
```

Java:

```java
public class Solution {
    public int hIndex(int[] citations) {
        int len = citations.length;

        int first = 0;
        int mid;
        int count = len;
        int step;

        while (count > 0) {
            step = count / 2;
            mid = first + step;
            if (citations[mid] < len - mid) {
                first = mid + 1;
                count -= (step + 1);
            }
            else {
                count = step;
            }
        }

        return len - first;
    }
}
```

Python:

```python
class Solution(object):
    def hIndex(self, citations):
        """
        :type citations: List[int]
        :rtype: int
        """

        length = len(citations)

        first = 0
        count = length

        while count > 0:
            step = count / 2
            mid = first + step
            if citations[mid] < length - mid:
                first = mid + 1
                count -= (step + 1)
            else:
                count = step

        return length - first
```

@daviantan1890 @ruichang Thank you for your comments and discussions.

I am very sure that two-branch binary search is more efficient than three branch binary search. and (low + high) is not good idea since it may rely on the overflow behavior. In fact, using `count` `step` `first` `mid` is the standard implement way of C++, so I do not think there are better ways to implement the binary search.

written by zhiqing_xiao original link here

Paint Fence(276)

Answer 1

```
public int numWays(int n, int k) {
    if(n == 0) return 0;
    else if(n == 1) return k;
    int diffColorCounts = k*(k-1);
    int sameColorCounts = k;
    for(int i=2; i<n; i++) {
        int temp = diffColorCounts;
        diffColorCounts = (diffColorCounts + sameColorCounts) * (k-1);
        sameColorCounts = temp;
    }
    return diffColorCounts + sameColorCounts;
}
```

We divided it into two cases.

1. the last two posts have the same color, the number of ways to paint in this case is *sameColorCounts*.
2. the last two posts have different colors, and the number of ways in this case is *diffColorCounts*.

The reason why we have these two cases is that we can easily compute both of them, and that is all I do. When adding a new post, we can use the same color as the last one (if allowed) or different color. If we use different color, there're *k-1* options, and the outcomes shoule belong to the *diffColorCounts* category. If we use same color, there's only one option, and we can only do this when the last two have different colors (which is the diffColorCounts). There we have our induction step.

Here is an example, let's say we have 3 posts and 3 colors. The first two posts we have 9 ways to do them, (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3). Now we know that

```
diffColorCounts = 6;
```

And

```
sameColorCounts = 3;
```

Now for the third post, we can compute these two variables like this:

If we use different colors than the last one (the second one), these ways can be added into *diffColorCounts*, so if the last one is 3, we can use 1 or 2, if it's 1, we can use 2 or 3, etc. Apparently there are `(diffColorCounts + sameColorCounts) * (k-1)` possible ways.

If we use the same color as the last one, we would trigger a violation in these three cases (1,1,1), (2,2,2) and (3,3,3). This is because they already used the same color for

the last two posts. So is there a count that rules out these kind of cases? YES, the *diffColorCounts*. So in cases within *diffColorCounts*, we can use the same color as the last one without worrying about triggering the violation. And now as we append a same-color post to them, the former *diffColorCounts* becomes the current *sameColorCounts*.

Then we can keep going until we reach the n. And finally just sum up these two variables as result.

Hope this would be clearer.

written by JennyShaw original link here

Answer 2

If n == 1, there would be k-ways to paint.

if n == 2, there would be two situations:

- 2.1 You paint same color with the previous post: k*1 ways to paint, named it as `same`
- 2.2 You paint differently with the previous post: k*(k-1) ways to paint this way, named it as `dif`

So, you can think, if n >= 3, you can always maintain these two situations, `You either paint the same color with the previous one, or differently`.

Since there is a rule: "no more than two adjacent fence posts have the same color."

We can further analyze:

- from 2.1, since previous two are in the same color, next one you could only paint differently, and it would form one part of "paint differently" case in the n == 3 level, and the number of ways to paint this way would equal to `same*(k-1)`.
- from 2.2, since previous two are not the same, you can either paint the same color this time (`dif*1`) ways to do so, or stick to paint differently (`dif*(k-1)`) times.

Here you can conclude, when seeing back from the next level, ways to paint the same, or variable `same` would equal to `dif*1 = dif`, and ways to paint differently, variable `dif`, would equal to `same*(k-1)+dif*(k-1) = (same + dif)*(k-1)`

So we could write the following codes:

```
    if n == 0:
        return 0
    if n == 1:
        return k
    same, dif = k, k*(k-1)
    for i in range(3, n+1):
        same, dif = dif, (same+dif)*(k-1)
    return same + dif
```

written by chungyushao original link here

Answer 3

Need two one-dimensional array dp1 and dp2, dp1[i] means the number of solutions when the color of last two fences (whose indexes are i-1,i-2) are same. dp2[i] means the number of solutions when the color of last two fences are different.

So

dp1[i]=dp2[i-1],

dp2[i]=(k-1)*(dp1[i-1]+dp2[i-1]) =(k-1)*(dp2[i-2]+dp2[i-1])

Final result is dp1[n-1]+dp2[n-1];

In the code, variable *a,c* mean the last two items of dp1, variable *b,d* mean the last two items of dp2, and c could be eliminated.

```
class Solution {
public:
    int numWays(int n, int k) {
        if(n<=1 || k==0)return n*k;
        int a=k,b=k*(k-1),c=0,d=0;
        for(int i=2;i<n;++i){
            d=(k-1)*(a+b);
            a=b;b=d;
        }
        return a+b;
    }
};
```

written by jiannan original link here

Find the Celebrity(277)

Answer 1

The idea is as follows:

first, if person A knows person B, then B could be the candidate of being a celebrity, A must not be a celebrity. We iterate all the n persons and we will have a candidate that everyone knows this candidate.

second, we check two things after we get this candidate. 1. If this candidate knows other person in the group, if the candidate knows anyone in the group, then the candidate is not celebrity, return -1; 2. if everyone knows this candidate, if anyone does not know the candidate, return -1;

// Forward declaration of the knows API.

bool knows(int a, int b);

class Solution {

public:

```
int findCelebrity(int n) {
    if(n<=1) return n;

    int candidate = 0;

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

        if ( !knows(i,candidate) ){
            candidate = i;
        }
    }


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

        if(j== candidate) continue;

        if( !knows(j,candidate) || knows(candidate,j) ){
              //if j does not know candidate, or candidate knows j, return -1;
            return -1;
        }

    }


    return candidate;

}
```

};

written by hbsophia original link here

Answer 2

The first pass is to pick out the candidate. If candidate knows i, then switch candidate. The second pass is to check whether the candidate is real.

```
public class Solution extends Relation {
    public int findCelebrity(int n) {
        int candidate = 0;
        for(int i = 1; i < n; i++){
            if(knows(candidate, i))
                candidate = i;
        }
        for(int i = 0; i < n; i++){
            if(i != candidate && (knows(candidate, i) || !knows(i, candidate))) r
eturn -1;
        }
        return candidate;
    }
}
```

written by czonzhu original link here

Answer 3

```java
public int findCelebrity(int n) {
    // base case
    if (n <= 0) return -1;
    if (n == 1) return 0;

    Stack<Integer> stack = new Stack<>();

    // put all people to the stack
    for (int i = 0; i < n; i++) stack.push(i);

    int a = 0, b = 0;

    while (stack.size() > 1) {
        a = stack.pop(); b = stack.pop();

        if (knows(a, b))
            // a knows b, so a is not the celebrity, but b may be
            stack.push(b);
        else
            // a doesn't know b, so b is not the celebrity, but a may be
            stack.push(a);
    }

    // double check the potential celebrity
    int c = stack.pop();

    for (int i = 0; i < n; i++)
        // c should not know anyone else
        if (i != c && (knows(c, i) || !knows(i, c)))
            return -1;

    return c;
}
```

written by jeantimex original link here

First Bad Version(278)

Answer 1

The binary search code:

```
public int firstBadVersion(int n) {
    int start = 1, end = n;
    while (start < end) {
        int mid = start + (end-start) / 2;
        if (!isBadVersion(mid)) start = mid + 1;
        else end = mid;
    }
    return start;
}
```

written by Pixel_ original link here

Answer 2

```
class Solution {
public:
    int firstBadVersion(int n) {
        int lower = 1, upper = n, mid;
        while(lower < upper) {
            mid = lower + (upper - lower) / 2;
            if(!isBadVersion(mid)) lower = mid + 1;   /* Only one call to API */
            else upper = mid;
        }
        return lower;   /* Because there will alway be a bad version, return lowe
r here */
    }
};
```

written by whaleking1990 original link here

Answer 3

Is there any difference between " (low + high) / 2 " and " low + (high - low) / 2 "?

When I use the first one, it told me "time limit exceed" but if I use the second one, it worked!

written by aaronwei original link here

Perfect Squares(279)

Answer 1

Came up with the 2 solutions of breadth-first search and dynamic programming. Also "copied" StefanPochmann's static dynamic programming solution (https://leetcode.com/discuss/56993/static-dp-c-12-ms-python-172-ms-ruby-384-ms) and davidtan1890's mathematical solution (https://leetcode.com/discuss/57066/4ms-c-code-solve-it-mathematically) here with minor style changes and some comments. Thank Stefan and David for posting their nice solutions!

1.Dynamic Programming: 440ms

```
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        // cntPerfectSquares[i] = the least number of perfect square numbers
        // which sum to i. Note that cntPerfectSquares[0] is 0.
        vector<int> cntPerfectSquares(n + 1, INT_MAX);
        cntPerfectSquares[0] = 0;
        for (int i = 1; i <= n; i++)
        {
            // For each i, it must be the sum of some number (i - j*j) and
            // a perfect square number (j*j).
            for (int j = 1; j*j <= i; j++)
            {
                cntPerfectSquares[i] =
                    min(cntPerfectSquares[i], cntPerfectSquares[i - j*j] + 1);
            }
        }

        return cntPerfectSquares.back();
    }
};
```

2.Static Dynamic Programming: 12ms

```cpp
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        // cntPerfectSquares[i] = the least number of perfect square numbers
        // which sum to i. Since cntPerfectSquares is a static vector, if
        // cntPerfectSquares.size() > n, we have already calculated the result
        // during previous function calls and we can just return the result now.
        static vector<int> cntPerfectSquares({0});

        // While cntPerfectSquares.size() <= n, we need to incrementally
        // calculate the next result until we get the result for n.
        while (cntPerfectSquares.size() <= n)
        {
            int m = cntPerfectSquares.size();
            int cntSquares = INT_MAX;
            for (int i = 1; i*i <= m; i++)
            {
                cntSquares = min(cntSquares, cntPerfectSquares[m - i*i] + 1);
            }

            cntPerfectSquares.push_back(cntSquares);
        }

        return cntPerfectSquares[n];
    }
};
```

3.Mathematical Solution: 4ms

```
class Solution
{
private:
    int is_square(int n)
    {
        int sqrt_n = (int)(sqrt(n));
        return (sqrt_n*sqrt_n == n);
    }

public:
    // Based on Lagrange's Four Square theorem, there
    // are only 4 possible results: 1, 2, 3, 4.
    int numSquares(int n)
    {
        // If n is a perfect square, return 1.
        if(is_square(n))
        {
            return 1;
        }

        // The result is 4 if n can be written in the
        // form of 4^k*(8*m + 7). Please refer to
        // Legendre's three-square theorem.
        while ((n & 3) == 0) // n%4 == 0
        {
            n >>= 2;
        }
        if ((n & 7) == 7) // n%8 == 7
        {
            return 4;
        }

        // Check whether 2 is the result.
        int sqrt_n = (int)(sqrt(n));
        for(int i = 1; i <= sqrt_n; i++)
        {
            if (is_square(n - i*i))
            {
                return 2;
            }
        }

        return 3;
    }
};
```

4.Breadth-First Search: 80ms

```
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
```

```
        return 0;
    }

    // perfectSquares contain all perfect square numbers which
    // are smaller than or equal to n.
    vector<int> perfectSquares;
    // cntPerfectSquares[i - 1] = the least number of perfect
    // square numbers which sum to i.
    vector<int> cntPerfectSquares(n);

    // Get all the perfect square numbers which are smaller than
    // or equal to n.
    for (int i = 1; i*i <= n; i++)
    {
        perfectSquares.push_back(i*i);
        cntPerfectSquares[i*i - 1] = 1;
    }

    // If n is a perfect square number, return 1 immediately.
    if (perfectSquares.back() == n)
    {
        return 1;
    }

    // Consider a graph which consists of number 0, 1,...,n as
    // its nodes. Node j is connected to node i via an edge if
    // and only if either j = i + (a perfect square number) or
    // i = j + (a perfect square number). Starting from node 0,
    // do the breadth-first search. If we reach node n at step
    // m, then the least number of perfect square numbers which
    // sum to n is m. Here since we have already obtained the
    // perfect square numbers, we have actually finished the
    // search at step 1.
    queue<int> searchQ;
    for (auto& i : perfectSquares)
    {
        searchQ.push(i);
    }

    int currCntPerfectSquares = 1;
    while (!searchQ.empty())
    {
        currCntPerfectSquares++;

        int searchQSize = searchQ.size();
        for (int i = 0; i < searchQSize; i++)
        {
            int tmp = searchQ.front();
            // Check the neighbors of node tmp which are the sum
            // of tmp and a perfect square number.
            for (auto& j : perfectSquares)
            {
                if (tmp + j == n)
                {
                    // We have reached node n.
                    return currCntPerfectSquares;
                }
```

```
                    else if ((tmp + j < n) && (cntPerfectSquares[tmp + j - 1] ==
0))
                    {
                        // If cntPerfectSquares[tmp + j - 1] > 0, this is not
                        // the first time that we visit this node and we should
                        // skip the node (tmp + j).
                        cntPerfectSquares[tmp + j - 1] = currCntPerfectSquares;
                        searchQ.push(tmp + j);
                    }
                    else if (tmp + j > n)
                    {
                        // We don't need to consider the nodes which are greater
]
                        // than n.
                        break;
                    }
                }

                searchQ.pop();
            }
        }

        return 0;
    }
};
```

written by zhukov original link here

Answer 2

These solutions use some number theory (see explanation further down).

Ruby solution

```
require 'prime'

def num_squares(n)
  n /= 4 while n % 4 == 0
  return 4 if n % 8 == 7
  return 3 if n.prime_division.any? { |p, e| p % 4 == 3 && e.odd? }
  (n**0.5).to_i**2 == n ? 1 : 2
end
```

Or:

```ruby
require 'prime'

def num_squares(n)
  n /= 4 while n % 4 == 0
  return 4 if n % 8 == 7
  pd = n.prime_division
  return 3 if pd.any? { |p, e| p % 4 == 3 && e.odd? }
  pd.any? { |_, e| e.odd? } ? 2 : 1
end
```

C++ solution

```cpp
int numSquares(int n) {
    while (n % 4 == 0)
        n /= 4;
    if (n % 8 == 7)
        return 4;
    bool min2 = false;
    for (int i=2; i<=n; ++i) {
        if (i > n/i)
            i = n;
        int e = 0;
        while (n % i == 0)
            n /= i, ++e;
        if (e % 2 && i % 4 == 3)
            return 3;
        min2 |= e % 2;
    }
    return 1 + min2;
}
```

C solution

Inspired by kevin36's solution. We don't really need to compute the prime factorization. Knowing that four squares always suffice and using the three-squares test is enough. Single-square and sum-of-two-squares cases can be done simpler.

```cpp
int numSquares(int n) {
    while (n % 4 == 0)
        n /= 4;
    if (n % 8 == 7)
        return 4;
    for (int a=0; a*a<=n; ++a) {
        int b = sqrt(n - a*a);
        if (a*a + b*b == n)
            return 1 + !!a;
    }
    return 3;
}
```

Explanation

I happen to have given a little talk about just this topic a while back in a number theory seminar. This problem is completely solved, in the sense of being reduced to simple checks of a number's prime factorization. A natural number is...

- ... a **square** if and only if each prime factor occurs to an even power in the number's prime factorization.
- ... a **sum of two squares** if and only if each prime factor that's 3 modulo 4 occurs to an even power in the number's prime factorization.
- ... a **sum of three squares** if and only if it's not of the form $4^a(8b+7)$ with integers a and b.
- ... a **sum of four squares**. Period. No condition. You never need more than four.

Of course single squares can also be identified by comparing a given number with the square of the rounded root of the number.

The problem statement says *"1, 4, 9, 16, ..."*, for some reason apparently excluding 0, but it really is a perfect square and the above theorems do consider it one. With that, you can for example always extend a sum of two squares a^2+b^2 to the sum of three squares $a^2+b^2+0^2$. Put differently, if n isn't a sum of three squares, then it also isn't a sum of two squares. So you can read the above statements as *"... a sum of m **(or fewer)** squares"*. Thanks to ruben3 for asking about this in the comments.

In my above solutions, I first divide the given number by 4 as often as possible and then do the three-squares check. Dividing by 4 doesn't affect the other checks, and the n % 8 == 7 is cheaper than the prime factorization, so this saves time in cases where we do need four squares.

Armed with just the knowledge that you never need more than four squares, it's also easy to write O(n) solutions, e.g.:

```
int numSquares(int n) {
    int ub = sqrt(n);
    for (int a=0; a<=ub; ++a) {
        for (int b=a; b<=ub; ++b) {
            int c = sqrt(n - a*a - b*b);
            if (a*a + b*b + c*c == n)
                return !!a + !!b + !!c;
        }
    }
    return 4;
}
```

written by StefanPochmann original link here

Answer 3

dp[n] indicates that the perfect squares count of the given n, and we have:

```
dp[0] = 0
dp[1] = dp[0]+1 = 1
dp[2] = dp[1]+1 = 2
dp[3] = dp[2]+1 = 3
dp[4] = Min{ dp[4-1*1]+1, dp[4-2*2]+1 }
      = Min{ dp[3]+1, dp[0]+1 }
      = 1
dp[5] = Min{ dp[5-1*1]+1, dp[5-2*2]+1 }
      = Min{ dp[4]+1, dp[1]+1 }
      = 2
                        .
                        .
                        .
dp[13] = Min{ dp[13-1*1]+1, dp[13-2*2]+1, dp[13-3*3]+1 }
       = Min{ dp[12]+1, dp[9]+1, dp[4]+1 }
       = 2
                        .
                        .
                        .
dp[n] = Min{ dp[n - i*i] + 1 },  n - i*i >=0 && i >= 1
```

and the sample code is like below:

```
public int numSquares(int n) {
    int[] dp = new int[n + 1];
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    for(int i = 1; i <= n; ++i) {
        int min = Integer.MAX_VALUE;
        int j = 1;
        while(i - j*j >= 0) {
            min = Math.min(min, dp[i - j*j] + 1);
            ++j;
        }
        dp[i] = min;
    }
    return dp[n];
}
```

Hope it can help to understand the DP solution.

written by Karci original link here

Wiggle Sort(280)

Answer 1

```
public class Solution {
    public void wiggleSort(int[] nums) {
        for(int i=0;i<nums.length;i++)
            if(i%2==1){
               if(nums[i-1]>nums[i]) swap(nums, i);
            }else if(i!=0 && nums[i-1]<nums[i]) swap(nums, i);
    }
    public void swap(int[] nums, int i){
          int tmp=nums[i];
          nums[i]=nums[i-1];
          nums[i-1]=tmp;
    }
}
```

written by WalkerIX original link here

Answer 2

The final sorted `nums` needs to satisfy two conditions:

1. If `i` is odd, then `nums[i] >= nums[i - 1]`;
2. If `i` is even, then `nums[i] <= nums[i - 1]`.

The code is just to fix the orderings of `nums` that do not satisfy 1 and 2.

```
class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        int n = nums.size();
        for (int i = 1; i < n; i++)
            if (((i & 1) && nums[i] < nums[i - 1]) || (!(i & 1) && nums[i] > nums
[i - 1]))
                swap(nums[i], nums[i - 1]);
    }
};
```

written by jianchao.li.fighter original link here

Answer 3

```java
public class Solution {
    public void wiggleSort(int[] nums) {
        for (int i = 1; i < nums.length; i++)
            if ((i % 2 == 1 && nums[i] < nums[i - 1]) || (i % 2 == 0 && nums[i] >
nums[i - 1])) {
                int tmp = nums[i];
                nums[i] = nums[i - 1];
                nums[i - 1] = tmp;
            }
    }
}
```

written by shuatidaxia original link here

Zigzag Iterator(281)

Answer 1

```
class ZigzagIterator {
public:
    ZigzagIterator(vector<int>& v1, vector<int>& v2) {
        if (v1.size() != 0)
            Q.push(make_pair(v1.begin(), v1.end()));
        if (v2.size() != 0)
            Q.push(make_pair(v2.begin(), v2.end()));
    }

    int next() {
        vector<int>::iterator it = Q.front().first;
        vector<int>::iterator endIt = Q.front().second;
        Q.pop();
        if (it + 1 != endIt)
            Q.push(make_pair(it+1, endIt));
        return *it;
    }

    bool hasNext() {
        return !Q.empty();
    }
private:
    queue<pair<vector<int>::iterator, vector<int>::iterator>> Q;
};
```

somehow similar to BFS.

written by lightmark original link here

Answer 2

Two iterators, one for each list. Switching them *before* reading the next number instead of afterwards saves a bit of code, I think.

```
public class ZigzagIterator {

    private Iterator<Integer> i, j, tmp;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        i = v2.iterator();
        j = v1.iterator();
    }

    public int next() {
        if (j.hasNext()) { tmp = j; j = i; i = tmp; }
        return i.next();
    }

    public boolean hasNext() {
        return i.hasNext() || j.hasNext();
    }
}
```

written by StefanPochmann original link here

Answer 3

Uses a linkedlist to store the iterators in different vectors. Every time we call next(), we pop an element from the list, and re-add it to the end to cycle through the lists.

```
public class ZigzagIterator {
    LinkedList<Iterator> list;
    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        list = new LinkedList<Iterator>();
        if(!v1.isEmpty()) list.add(v1.iterator());
        if(!v2.isEmpty()) list.add(v2.iterator());
    }

    public int next() {
        Iterator poll = list.remove();
        int result = (Integer)poll.next();
        if(poll.hasNext()) list.add(poll);
        return result;
    }

    public boolean hasNext() {
        return !list.isEmpty();
    }
}
```

written by kevinhsu original link here

Expression Add Operators(282)

Answer 1

This problem has a lot of edge cases to be considered:

1. overflow: we use a long type once it is larger than Integer.MAX_VALUE or minimum, we get over it.
2. 0 sequence: because we can't have numbers with multiple digits started with zero, we have to deal with it too.
3. a little trick is that we should save the value that is to be multiplied in the next recursion.

```
public class Solution {
    public List<String> addOperators(String num, int target) {
        List<String> rst = new ArrayList<String>();
        if(num == null || num.length() == 0) return rst;
        helper(rst, "", num, target, 0, 0, 0);
        return rst;
    }
    public void helper(List<String> rst, String path, String num, int target, int
pos, long eval, long multed){
        if(pos == num.length()){
            if(target == eval)
                rst.add(path);
            return;
        }
        for(int i = pos; i < num.length(); i++){
            if(i != pos && num.charAt(pos) == '0') break;
            long cur = Long.parseLong(num.substring(pos, i + 1));
            if(pos == 0){
                helper(rst, path + cur, num, target, i + 1, cur, cur);
            }
            else{
                helper(rst, path + "+" + cur, num, target, i + 1, eval + cur , cu
r);

                helper(rst, path + "-" + cur, num, target, i + 1, eval -cur, -cur
);

                helper(rst, path + "*" + cur, num, target, i + 1, eval - multed +
multed * cur, multed * cur );
            }
        }
    }
}
```

written by czonzhu original link here

Answer 2

```
class Solution {
private:
    // cur: {string} expression generated so far.
    // pos: {int}    current visiting position of num.
    // cv:  {long}   cumulative value so far.
    // pv:  {long}   previous operand value.
    // op:  {char}   previous operator used.
    void dfs(std::vector<string>& res, const string& num, const int target, strin
g cur, int pos, const long cv, const long pv, const char op) {
        if (pos == num.size() && cv == target) {
            res.push_back(cur);
        } else {
            for (int i=pos+1; i<=num.size(); i++) {
                string t = num.substr(pos, i-pos);
                long now = stol(t);
                if (to_string(now).size() != t.size()) continue;
                dfs(res, num, target, cur+'+'+t, i, cv+now, now, '+');
                dfs(res, num, target, cur+'-'+t, i, cv-now, now, '-');
                dfs(res, num, target, cur+'*'+t, i, (op == '-') ? cv+pv - pv*now
: ((op == '+') ? cv-pv + pv*now : pv*now), pv*now, op);
            }
        }
    }

public:
    vector<string> addOperators(string num, int target) {
        vector<string> res;
        if (num.empty()) return res;
        for (int i=1; i<=num.size(); i++) {
            string s = num.substr(0, i);
            long cur = stol(s);
            if (to_string(cur).size() != s.size()) continue;
            dfs(res, num, target, s, i, cur, cur, '#');         // no operator de
fined.
        }

        return res;
    }
};
```

written by cmyzx original link here

Answer 3

```
void addOperators(vector<string>& result, string nums, string t, long long last,
long long curVal, int target) {
    if (nums.length() == 0) {
        if (curVal == target)
            result.push_back(t);
        return;
    }

    for (int i = 1; i<=nums.length(); i++) {
        string num = nums.substr(0, i);
        if(num.length() > 1 && num[0] == '0')
            return;

        string nextNum = nums.substr(i);

        if (t.length() > 0) {
            addOperators(result, nextNum, t + "+" + num, stoll(num), curVal + sto
ll(num), target);
            addOperators(result, nextNum, t + "-" + num, -stoll(num), curVal - st
oll(num), target);
            addOperators(result, nextNum, t + "*" + num, last * stoll(num), (curV
al - last) + (last * stoll(num)), target);
        }
        else
            addOperators(result, nextNum, num, stoll(num), stoll(num), target);
    }
}

vector<string> addOperators(string num, int target) {
    vector<string> result;
    addOperators(result, num, "", 0, 0, target);
    return result;
}
```

written by jaewoo original link here

Move Zeroes(283)

Answer 1

```
// Shift non-zero values as far forward as possible
// Fill remaining space with zeros

public void moveZeroes(int[] nums) {
    if (nums == null || nums.length == 0) return;

    int insertPos = 0;
    for (int num: nums) {
        if (num != 0) nums[insertPos++] = num;
    }

    while (insertPos < nums.length) {
        nums[insertPos++] = 0;
    }
}
```

written by Kurteck original link here

Answer 2

```
void moveZeroes(vector<int>& nums) {
    int last = 0, cur = 0;

    while(cur < nums.size()) {
        if(nums[cur] != 0) {
            swap(nums[last], nums[cur]);
            last++;
        }

        cur++;
    }
}
```

written by jaewoo original link here

Answer 3

The idea comes from the c++ erase/remove idiom.

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        fill(remove(nums.begin(), nums.end(),0), nums.end(), 0);
    }
};
```

written by youbuer original link here

Peeking Iterator(284)

Answer 1

```
class PeekingIterator implements Iterator<Integer> {
    private Integer next = null;
    private Iterator<Integer> iter;

    public PeekingIterator(Iterator<Integer> iterator) {
        // initialize any member here.
        iter = iterator;
        if (iter.hasNext())
            next = iter.next();
    }

    // Returns the next element in the iteration without advancing the iterator.
    public Integer peek() {
        return next;
    }

    // hasNext() and next() should behave the same as in the Iterator interface.
    // Override them if needed.
    @Override
    public Integer next() {
        Integer res = next;
        next = iter.hasNext() ? iter.next() : null;
        return res;
    }

    @Override
    public boolean hasNext() {
        return next != null;
    }
}
```

cache the next element. If next is null, there is no more elements in iterator.

written by chouclee original link here

Answer 2

Since `Iterator` has a copy constructor, we can just use it:

```
class PeekingIterator : public Iterator
{
public:
    PeekingIterator(const vector<int> &nums) : Iterator(nums)
    {
    }

    int peek()
    {
        return Iterator(*this).next();
    }

    int next()
    {
        return Iterator::next();
    }

    bool hasNext() const
    {
        return Iterator::hasNext();
    }
};
```

written by efanzh original link here

Answer 3

```java
class PeekingIterator implements Iterator<Integer> {
    Integer n = null;
    private Iterator<Integer> iterator = null;
    public PeekingIterator(Iterator<Integer> iterator) {
        this.iterator = iterator;
    }

    public Integer peek() {
        if (n == null && iterator.hasNext()){
            n = iterator.next();
        }
        return n;
    }
    public Integer next() {
        if (n!=null){
            int temp = n;
            n = null;
            return temp;
        }
        return iterator.next();
    }
    public boolean hasNext() {
        if (n!=null){
            return true;
        }
        return iterator.hasNext();
    }
}
```

written by qgambit2 original link here

Inorder Successor in BST(285)

Answer 1

Just want to share my recursive solution for both getting the successor and predecessor for a given node in BST.

Successor

```
public TreeNode successor(TreeNode root, TreeNode p) {
  if (root == null)
    return null;

  if (root.val <= p.val) {
    return successor(root.right, p);
  } else {
    TreeNode left = successor(root.left, p);
    return (left != null) ? left : root;
  }
}
```

Predecessor

```
public TreeNode predecessor(TreeNode root, TreeNode p) {
  if (root == null)
    return null;

  if (root.val >= p.val) {
    return predecessor(root.left, p);
  } else {
    TreeNode right = predecessor(root.right, p);
    return (right != null) ? right : root;
  }
}
```

written by jeantimex original link here

Answer 2

Update: Ugh, turns out I didn't think it through and the big case distinction is unnecessary. Just search from root to bottom, trying to find the smallest node larger than p and return the last one that was larger. D'oh. Props to smileyogurt.966 for doing that first, I think. I'll just write it in my ternary style for C++:

```
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    TreeNode* candidate = NULL;
    while (root)
        root = (root->val > p->val) ? (candidate = root)->left : root->right;
    return candidate;
}
```

Old: If `p` has a right subtree, then get its successor from there. Otherwise do a regular search from `root` to `p` but remember the node of the last left-turn and return that. Same solution as everyone, I guess, just written a bit shorter. Runtime O(h), where h is the height of the tree.

C++

```
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    if (p->right) {
        p = p->right;
        while (p->left)
            p = p->left;
        return p;
    }
    TreeNode* candidate = NULL;
    while (root != p)
        root = (p->val > root->val) ? root->right : (candidate = root)->left;
    return candidate;
}
```

Java

```
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    if (p.right != null) {
        p = p.right;
        while (p.left != null)
            p = p.left;
        return p;
    }
    TreeNode candidate = null;
    while (root != p)
        root = (p.val > root.val) ? root.right : (candidate = root).left;
    return candidate;
}
```

written by StefanPochmann original link here

Answer 3

The inorder traversal of a BST is the nodes in ascending order. To find a successor, you just need to find the smallest one that is larger than the given value since there are no duplicate values in a BST. It just like the binary search in a sorted list. The time complexity should be `O(h)` where h is the depth of the result node. `succ` is a pointer that keeps the possible successor. Whenever you go left the current root is the new possible successor, otherwise the it remains the same.

Only in a balanced BST `O(h) = O(log n)`. In the worst case `h` can be as large as `n`.

Java

```java
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    TreeNode succ = null;
    while (root != null) {
        if (p.val < root.val) {
            succ = root;
            root = root.left;
        }
        else
            root = root.right;
    }
    return succ;
}

// 29 / 29 test cases passed.
// Status: Accepted
// Runtime: 5 ms
```

Python

```python
def inorderSuccessor(self, root, p):
    succ = None
    while root:
        if p.val < root.val:
            succ = root
            root = root.left
        else:
            root = root.right
    return succ

# 29 / 29 test cases passed.
# Status: Accepted
# Runtime: 112 ms
```

written by dietpepsi original link here

Walls and Gates(286)

Answer 1

Push all gates into queue first. Then for each gate update its neighbor cells and push them to the queue.

Repeating above steps until there is nothing left in the queue.

```
public class Solution {
    public void wallsAndGates(int[][] rooms) {
        if (rooms.length == 0 || rooms[0].length == 0) return;
        Queue<int[]> queue = new LinkedList<>();
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) queue.add(new int[]{i, j});
            }
        }
        while (!queue.isEmpty()) {
            int[] top = queue.remove();
            int row = top[0], col = top[1];
            if (row > 0 && rooms[row - 1][col] == Integer.MAX_VALUE) {
                rooms[row - 1][col] = rooms[row][col] + 1;
                queue.add(new int[]{row - 1, col});
            }
            if (row < rooms.length - 1 && rooms[row + 1][col] == Integer.MAX_VALU
E) {
                rooms[row + 1][col] = rooms[row][col] + 1;
                queue.add(new int[]{row + 1, col});
            }
            if (col > 0 && rooms[row][col - 1] == Integer.MAX_VALUE) {
                rooms[row][col - 1] = rooms[row][col] + 1;
                queue.add(new int[]{row, col - 1});
            }
            if (col < rooms[0].length - 1 && rooms[row][col + 1] == Integer.MAX_V
ALUE) {
                rooms[row][col + 1] = rooms[row][col] + 1;
                queue.add(new int[]{row, col + 1});
            }
        }
    }
}
```

written by chase1991 original link here

Answer 2

```
public class Solution {
int[][] dir ={{0,1},{0,-1},{1,0},{-1,0}};
public void wallsAndGates(int[][] rooms) {
    for(int i=0;i<rooms.length;i++){
        for(int j=0;j<rooms[0].length;j++){
            if(rooms[i][j]==0)
                bfs(rooms,i,j);
        }
    }
}
public void bfs(int[][] rooms,int i,int j){
    for(int[] d:dir){
        if(i+d[0]>=0 && i+d[0]<rooms.length && j+d[1]>=0 && j+d[1]<rooms[0].lengt
h && rooms[i+d[0]][j+d[1]]>rooms[i][j]+1){
            rooms[i+d[0]][j+d[1]]=rooms[i][j]+1;
            bfs(rooms,i+d[0],j+d[1]);
        }
    }
}
```

}

written by DREAM123 original link here

Answer 3

```
void wallsAndGates(vector<vector<int>>& rooms) {
    const int row = rooms.size();
    if (0 == row) return;
    const int col = rooms[0].size();
    queue<pair<int, int>> canReach;  // save all element reachable
    vector<pair<int, int>> dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}; // four dir
ections for each reachable
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            if(0 == rooms[i][j])
                canReach.emplace(i, j);
        }
    }
    while(!canReach.empty()){
        int r = canReach.front().first, c = canReach.front().second;
        canReach.pop();
        for (auto dir : dirs) {
            int x = r + dir.first,  y = c + dir.second;
            // if x y out of range or it is obstasle, or has small distance aread
y
            if (x < 0 || y < 0 || x >= row || y >= col || rooms[x][y] <= rooms[r]
[c]+1) continue;
            rooms[x][y] = rooms[r][c] + 1;
            canReach.emplace(x, y);
        }
    }
}
```

written by lchen77 original link here

Find the Duplicate Number(287)

Answer 1

The main idea is the same with problem***Linked List Cycle II***,*https://leetcode.com/problems/linked-list-cycle-ii/*. Use two pointers the fast and the slow. The fast one goes forward two steps each time, while the slow one goes only step each time. They must meet the same item when slow==fast. In fact, they meet in a circle, the duplicate number must be the entry point of the circle when visiting the array from nums[0]. Next we just need to find the entry point. We use a point(we can use the fast one before) to visit form begining with one step each time, do the same job to slow. When fast==slow, they meet at the entry point of the circle. The easy understood code is as follows.

```
int findDuplicate3(vector<int>& nums)
{
    if (nums.size() > 1)
    {
        int slow = nums[0];
        int fast = nums[nums[0]];
        while (slow != fast)
        {
            slow = nums[slow];
            fast = nums[nums[fast]];
        }

        fast = 0;
        while (fast != slow)
        {
            fast = nums[fast];
            slow = nums[slow];
        }
        return slow;
    }
    return -1;
}
```

written by echoxiaolee original link here

Answer 2

This solution is based on binary search.

At first the search space is numbers between 1 to n. Each time I select a number `mid` (which is the one in the middle) and count all the numbers equal to or less than `mid`. Then if the `count` is more than `mid`, the search space will be `[1 mid]` otherwise `[mid+1 n]`. I do this until search space is only one number.

Let's say `n=10` and I select `mid=5`. Then I count all the numbers in the array which are less than equal `mid`. If the there are more than `5` numbers that are less than `5`, then by Pigeonhole Principle (https://en.wikipedia.org/wiki/Pigeonhole_principle) one of them has occurred more than once. So I shrink the search space from `[1`

`10]` to `[1 5]`. Otherwise the duplicate number is in the second half so for the next step the search space would be `[6 10]`.

```
class Solution(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        low = 1
        high = len(nums)-1

        while low < high:
            mid = low+(high-low)/2
            count = 0
            for i in nums:
                if i <= mid:
                    count+=1
            if count <= mid:
                low = mid+1
            else:
                high = mid
        return low
```

There's also a better algorithm with `O(n)` time. Please read this very interesting solution here: http://keithschwarz.com/interesting/code/?dir=find-duplicate

written by mehran original link here

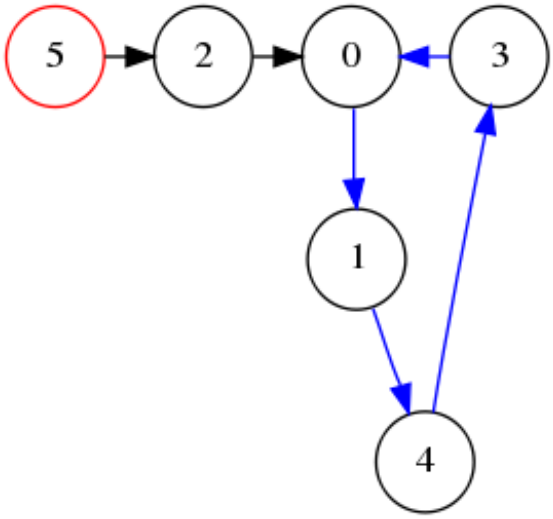
Answer 3

suppose the array is

> index: 0 1 2 3 4 5
>
> value: 2 5 1 1 4 3

first subtract 1 from each element in the array, so it is much easy to understand. use the value as pointer. the array becomes:

> index: 0 1 2 3 4 5
>
> value: 1 4 0 0 3 2



Second if the array is

> index: 0 1 2 3 4 5
>
> value: 0 1 2 4 2 3

we must choose the last element as the head of the linked list. If we choose 0, we can not detect the cycle.

Now the problem is the same as find the cycle in linkedlist!

```
public int findDuplicate(int[] nums) {
    int n = nums.length;
    for(int i=0;i<nums.length;i++) nums[i]--;
    int slow = n-1;
    int fast = n-1;
    do{
        slow = nums[slow];
        fast = nums[nums[fast]];
    }while(slow != fast);
    slow = n-1;
    while(slow != fast){
        slow = nums[slow];
        fast = nums[fast];
    }
    return slow+1;
}
```

One condition is we cannot modify the array. So the solution is

```java
public int findDuplicate(int[] nums) {
    int n = nums.length;
    int slow = n;
    int fast = n;
    do{
        slow = nums[slow-1];
        fast = nums[nums[fast-1]-1];
    }while(slow != fast);
    slow = n;
    while(slow != fast){
        slow = nums[slow-1];
        fast = nums[fast-1];
    }
    return slow;
}
```

written by zq670067 original link here

Unique Word Abbreviation(288)

Answer 1

To check for unique abbreviation, we maintain a mapping from a specific abbreviation to all words which have the abbreviation. Then we just need to check no other words have the same abbreviation as the given word.

The code is as follows.

```
class ValidWordAbbr {
public:
    ValidWordAbbr(vector<string> &dictionary) {
        for (string& d : dictionary) {
            int n = d.length();
            string abbr = d[0] + to_string(n) + d[n - 1];
            mp[abbr].insert(d);
        }
    }

    bool isUnique(string word) {
        int n = word.length();
        string abbr = word[0] + to_string(n) + word[n - 1];
        return mp[abbr].count(word) == mp[abbr].size();
    }
private:
    unordered_map<string, unordered_set<string>> mp;
};


// Your ValidWordAbbr object will be instantiated and called as such:
// ValidWordAbbr vwa(dictionary);
// vwa.isUnique("hello");
// vwa.isUnique("anotherWord");
```

written by jianchao.li.fighter original link here

Answer 2

```
public class ValidWordAbbr {

Map<String, String> map = new HashMap<>();

public ValidWordAbbr(String[] dictionary) {
    for (String dic : dictionary) {
        String key = getKey(dic);
        if (map.containsKey(key)) {
            map.put(key, "");
        } else {
            map.put(key, dic);
        }
    }
}

public boolean isUnique(String word) {
    String key = getKey(word);
    return !map.containsKey(key) || map.get(key).equals(word);
}

private String getKey(String word) {
    String key = word.charAt(0) + Integer.toString(word.length() - 2) + word.char
At(word.length() - 1);
    return key;
}
```

}

written by shuatidaxia original link here

Answer 3

The idea is pretty straightforward, we use a map to track a set of words that have the same abbreviation. The word is unique when its abbreviation does not exist in the map or it's the only one in the set.

```java
public class ValidWordAbbr {
  Map<String, Set<String>> map = new HashMap<>();

  public ValidWordAbbr(String[] dictionary) {
    // build the hashmap
    // the key is the abbreviation
    // the value is a hash set of the words that have the same abbreviation
    for (int i = 0; i < dictionary.length; i++) {
      String a = abbr(dictionary[i]);
      Set<String> set = map.containsKey(a) ? map.get(a) : new HashSet<>();
      set.add(dictionary[i]);
      map.put(a, set);
    }
  }

  public boolean isUnique(String word) {
    String a = abbr(word);
    // it's unique when the abbreviation does not exist in the map or
    // it's the only word in the set
    return !map.containsKey(a) || (map.get(a).contains(word) && map.get(a).size()
== 1);
  }

  String abbr(String s) {
    if (s.length() < 3) return s;
    return s.substring(0, 1) + String.valueOf(s.length() - 2) + s.substring(s.len
gth() - 1);
  }
}
```

written by jeantimex original link here

Game of Life(289)

Answer 1

Since the board has ints but only the 1-bit is used, I use the 2-bit to store the new state. At the end, replace the old state with the new state by shifting all values one bit to the right.

```
void gameOfLife(vector<vector<int>>& board) {
    int m = board.size(), n = m ? board[0].size() : 0;
    for (int i=0; i<m; ++i) {
        for (int j=0; j<n; ++j) {
            int count = 0;
            for (int I=max(i-1, 0); I<min(i+2, m); ++I)
                for (int J=max(j-1, 0); J<min(j+2, n); ++J)
                    count += board[I][J] & 1;
            if (count == 3 || count - board[i][j] == 3)
                board[i][j] |= 2;
        }
    }
    for (int i=0; i<m; ++i)
        for (int j=0; j<n; ++j)
            board[i][j] >>= 1;
}
```

Note that the above `count` counts the live ones among a cell's neighbors and the cell itself. Starting with `int count = -board[i][j]` counts only the live neighbors and allows the neat

```
if ((count | board[i][j]) == 3)
```

test. Thanks to aileenbai for showing that one in the comments.

written by StefanPochmann original link here

Answer 2

To solve it in place, we use 2 bits to store 2 states:

```
[2nd bit, 1st bit] = [next state, current state]

- 00 dead (current) -> dead (next)
- 01 live (current) -> dead (next)
- 10 dead (current) -> live (next)
- 11 live (current) -> live (next)
```

In the beginning, every **2nd** state is `0`; when next becomes alive change **2nd** bit to `1`:

- live -> die: `nbs < 2 || nbs > 3` (*we don't need to care*!)
- live -> live: `nbs >= 2 && nbs <= 3`
- dead -> live: `nbs == 3`

To get this state, we simple do:

```
board[i][j] & 1
```

To get next state, we simple do:

```
board[i][j] >> 1
```

Hope this helps!

```
public void gameOfLife(int[][] board) {
    if(board == null || board.length == 0) return;
    int m = board.length, n = board[0].length;

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            int lives = liveNeighbors(board, m, n, i, j);

            // In the beginning, every 2nd bit is 0;
            // So we only need to care about when 2nd bit will become 1.
            if((board[i][j] & 1) == 1 && (lives >= 2 && lives <= 3)) {
                board[i][j] = 3; // Make the 2nd bit 1: 01 ---> 11
            }
            if((board[i][j] & 1) == 0 && lives == 3) {
                board[i][j] = 2; // Make the 2nd bit 1: 00 ---> 10
            }
        }
    }

    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            board[i][j] >>= 1;  // Get the 2nd state.
        }
    }
}

public int liveNeighbors(int[][] board, int m, int n, int i, int j) {
    int lives = 0;
    for(int p = Math.max(i - 1, 0); p <= Math.min(i + 1, m - 1); p++) {
        for(int q = Math.max(j - 1, 0); q <= Math.min(j + 1, n - 1); q++) {
            lives += board[p][q] & 1;
        }
    }
    lives -= board[i][j] & 1;
    return lives;
}
```

written by yavinci original link here

Answer 3

```
public class Solution {
int[][] dir ={{1,-1},{1,0},{1,1},{0,-1},{0,1},{-1,-1},{-1,0},{-1,1}};
public void gameOfLife(int[][] board) {
    for(int i=0;i<board.length;i++){
        for(int j=0;j<board[0].length;j++){
            int live=0;
            for(int[] d:dir){
                if(d[0]+i<0 || d[0]+i>=board.length || d[1]+j<0 || d[1]+j>=board[
0].length) continue;
                if(board[d[0]+i][d[1]+j]==1 || board[d[0]+i][d[1]+j]==2) live++;
            }
            if(board[i][j]==0 && live==3) board[i][j]=3;
            if(board[i][j]==1 && (live<2 || live>3)) board[i][j]=2;
        }
    }
    for(int i=0;i<board.length;i++){
        for(int j=0;j<board[0].length;j++){
            board[i][j] %=2;
        }
    }
}
```

}

written by DREAM123 original link here

Word Pattern(290)

Answer 1

```
public boolean wordPattern(String pattern, String str) {
    String[] words = str.split(" ");
    if (words.length != pattern.length())
        return false;
    Map index = new HashMap();
    for (Integer i=0; i<words.length; ++i)
        if (index.put(pattern.charAt(i), i) != index.put(words[i], i))
            return false;
    return true;
}
```

I go through the pattern letters and words in parallel and compare the indexes where they last appeared.

Edit 1: Originally I compared the **first** indexes where they appeared, using `putIfAbsent` instead of `put`. That was based on mathsam's solution for the old Isomorphic Strings problem. But then czonzhu's answer below made me realize that `put` works as well and why.

Edit 2: Switched from

```
    for (int i=0; i<words.length; ++i)
        if (!Objects.equals(index.put(pattern.charAt(i), i),
                            index.put(words[i], i)))
            return false;
```

to the current version with `i` being an `Integer` object, which allows to compare with just `!=` because there's no autoboxing-same-value-to-different-objects-problem anymore. Thanks to lap_218 for somewhat pointing that out in the comments.

written by StefanPochmann original link here

Answer 2

I think all previous C++ solutions read all words into a vector at the start. Here I read them on the fly.

```
bool wordPattern(string pattern, string str) {
    map<char, int> p2i;
    map<string, int> w2i;
    istringstream in(str);
    int i = 0, n = pattern.size();
    for (string word; in >> word; ++i) {
        if (i == n || p2i[pattern[i]] != w2i[word])
            return false;
        p2i[pattern[i]] = w2i[word] = i + 1;
    }
    return i == n;
}
```

written by StefanPochmann original link here

Answer 3

This problem is pretty much equivalent to Isomorphic Strings. Let me reuse two old solutions.

From here:

```
def wordPattern(self, pattern, str):
    s = pattern
    t = str.split()
    return map(s.find, s) == map(t.index, t)
```

Improved version also from there:

```
def wordPattern(self, pattern, str):
    f = lambda s: map({}.setdefault, s, range(len(s)))
    return f(pattern) == f(str.split())
```

From here:

```
def wordPattern(self, pattern, str):
    s = pattern
    t = str.split()
    return len(set(zip(s, t))) == len(set(s)) == len(set(t)) and len(s) == len(t)
```

Thanks to zhang38 for pointing out the need to check len(s) == len(t) here.

written by StefanPochmann original link here

Word Pattern II(291)

Answer 1

We can solve this problem using backtracking, we just have to keep trying to use a character in the pattern to match different length of substrings in the input string, keep trying till we go through the input string and the pattern.

For example, input string is `"redblueredblue"`, and the pattern is `"abab"`, first let's use `'a'` to match `"r"`, `'b'` to match `"e"`, then we find that `'a'` does not match `"d"`, so we do backtracking, use `'b'` to match `"ed"`, so on and so forth ...

When we do the recursion, if the pattern character exists in the hash map already, we just have to see if we can use it to match the same length of the string. For example, let's say we have the following map:

`'a': "red"`

`'b': "blue"`

now when we see `'a'` again, we know that it should match `"red"`, the length is `3`, then let's see if `str[i ... i+3]` matches `'a'`, where `i` is the current index of the input string. Thanks to StefanPochmann's suggestion, in Java we can elegantly use `str.startsWith(s, i)` to do the check.

Also thanks to i-tikhonov's suggestion, we can use a hash set to avoid duplicate matches, if character `a` matches string `"red"`, then character `b` cannot be used to match `"red"`. In my opinion though, we can say apple (pattern `'a'`) is "fruit", orange (pattern `'o'`) is "fruit", so they can match the same string, anyhow, I guess it really depends on how the problem states.

The following code should pass OJ now, if we don't need to worry about the duplicate matches, just remove the code that associates with the hash set.

```
public class Solution {

  public boolean wordPatternMatch(String pattern, String str) {
    Map<Character, String> map = new HashMap<>();
    Set<String> set = new HashSet<>();

    return isMatch(str, 0, pattern, 0, map, set);
  }

  boolean isMatch(String str, int i, String pat, int j, Map<Character, String> ma
p, Set<String> set) {
    // base case
    if (i == str.length() && j == pat.length()) return true;
    if (i == str.length() || j == pat.length()) return false;

    // get current pattern character
    char c = pat.charAt(j);

    // if the pattern character exists
    if (map.containsKey(c)) {
```

```
      String s = map.get(c);

      // then check if we can use it to match str[i...i+s.length()]
      if (!str.startsWith(s, i)) {
        return false;
      }

      // if it can match, great, continue to match the rest
      return isMatch(str, i + s.length(), pat, j + 1, map, set);
    }

    // pattern character does not exist in the map
    for (int k = i; k < str.length(); k++) {
      String p = str.substring(i, k + 1);

      if (set.contains(p)) {
        continue;
      }

      // create or update it
      map.put(c, p);
      set.add(p);

      // continue to match the rest
      if (isMatch(str, k + 1, pat, j + 1, map, set)) {
        return true;
      }

      // backtracking
      map.remove(c);
      set.remove(p);
    }

    // we've tried our best but still no luck
    return false;
  }

}
```

written by jeantimex original link here

Answer 2

```
public class Solution {
Map<Character,String> map =new HashMap();
Set<String> set =new HashSet();
public boolean wordPatternMatch(String pattern, String str) {
    if(pattern.isEmpty()) return str.isEmpty();
    if(map.containsKey(pattern.charAt(0))){
        String value= map.get(pattern.charAt(0));
        if(str.length()<value.length() || !str.substring(0,value.length()).equals
(value)) return false;
        if(wordPatternMatch(pattern.substring(1),str.substring(value.length())))
return true;
    }else{
        for(int i=1;i<=str.length();i++){
            if(set.contains(str.substring(0,i))) continue;
            map.put(pattern.charAt(0),str.substring(0,i));
            set.add(str.substring(0,i));
            if(wordPatternMatch(pattern.substring(1),str.substring(i))) return tr
ue;
            set.remove(str.substring(0,i));
            map.remove(pattern.charAt(0));
        }
    }
    return false;
}
```

}

written by DREAM123 original link here

Answer 3

```
class Solution {
public:
    unordered_map<char, string> pDict;
    unordered_map<string, char> sDict;
    bool wordPatternMatch(string pattern, string str) {
        return match(pattern, 0, str, 0);
    }

    bool match(string &pattern, int i, string &str, int j) {
        int m = pattern.size();
        int n = str.size();
        if (i == m || j == n) {
            if (i == m && j == n)
                return true;
            return false;
        }
        bool ins = false;
        for (int k = j; k < n; k++) {
            string s = str.substr(j, k - j + 1);
            if (pDict.find(pattern[i]) != pDict.end()) {
                if (pDict[pattern[i]] != s)
                    continue;
            } else if (sDict.find(s) != sDict.end()) {
                if (sDict[s] != pattern[i])
                    continue;
            } else {
                pDict[pattern[i]] = s;
                sDict[s] = pattern[i];
                ins = true;
            }
            if (match(pattern, i + 1, str, k + 1))
                return true;
            if (ins) {
                pDict.erase(pattern[i]);
                sDict.erase(s);
            }
        }
        return false;
    }
};
```

C++ backtracking. ins indicates whether current round has inserted new mapping pair. edited with two maps to ensure on-to-one mapping. .

written by lightmark original link here

Nim Game(292)

Answer 1

> Theorem: The first one who got the number that is multiple of 4 (i.e. n % 4 == 0) will lost, otherwise he/she will win.

Proof:

1. the base case: when `n = 4`, as suggested by the hint from the problem, no matter which number that that first player, the second player would always be able to pick the remaining number.
2. For `1* 4 < n < 2 * 4, (n = 5, 6, 7)`, the first player can reduce the initial number into 4 accordingly, which will leave the death number 4 to the second player. i.e. The numbers 5, 6, 7 are winning numbers for any player who got it first.
3. Now to the beginning of the next cycle, `n = 8`, no matter which number that the first player picks, it would always leave the winning numbers (5, 6, 7) to the second player. Therefore, 8 % 4 == 0, again is a death number.
4. Following the second case, for numbers between (2*4 = 8) and (3*4=12), which are `9, 10, 11`, are winning numbers for the first player again, because the first player can always reduce the number into the death number 8.

Following the above theorem and proof, the solution could not be simpler:

```
public boolean canWinNim(int n) {
    return n % 4 != 0 ;
}
```

written by liaison original link here

Answer 2

suppose there are x stones left for first player (A), he can take 1,2,3 stones away, so second player B will have three cases to deal with (x -1), (x-2), (x-3). after he pick the stones, there will be 9 cases left for A.

```
B (x-1) -> A: (x-2), (x-3), (x-4)
B (x-2) -> A: (x-3), (x-4), (x-5)
B (x-3) -> A: (x-4), (x-5), (x-6)
```

Now, if A can guarantee he win at either of three groups, then he can force B to into that one of the three states and A can end up in that particular group after B's move.

```
f(x) = (f(x-2)&&f(x-3)&&f(x-4)) || (f(x-3)&&f(x-4)&&f(x-5)) || (f(x-4)&&f(x-5)&&f
(x-6))
```

if we examine the equation a little closer, we can find f(x - 4) is a critical point, if f(x-4) is false, then f(x) will be always false.

we can also find out the initial conditions, f(1), f(2), f(3) will be true (A always win), and f(4) will be false. so based on previous equation and initial conditions f(5) = f(6) = f(7) = true, f(8) = false; obviously, f(1), f(2), f(3) can make all f(4n + 1), f(4n + 2), f(4n + 3) to be true, only f(4n) will be false then. so here we go our one line solution:

return (n % 4 != 0);

written by kennethliaoke original link here

Answer 3

DP : Line 7: java.lang.OutOfMemoryError: Java heap space

```
public boolean canWinNim(int n) {
    if(n <= 0)
        throw new IllegalArgumentException();
    if(n < 4)
        return true;
    boolean[] res = new boolean[n + 1];
    res[0] = true;
    res[1] = true;
    res[2] = true;
    res[3] = true;
    for(int i = 4 ; i <= n ; i++)
        res[i] = !(res[i - 1] && res[i - 2] && res[i - 3]);
    return res[n];
}
```

Directly

```
if(n <= 0)
    throw new IllegalArgumentException();
return !(n % 4 == 0);
```

written by TianhaoSong original link here

Flip Game(293)

Answer 1

```
public List<String> generatePossibleNextMoves(String s) {
    List list = new ArrayList();
    for (int i=-1; (i = s.indexOf("++", i+1)) >= 0; )
        list.add(s.substring(0, i) + "--" + s.substring(i+2));
    return list;
}
```

written by StefanPochmann original link here

Answer 2

We start from the second character of the input string and check whether current and previous characters are both equal to '+'. If true, then we combine the characters : characters before previous character + '--' + characters after current character.

```
public List<String> generatePossibleNextMoves(String s) {

    List<String> list = new ArrayList<String>();

    for (int i = 1; i < s.length(); i++) {
        if (s.charAt(i) == '+' && s.charAt(i - 1) == '+') {
            list.add(s.substring(0, i - 1) + "--" + s.substring(i + 1, s.length()
));
        }
    }

    return list;

}
```

written by lDreaml original link here

Answer 3

```
def generatePossibleNextMoves(self, s):
    return [s[:i] + "--" + s[i + 2:] for i in xrange(len(s) - 1) if s[i:i + 2] ==
'++']


# 25 / 25 test cases passed.
# Status: Accepted
# Runtime: 44 ms
```

It is a simple list comprehension and a filter

written by dietpepsi original link here

Flip Game II(294)

Answer 1

At first glance, backtracking seems to be the only feasible solution to this problem. We can basically try every possible move for the first player (Let's call him 1P from now on), and recursively check if the second player 2P has any chance to win. If 2P is guaranteed to lose, then we know the current move 1P takes must be the winning move. The implementation is actually very simple:

```
int len;
string ss;
bool canWin(string s) {
    len = s.size();
    ss = s;
    return canWin();
}
bool canWin() {
    for (int is = 0; is <= len-2; ++is) {
        if (ss[is] == '+' && ss[is+1] == '+') {
            ss[is] = '-'; ss[is+1] = '-';
            bool wins = !canWin();
            ss[is] = '+'; ss[is+1] = '+';
            if (wins) return true;
        }
    }
    return false;
}
```

For most interviews, this is the expected solution. Now let's check the time complexity: Suppose originally the board of size N contains only '+' signs, then roughly we have:

```
T(N) = T(N-2) + T(N-3) + [T(2) + T(N-4)] + [T(3) + T(N-5)] + ...
        [T(N-5) + T(3)] + [T(N-4) + T(2)] + T(N-3) + T(N-2)
     = 2 * sum(T[i])  (i = 3..N-2)
```

You will find that T(N) = 2^(N-1) satisfies the above equation. Therefore, this algorithm is at least exponential.

Can we do better than that? Sure! Below I'll show the time complexity can be reduced to O(N^2) using Dynamic Programming, but the improved method requires some non-trivial understanding of the game theory, and therefore is not expected in a real interview. If you are not interested, please simply skip the rest of the article:

> Concept 1 (**Impartial Game**): Given a particular arrangement of the game board, if either player have exactly the same set of moves should he move first, and both players have exactly the same winning condition, then this game is

> called **impartial game**. For example, chess is not impartial because the players can control only their own pieces, and the +- flip game, on the other hand, is impartial.

--

> Concept 2 (**Normal Play vs Misere Play**): If the winning condition of the game is that the **opponent has no valid moves**, then this game is said to follow the **normal play convention**; if, alternatively, the winning condition is that the **player himself has no valid moves,** then the game is a **Misere** game. Our +- flip has apprently normal play.

Now we understand the the flip game is an impartial game under normal play. Luckily, this type of game has been extensively studied. Note that our following discussion only applies to normal impartial games.

In order to simplify the solution, we still need to understand one more concept:

> Concept 3 (**Sprague-Grundy Function**): Suppose x represents a particular arrangement of board, and x*0, x*1, x*2, ... ,x*k represent the board after a valid move, then we define the Sprague-Grundy function as:

```
g(x) = FirstMissingNumber(g(x_0), g(x_1), g(x_2), ... , g(x_k)).
```

> where FirstMissingNumber(y) stands for the smallest positive number that is not in set y. For instance, if g(x*0) = 0, g(x*1) = 0, g(x_k) = 2, then g(x) = FMV({0, 0, 2}) = 1.

Why do we need this bizarre looking S-G function? Because we can instantly decide whether 1P has a winning move simply by looking at its value. I don't want to write a book out of it, so for now, please simply take the following theorem for granted:

> Theorem 1: If g(x) != 0, then 1P must have a guaranteed winning move from board state x. Otherwise, no matter how 1P moves, 2P must then have a winning move.

So our task now is to calculate g(board). But how to do that? Let's first of all find a way to numerically describe the board. Since we can only flip ++ to --, then apparently, we only need to write down the lengths of consecutive ++'s of length >= 2 to define a board. For instance, ++--+-++++-+----- can be represented as (2, 4).

(2, 4) has two separate '+' subsequences. Any operation made on one subsequence does not interfere with the state of the other. Therefore, we say (2, 4) consists of two **subgames**: (2) and (4).

Okay now we are only one more theorem away from the solution. This is the last theorem. I promise:

> Theorem 2 (**Sprague-Grundy Theorem**): The S-G function of game x = (s1, s2, ..., sk) equals the XOR of all its subgames s1, s2, ..., sk. e.g. g((s1, s2, s3)) = g(s1) XOR g(s2) XOR g(s3).

With the S-G theorem, we can now compute any arbitrary g(x). If x contains only one number N (there is only one '+' subsequence), then

```
g(x) = FMV(g(0, N-2), g(1, N-3), g(2, N-4), ... , g(N/2-1, N-N/2-2));
     = FMV(g(0)^g(N-2), g(1)^g(N-3), g(2)^g(N-4)), ... g(N/2-1, N-N/2-2));
```

Now we have the whole algorithm:

```
Convert the board to numerical representation: x = (s1, s2, ..., sk)
Calculate g(0) to g(max(si)) using DP.
if g(s1)^g(s2)^...^g(sk) != 0 return true, otherwise return false.
```

Calculating g(N) takes O(N) time (N/2 XOR operations plus the O(N) First Missing Number algorithm). And we must calculate from g(1) all the way to g(N). So overall, the algorithm has an O(N^2) time complexity.

Naturally, the code is bit more complicated than the backtracking version. But it reduces the running time from ~128ms to less than 1ms. The huge improvement is definitely worth all the hassle we went through:

```
int firstMissingNumber(unordered_set<int> lut) {
    int m = lut.size();
    for (int i = 0; i < m; ++i) {
        if (lut.count(i) == 0) return i;
    }
    return m;
}

bool canWin(string s) {
    int curlen = 0, maxlen = 0;
    vector<int> board_init_state;
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] == '+') curlen++;              // Find the length of all continu
ous '+' signs
        if (i+1 == s.size() || s[i] == '-') {
            if (curlen >= 2) board_init_state.push_back(curlen);    // only lengt
h >= 2 counts
            maxlen = max(maxlen, curlen);       // Also get the maximum continuou
s length
            curlen = 0;
        }
    }          // For instance ++--+--++++-+ will be represented as (2, 4)
    vector<int> g(maxlen+1, 0);    // Sprague-Grundy function of 0 ~ maxlen
    for (int len = 2; len <= maxlen; ++len) {
        unordered_set<int> gsub;    // the S-G value of all subgame states
        for (int len_first_game = 0; len_first_game < len/2; ++len_first_game) {
            int len_second_game = len - len_first_game - 2;
            // Theorem 2: g[game] = g[subgame1]^g[subgame2]^g[subgame3]...;
            gsub.insert(g[len_first_game] ^ g[len_second_game]);
        }
        g[len] = firstMissingNumber(gsub);
    }

    int g_final = 0;
    for (auto& s: board_init_state) g_final ^= g[s];
    return g_final != 0;    // Theorem 1: First player must win iff g(current_sta
te) != 0
 }
```

---

written by stellari original link here

Answer 2

The idea is try to replace every `"++"` in the current string `s` to `"--"` and see if the opponent can win or not, if the opponent cannot win, great, we win!

For the time complexity, here is what I thought, let's say the length of the input string `s` is `n`, there are at most `n - 1` ways to replace `"++"` to `"--"` (imagine `s` is all `"+++..."`), once we replace one `"++"`, there are at most `(n - 2) - 1` ways to do the replacement, it's a little bit like solving the N-Queens problem, the time complexity is `(n - 1) x (n - 3) x (n - 5) x ...`, so it's `O(n!!)`, double factorial.

That's what I thought, but I could be wrong :)

```
public boolean canWin(String s) {
  if (s == null || s.length() < 2) {
    return false;
  }

  for (int i = 0; i < s.length() - 1; i++) {
    if (s.startsWith("++", i)) {
      String t = s.substring(0, i) + "--" + s.substring(i + 2);

      if (!canWin(t)) {
        return true;
      }
    }
  }

  return false;
}
```

written by jeantimex original link here

Answer 3

```
public boolean canWin(String s) {
    List<String> list = new ArrayList<>();
    for(int i = 0; i < s.length() - 1; i++){
        if(s.charAt(i) == '+' && s.charAt(i + 1) == '+')
            list.add(s.substring(0, i) + "--" + s.substring(i + 2, s.length()));
// generate all possible sequence after every attempt
    }
    /*if(list.isEmpty())
        return false;*/
    for(String str : list){
        if(!canWin(str))              // if there is any one way the next player c
an't win, take it and you'll win
            return true;
    }
    return false;
}
```

written by skyflash original link here

Find Median from Data Stream(295)

Answer 1

I keep two heaps (or priority queues):

- Max-heap `small` has the smaller half of the numbers.
- Min-heap `large` has the larger half of the numbers.

This gives me direct access to the one or two middle values (they're the tops of the heaps), so getting the median takes O(1) time. And adding a number takes O(log n) time.

Supporting both min- and max-heap is more or less cumbersome, depending on the language, so I simply negate the numbers in the heap in which I want the reverse of the default order. To prevent this from causing a bug with -2^{31} (which negated is itself, when using 32-bit ints), I use integer types larger than 32 bits.

Using larger integer types also prevents an overflow error when taking the mean of the two middle numbers. I think almost all solutions posted previously have that bug.

Update: These are pretty short already, but by now I wrote even shorter ones.

Java

```
class MedianFinder {

    private Queue<Long> small = new PriorityQueue(),
                        large = new PriorityQueue();

    public void addNum(int num) {
        large.add((long) num);
        small.add(-large.poll());
        if (large.size() < small.size())
            large.add(-small.poll());
    }

    public double findMedian() {
        return large.size() > small.size()
               ? large.peek()
               : (large.peek() - small.peek()) / 2.0;
    }
};
```

Props to larrywang2014's solution for making me aware that I can use Queue in the declaration instead of PriorityQueue (that's all I got from him, though (just saying because I just saw he changed his previously longer addNum and it's now equivalent to mine)).

C++

```
class MedianFinder {
    priority_queue<long> small, large;
public:

    void addNum(int num) {
        small.push(num);
        large.push(-small.top());
        small.pop();
        if (small.size() < large.size()) {
            small.push(-large.top());
            large.pop();
        }
    }

    double findMedian() {
        return small.size() > large.size()
               ? small.top()
               : (small.top() - large.top()) / 2.0;
    }
};
```

Big thanks to jianchao.li.fighter for telling me that C++'s priority_queue is a max-queue (see comments below).

Python

```
from heapq import *

class MedianFinder:

    def __init__(self):
        self.heaps = [], []

    def addNum(self, num):
        small, large = self.heaps
        heappush(small, -heappushpop(large, num))
        if len(large) < len(small):
            heappush(large, -heappop(small))

    def findMedian(self):
        small, large = self.heaps
        if len(large) > len(small):
            return float(large[0])
        return (large[0] - small[0]) / 2.0
```

written by StefanPochmann original link here

Answer 2

Not sure why it is marked as hard, i think this is one of the easiest questions on leetcode.

```
class MedianFinder {
    // max queue is always larger or equal to min queue
    PriorityQueue<Integer> min = new PriorityQueue();
    PriorityQueue<Integer> max = new PriorityQueue(1000, Collections.reverseOrder
());
    // Adds a number into the data structure.
    public void addNum(int num) {
        max.offer(num);
        min.offer(max.poll());
        if (max.size() < min.size()){
            max.offer(min.poll());
        }
    }

    // Returns the median of current data stream
    public double findMedian() {
        if (max.size() == min.size()) return (max.peek() + min.peek()) /  2.0;
        else return max.peek();
    }
};
```

written by kennethliaoke original link here

Answer 3

Same idea as before, but really exploiting the symmetry of the two heaps by switching them whenever a number is added. Still O(log n) for adding and O(1) for median. Partially inspired by peisi's updated solution.

Update: Added a new Java version (the first one).

Java

```
class MedianFinder {

    Queue<Integer> q = new PriorityQueue(), z = q, t,
                   Q = new PriorityQueue(Collections.reverseOrder());

    public void addNum(int num) {
        (t=Q).add(num);
        (Q=q).add((q=t).poll());
    }

    public double findMedian() {
        return (Q.peek() + z.peek()) / 2.;
    }
};
```

Or:

```
class MedianFinder {

    Queue[] q = {new PriorityQueue(), new PriorityQueue(Collections.reverseOrder(
))};
    int i = 0;

    public void addNum(int num) {
        q[i].add(num);
        q[i^=1].add(q[i^1].poll());
    }

    public double findMedian() {
        return ((int)(q[1].peek()) + (int)(q[i].peek())) / 2.0;
    }
};
```

Python

```
from heapq import *

class MedianFinder:

    def __init__(self):
        self.heaps = None, [], []
        self.i = 1

    def addNum(self, num):
        heappush(self.heaps[-self.i], -heappushpop(self.heaps[self.i], num * self
.i))
        self.i *= -1

    def findMedian(self):
        return (self.heaps[self.i][0] * self.i - self.heaps[-1][0]) / 2.0
```

Or:

```
from heapq import *

class MedianFinder:

    def __init__(self):
        self.data = 1, [], []

    def addNum(self, num):
        sign, h1, h2 = self.data
        heappush(h2, -heappushpop(h1, num * sign))
        self.data = -sign, h2, h1

    def findMedian(self):
        sign, h1, h2 = d = self.data
        return (h1[0] * sign - d[-sign][0]) / 2.0
```

written by StefanPochmann original link here

Best Meeting Point(296)

Answer 1

```
public int minTotalDistance(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;

    List<Integer> I = new ArrayList<>(m);
    List<Integer> J = new ArrayList<>(n);

    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(grid[i][j] == 1){
                I.add(i);
                J.add(j);
            }
        }
    }

    return getMin(I) + getMin(J);
}

private int getMin(List<Integer> list){
    int ret = 0;

    Collections.sort(list);

    int i = 0;
    int j = list.size() - 1;
    while(i < j){
        ret += list.get(j--) - list.get(i++);
    }

    return ret;
}
```

written by larrywang2014 original link here

Answer 2

Two O(mn) solutions, both take 2ms.

The neat `total += Z[hi--] - Z[lo++]` -style summing is from larrywang2014's solution.
Originally I used `total += abs(Z[i] - median)` -style.

Solution 1

No need to sort the coordinates if we **collect** them in sorted order.

```
public int minTotalDistance(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    int total = 0, Z[] = new int[m*n];
    for (int dim=0; dim<2; ++dim) {
        int i = 0, j = 0;
        if (dim == 0) {
            for (int x=0; x<n; ++x)
                for (int y=0; y<m; ++y)
                    if (grid[y][x] == 1)
                        Z[j++] = x;
        } else {
            for (int y=0; y<m; ++y)
                for (int g : grid[y])
                    if (g == 1)
                        Z[j++] = y;
        }
        while (i < --j)
            total += Z[j] - Z[i++];
    }
    return total;
}
```

Solution 2

BucketSort-ish. Count how many people live in each row and each column. Only O(m+n) space.

```
public int minTotalDistance(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    int[] I = new int[m], J = new int[n];
    for (int i=0; i<m; ++i)
        for (int j=0; j<n; ++j)
            if (grid[i][j] == 1) {
                ++I[i];
                ++J[j];
            }
    int total = 0;
    for (int[] K : new int[][]{ I, J }) {
        int i = 0, j = K.length - 1;
        while (i < j) {
            int k = Math.min(K[i], K[j]);
            total += k * (j - i);
            if ((K[i] -= k) == 0) ++i;
            if ((K[j] -= k) == 0) --j;
        }
    }
    return total;
}
```

Not sure Larry's way is actually better here. I'll have to try the other style as well...

written by StefanPochmann original link here

Answer 3

When I first saw this question, intuitively I know shortest meeting point should be found in two separate dimension, however, even if on 1-D, how could I find the shortest meeting point? Then I clicked discuss and found out everybody's solution was using median to get shortest meeting point? WHY?

Actually, there is a famous conclusion in statistics that the median minimizes the sum of absolute deviations.

written by TonyLic original link here

Serialize and Deserialize Binary Tree(297)

Answer 1

The idea is simple: print the tree in pre-order traversal and use "X" to denote null node and split node with ",". We can use a StringBuilder for building the string on the fly. For deserializing, we use a Queue to store the pre-order traversal and since we have "X" as null node, we know exactly how to where to end building subtress.

```
public class Codec {
    private static final String spliter = ",";
    private static final String NN = "X";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NN).append(spliter);
        } else {
            sb.append(node.val).append(spliter);
            buildString(node.left, sb);
            buildString(node.right,sb);
        }
    }
    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Deque<String> nodes = new LinkedList<>();
        nodes.addAll(Arrays.asList(data.split(spliter)));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Deque<String> nodes) {
        String val = nodes.remove();
        if (val.equals(NN)) return null;
        else {
            TreeNode node = new TreeNode(Integer.valueOf(val));
            node.left = buildTree(nodes);
            node.right = buildTree(nodes);
            return node;
        }
    }
}
```

written by gavinlinasd original link here

Answer 2

Python

```python
class Codec:

    def serialize(self, root):
        def doit(node):
            if node:
                vals.append(str(node.val))
                doit(node.left)
                doit(node.right)
            else:
                vals.append('#')
        vals = []
        doit(root)
        return ' '.join(vals)

    def deserialize(self, data):
        def doit():
            val = next(vals)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = doit()
            node.right = doit()
            return node
        vals = iter(data.split())
        return doit()
```

C++

```
class Codec {
public:

    string serialize(TreeNode* root) {
        ostringstream out;
        serialize(root, out);
        return out.str();
    }

    TreeNode* deserialize(string data) {
        istringstream in(data);
        return deserialize(in);
    }

private:

    void serialize(TreeNode* root, ostringstream& out) {
        if (root) {
            out << root->val << ' ';
            serialize(root->left, out);
            serialize(root->right, out);
        } else {
            out << "# ";
        }
    }

    TreeNode* deserialize(istringstream& in) {
        string val;
        in >> val;
        if (val == "#")
            return nullptr;
        TreeNode* root = new TreeNode(stoi(val));
        root->left = deserialize(in);
        root->right = deserialize(in);
        return root;
    }
};
```

written by StefanPochmann original link here

Answer 3

```
 public String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    helperS(root, sb);
    return sb.toString();
}

private void helperS(TreeNode node, StringBuilder sb){
    if(node == null){
        sb.append("null").append(",");
        return;
    }

    sb.append(node.val).append(",");

    helperS(node.left, sb);
    helperS(node.right, sb);
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    String[] vals  = data.split("[,]");
    int[]    index = new int[]{0};
    return helperD(vals, index);
}

private TreeNode helperD(String[] vals, int[] index){
    if(index[0] == vals.length){
        return null;
    }

    String visiting = vals[index[0]++];
    if(visiting.equals("null")){
        return null;
    }

    TreeNode node = new TreeNode(Integer.valueOf(visiting));
    node.left     = helperD(vals, index);
    node.right    = helperD(vals, index);

    return node;
}
```

written by larrywang2014 original link here

Binary Tree Longest Consecutive Sequence(298)

Answer 1

Just very intuitive depth-first search, send cur node value to the next level and compare it with the next level node.

```
public class Solution {
    private int max = 0;
    public int longestConsecutive(TreeNode root) {
        if(root == null) return 0;
        helper(root, 0, root.val);
        return max;
    }

    public void helper(TreeNode root, int cur, int target){
        if(root == null) return;
        if(root.val == target) cur++;
        else cur = 1;
        max = Math.max(cur, max);
        helper(root.left, cur, root.val + 1);
        helper(root.right, cur, root.val + 1);
    }
}
```

written by czonzhu original link here

Answer 2

```
class Solution {
public:
    int longestConsecutive(TreeNode* root) {
        return search(root, nullptr, 0);
    }

    int search(TreeNode *root, TreeNode *parent, int len) {
        if (!root) return len;
        len = (parent && root->val == parent->val + 1)?len+1:1;
        return max(len, max(search(root->left, root, len), search(root->right, ro
ot, len)));
    }
};
```

len stores the longest path till current node.

written by lightmark original link here

Answer 3

```java
public class Solution {
    public int longestConsecutive(TreeNode root) {
        return (root==null)?0:Math.max(dfs(root.left, 1, root.val), dfs(root.righ
t, 1, root.val));
    }

    public int dfs(TreeNode root, int count, int val){
        if(root==null) return count;
        count = (root.val - val == 1)?count+1:1;
        int left = dfs(root.left, count, root.val);
        int right = dfs(root.right, count, root.val);
        return Math.max(Math.max(left, right), count);
    }
}
```

written by nightowl original link here

Bulls and Cows(299)

Answer 1

The idea is to iterate over the numbers in `secret` and in `guess` and count all bulls right away. For cows maintain an array that stores count of the number appearances in `secret` and in `guess`. Increment cows when either number from `secret` was already seen in `guest` or vice versa.

```
public String getHint(String secret, String guess) {
    int bulls = 0;
    int cows = 0;
    int[] numbers = new int[10];
    for (int i = 0; i<secret.length(); i++) {
        int s = Character.getNumericValue(secret.charAt(i));
        int g = Character.getNumericValue(guess.charAt(i));
        if (s == g) bulls++;
        else {
            if (numbers[s] < 0) cows++;
            if (numbers[g] > 0) cows++;
            numbers[s] ++;
            numbers[g] --;
        }
    }
    return bulls + "A" + cows + "B";
}
```

A slightly more concise version:

```
public String getHint(String secret, String guess) {
    int bulls = 0;
    int cows = 0;
    int[] numbers = new int[10];
    for (int i = 0; i<secret.length(); i++) {
        if (secret.charAt(i) == guess.charAt(i)) bulls++;
        else {
            if (numbers[secret.charAt(i)-'0']++ < 0) cows++;
            if (numbers[guess.charAt(i)-'0']-- > 0) cows++;
        }
    }
    return bulls + "A" + cows + "B";
}
```

written by ruben3 original link here

Answer 2

```java
public class Solution {
    public String getHint(String secret, String guess) {
        int bull = 0, cow = 0;

        int[] sarr = new int[10];
        int[] garr = new int[10];

        for(int i = 0; i < secret.length(); i++){
            if(secret.charAt(i) != guess.charAt(i)){
                sarr[secret.charAt(i)-'0']++;
                garr[guess.charAt(i)-'0']++;
            }else{
                bull++;
            }
        }

        for(int i = 0; i <= 9; i++){
            cow += Math.min(sarr[i], garr[i]);
        }

        return (bull + "A" + cow + "B");
    }
}
```

written by harish-v original link here

Answer 3

use `Counter` to count `guess` and `secret` and sum their overlap. Then use `zip` to count `A`.

```python
    s, g = Counter(secret), Counter(guess)
    a = sum(i == j for i, j in zip(secret, guess))
    return '%sA%sB' % (a, sum((s & g).values()) - a)
```

written by xcv58 original link here

Longest Increasing Subsequence(300)

Answer 1

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 0;

        for(int x : nums) {
            int i = Arrays.binarySearch(dp, 0, len, x);
            if(i < 0) i = -(i + 1);
            dp[i] = x;
            if(i == len) len++;
        }

        return len;
    }
}
```

written by jopiko123 original link here

Answer 2

Inspired by http://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> res;
    for(int i=0; i<nums.size(); i++) {
        auto it = std::lower_bound(res.begin(), res.end(), nums[i]);
        if(it==res.end()) res.push_back(nums[i]);
        else *it = nums[i];
    }
    return res.size();
}
```

written by dtccwl original link here

Answer 3

This is a classic problem and here is a DP solution for reference Please note a NLogN solution can be found in the following link Geek for Geek

```cpp
class Solution {
public:
    // There's a typical DP solution with O(N^2) Time and O(N) space
    // DP[i] means the result ends at i
    // So for dp[i], dp[i] is max(dp[j]+1), for all j < i and nums[j] < nums[i]
    int lengthOfLIS(vector<int>& nums) {
        const int size = nums.size();
        if (size == 0) { return 0; }
        vector<int> dp(size, 1);
        int res = 1;
        for (int i = 1; i < size; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j]+1);
                }
            }
            res = max (res, dp[i]);
        }
        return res;
    }
};
```

written by moumoutsay original link here

Remove Invalid Parentheses(301)

Answer 1

The idea is straightforward, with the input string `s`, we generate all possible states by removing one `(` or `)`, check if they are valid, if found valid ones on the current level, put them to the final result list and we are done, otherwise, add them to a queue and carry on to the next level.

The good thing of using BFS is that we can guarantee the number of parentheses that need to be removed is minimal, also no recursion call is needed in BFS.

Thanks to @peisi, we don't need stack to check valid parentheses.

Time complexity:

In BFS we handle the states level by level, in the worst case, we need to handle all the levels, we can analyze the time complexity level by level and add them up to get the final complexity.

On the first level, there's only one string which is the input string `s`, let's say the length of it is `n`, to check whether it's valid, we need `O(n)` time. On the second level, we remove one `(` or `)` from the first level, so there are `C(n, n-1)` new strings, each of them has `n-1` characters, and for each string, we need to check whether it's valid or not, thus the total time complexity on this level is `(n-1)` x `C(n, n-1)`. Come to the third level, total time complexity is `(n-2)` x `C(n, n-2)`, so on and so forth...

Finally we have this formula:

`T(n)` = `n` x `C(n, n)` + `(n-1)` x `C(n, n-1)` + ... + `1` x `C(n, 1)` = `n` x `2^(n-1)`.

Following is the Java solution:

```
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
      List<String> res = new ArrayList<>();

      // sanity check
      if (s == null) return res;

      Set<String> visited = new HashSet<>();
      Queue<String> queue = new LinkedList<>();

      // initialize
      queue.add(s);
      visited.add(s);

      boolean found = false;

      while (!queue.isEmpty()) {
        s = queue.poll();

        if (isValid(s)) {
          // found an answer, add to the result
```

```
          res.add(s);
          found = true;
        }

        if (found) continue;

        // generate all possible states
        for (int i = 0; i < s.length(); i++) {
          // we only try to remove left or right paren
          if (s.charAt(i) != '(' && s.charAt(i) != ')') continue;

          String t = s.substring(0, i) + s.substring(i + 1);

          if (!visited.contains(t)) {
            // for each state, if it's not visited, add it to the queue
            queue.add(t);
            visited.add(t);
          }
        }
      }

      return res;
    }

    // helper function checks if string s contains valid parantheses
    boolean isValid(String s) {
      int count = 0;

      for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(') count++;
        if (c == ')' && count-- == 0) return false;
      }

      return count == 0;
    }
}
```

written by jeantimex original link here

Answer 2

```
class Solution {
public:
    vector<string> removeInvalidParentheses(string s) {
        unordered_set<string> result;
        int left_removed = 0;
        int right_removed = 0;
        for(auto c : s) {
            if(c == '(') {
                ++left_removed;
            }
            if(c == ')') {
                if(left_removed != 0) {
                    --left_removed;
                }
```

```
                }
                else {
                    ++right_removed;
                }
            }
        }
        helper(s, 0, left_removed, right_removed, 0, "", result);
        return vector<string>(result.begin(), result.end());
    }
private:
    void helper(string s, int index, int left_removed, int right_removed, int pair
, string path, unordered_set<string>& result) {
        if(index == s.size()) {
            if(left_removed == 0 && right_removed == 0 && pair == 0) {
                result.insert(path);
            }
            return;
        }
        if(s[index] != '(' && s[index] != ')') {
            helper(s, index + 1, left_removed, right_removed, pair, path + s[inde
x], result);
        }
        else {
            if(s[index] == '(') {
                if(left_removed > 0) {
                    helper(s, index + 1, left_removed - 1, right_removed, pair, p
ath, result);
                }
                helper(s, index + 1, left_removed, right_removed, pair + 1, path
+ s[index], result);
            }
            if(s[index] == ')') {
                if(right_removed > 0) {
                    helper(s, index + 1, left_removed, right_removed - 1, pair, p
ath, result);
                }
                if(pair > 0) {
                    helper(s, index + 1, left_removed, right_removed, pair - 1, p
ath + s[index], result);
                }
            }
        }
    }
};
```

written by ethan_noc original link here

Answer 3

DFS solution with optimizations:

1. Before starting DFS, calculate the total numbers of opening and closing parentheses that need to be removed in the final solution, then these two numbers could be used to speed up the DFS process.
2. Use while loop to avoid duplicate result in DFS, instead of using HashSet.
3. Use count variable to validate the parentheses dynamically.

```java
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
        int count = 0, openN = 0, closeN = 0;

        // calculate the total numbers of opening and closing parentheses
        // that need to be removed in the final solution
        for (char c : s.toCharArray()) {
            if (c == '(') {
                count++;
            } else if (c == ')') {
                if (count == 0) closeN++;
                else count--;
            }
        }
        openN = count;
        count = 0;

        if (openN == 0 && closeN == 0) return Arrays.asList(s);

        List<String> result = new ArrayList<>();
        StringBuilder sb = new StringBuilder();

        dfs(s.toCharArray(), 0, count, openN, closeN, result, sb);

        return result;
    }

    private void dfs(char[] s, int p, int count, int openN, int closeN, List<Stri
ng> result, StringBuilder sb) {
        if (count < 0) return; // the parentheses is invalid

        if (p == s.length) {
            if (openN == 0 && closeN == 0) { // the minimum number of invalid par
entheses have been removed
                result.add(sb.toString());
            }
            return;
        }

        if (s[p] != '(' && s[p] != ')') {
            sb.append(s[p]);
            dfs(s, p + 1, count, openN, closeN, result, sb);
            sb.deleteCharAt(sb.length() - 1);
        } else if (s[p] == '(') {
            int i = 1;
            while (p + i < s.length && s[p + i] == '(') i++; // use while loop to
avoid duplicate result in DFS, instead of using HashSet
            sb.append(s, p, i);
            dfs(s, p + i, count + i, openN, closeN, result, sb);
            sb.delete(sb.length() - i, sb.length());

            if (openN > 0) {
                // remove the current opening parenthesis
                dfs(s, p + 1, count, openN - 1, closeN, result, sb);
            }
```

```
        } else {
            int i = 1;
            while (p + i < s.length && s[p + i] == ')') i++; // use while loop to
avoid duplicate result in DFS, instead of using HashSet
            sb.append(s, p, i);
            dfs(s, p + i, count - i, openN, closeN, result, sb);
            sb.delete(sb.length() - i, sb.length());

            if (closeN > 0) {
                // remove the current closing parenthesis
                dfs(s, p + 1, count, openN, closeN - 1, result, sb);
            }
        }
    }
}
```

written by WHJ425 original link here

Smallest Rectangle Enclosing Black Pixels(302)

Answer 1

Suppose we have a 2D array

```
"000000111000000"
"000000101000000"
"000000101100000"
"000001100100000"
```

Imagine we project the 2D array to the bottom axis with the rule "if a column has any black pixel it's projection is black otherwise white". The projected 1D array is

```
"000001111100000"
```

> **Theorem**
>
> If there are only one black pixel region, then in a projected 1D array all the black pixels are connected.
>
> **Proof by contradiction**
>
> Assume to the contrary that there are disconnected black pixels at `i` and `j` where `i < j` in the 1D projection array. Thus there exists one column `k`, `k in (i, j)` and and the column `k` in the 2D array has no black pixel. Therefore in the 2D array there exists at least 2 black pixel regions separated by column `k` which contradicting the condition of "only one black pixel region".
>
> Therefore we conclude that all the black pixels in the 1D projection array is connected.

This means we can do a binary search in each half to find the boundaries, if we know one black pixel's position. And we do know that.

To find the left boundary, do the binary search in the `[0, y)` range and find the first column vector who has any black pixel.

To determine if a column vector has a black pixel is `O(m)` so the search in total is `O(m log n)`

We can do the same for the other boundaries. The area is then calculated by the boundaries. Thus the algorithm runs in `O(m log n + n log m)`

Java

```java
private char[][] image;
public int minArea(char[][] iImage, int x, int y) {
    image = iImage;
    int m = image.length, n = image[0].length;
    int left = searchColumns(0, y, 0, m, true);
    int right = searchColumns(y + 1, n, 0, m, false);
    int top = searchRows(0, x, left, right, true);
    int bottom = searchRows(x + 1, m, left, right, false);
    return (right - left) * (bottom - top);
}
private int searchColumns(int i, int j, int top, int bottom, boolean opt) {
    while (i != j) {
        int k = top, mid = (i + j) / 2;
        while (k < bottom && image[k][mid] == '0') ++k;
        if (k < bottom == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
private int searchRows(int i, int j, int left, int right, boolean opt) {
    while (i != j) {
        int k = left, mid = (i + j) / 2;
        while (k < right && image[mid][k] == '0') ++k;
        if (k < right == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
//  Runtime: 1 ms
```

C++

```cpp
vector<vector<char>> *image;
int minArea(vector<vector<char>> &iImage, int x, int y) {
    image = &iImage;
    int m = int(image->size()), n = int((*image)[0].size());
    int top = searchRows(0, x, 0, n, true);
    int bottom = searchRows(x + 1, m, 0, n, false);
    int left = searchColumns(0, y, top, bottom, true);
    int right = searchColumns(y + 1, n, top, bottom, false);
    return (right - left) * (bottom - top);
}
int searchRows(int i, int j, int low, int high, bool opt) {
    while (i != j) {
        int k = low, mid = (i + j) / 2;
        while (k < high && (*image)[mid][k] == '0') ++k;
        if (k < high == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
int searchColumns(int i, int j, int low, int high, bool opt) {
    while (i != j) {
        int k = low, mid = (i + j) / 2;
        while (k < high && (*image)[k][mid] == '0') ++k;
        if (k < high == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
// Runtime: 20 ms
```

Python

```
def minArea(self, image, x, y):
    top = self.searchRows(image, 0, x, True)
    bottom = self.searchRows(image, x + 1, len(image), False)
    left = self.searchColumns(image, 0, y, top, bottom, True)
    right = self.searchColumns(image, y + 1, len(image[0]), top, bottom, False)
    return (right - left) * (bottom - top)

def searchRows(self, image, i, j, opt):
    while i != j:
        m = (i + j) / 2
        if ('1' in image[m]) == opt:
            j = m
        else:
            i = m + 1
    return i

def searchColumns(self, image, i, j, top, bottom, opt):
    while i != j:
        m = (i + j) / 2
        if any(image[k][m] == '1' for k in xrange(top, bottom)) == opt:
            j = m
        else:
            i = m + 1
    return i
# Runtime: 56 ms
```

Java (DRY)

```
private char[][] image;
public int minArea(char[][] iImage, int x, int y) {
    image = iImage;
    int m = image.length, n = image[0].length;
    int top = search(0, x, 0, n, true, true);
    int bottom = search(x + 1, m, 0, n, false, true);
    int left = search(0, y, top, bottom, true, false);
    int right = search(y + 1, n, top, bottom, false, false);
    return (right - left) * (bottom - top);
}
private boolean isWhite(int mid, int k, boolean isRow) {
    return ((isRow) ? image[mid][k] : image[k][mid]) == '0';
}
private int search(int i, int j, int low, int high, boolean opt, boolean isRow) {
    while (i != j) {
        int k = low, mid = (i + j) / 2;
        while (k < high && isWhite(mid, k, isRow)) ++k;
        if (k < high == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
//  Runtime: 2 ms
```

C++ (DRY)

```
vector<vector<char>> *image;
int minArea(vector<vector<char>> &iImage, int x, int y) {
    image = &iImage;
    int m = int(image->size()), n = int((*image)[0].size());
    int top = search(0, x, 0, n, true, true);
    int bottom = search(x + 1, m, 0, n, false, true);
    int left = search(0, y, top, bottom, true, false);
    int right = search(y + 1, n, top, bottom, false, false);
    return (right - left) * (bottom - top);
}
bool isWhite(int mid, int k, bool isRow) {
    return ((isRow) ? (*image)[mid][k]:(*image)[k][mid]) == '0';
}
int search(int i, int j, int low, int high, bool opt, bool isRow) {
    while (i != j) {
        int k = low, mid = (i + j) / 2;
        while (k < high && isWhite(mid, k, isRow)) ++k;
        if (k < high == opt)
            j = mid;
        else
            i = mid + 1;
    }
    return i;
}
// Runtime: 24 ms
```

Python (DRY, from Stefan's cool solution)

```
def minArea(self, image, x, y):
    top = self.search(0, x, lambda mid: '1' in image[mid])
    bottom = self.search(x + 1, len(image), lambda mid: '1' not in image[mid])
    left = self.search(0, y, lambda mid: any(image[k][mid] == '1' for k in xrange
(top, bottom)))
    right = self.search(y + 1, len(image[0]), lambda mid: all(image[k][mid] == '0
' for k in xrange(top, bottom)))
    return (right - left) * (bottom - top)

def search(self, i, j, check):
    while i != j:
        mid = (i + j) / 2
        if check(mid):
            j = mid
        else:
            i = mid + 1
    return i
# Runtime: 56 ms
```

written by dietpepsi original link here

Answer 2

```
def minArea(self, image, x, y):
    def first(lo, hi, check):
        while lo < hi:
            mid = (lo + hi) / 2
            if check(mid):
                hi = mid
            else:
                lo = mid + 1
        return lo
    top    = first(0, x,             lambda x: '1' in image[x])
    bottom = first(x, len(image),    lambda x: '1' not in image[x])
    left   = first(0, y,             lambda y: any(row[y] == '1' for row in image
))
    right  = first(y, len(image[0]), lambda y: all(row[y] == '0' for row in image
))
    return (bottom - top) * (right - left)
```

written by StefanPochmann original link here

Answer 3

DFS or BFS is the intuitive solution for this problem while the problem is with a tag "binary search". So can anyone provide a binary search answer. DFS complexity is O(m * n) and if binary search it would be O(n * lgm + m * lgn)

```
public class Solution {
    private int minX = Integer.MAX_VALUE, minY = Integer.MAX_VALUE, maxX = 0, max
Y = 0;
    public int minArea(char[][] image, int x, int y) {
        if(image == null || image.length == 0 || image[0].length == 0) return 0;
        dfs(image, x, y);
        return(maxX - minX + 1) * (maxY - minY + 1);
    }
    private void dfs(char[][] image, int x, int y){
        int m = image.length, n = image[0].length;
        if(x < 0 || y < 0 || x >= m || y >= n || image[x][y] == '0') return;
        image[x][y] = '0';
        minX = Math.min(minX, x);
        maxX = Math.max(maxX, x);
        minY = Math.min(minY, y);
        maxY = Math.max(maxY, y);
        dfs(image, x + 1, y);
        dfs(image, x - 1, y);
        dfs(image, x, y - 1);
        dfs(image, x, y + 1);
    }

}
```

written by czonzhu original link here

Range Sum Query - Immutable(303)

Answer 1

public class NumArray {

```
int[] nums;

public NumArray(int[] nums) {
    for(int i = 1; i < nums.length; i++)
        nums[i] += nums[i - 1];

    this.nums = nums;
}

public int sumRange(int i, int j) {
    if(i == 0)
        return nums[j];

    return nums[j] - nums[i - 1];
}
```

}

written by arthur13 original link here

Answer 2

The idea is fairly straightforward: create an array `accu` that stores the accumulated sum for `nums` such that `accu[i] = nums[0] + ... + nums[i - 1]` in the initializer of `NumArray`. Then just return `accu[j + 1] - accu[i]` in `sumRange`. You may try the example in the problem statement to convince yourself of this idea.

The code is as follows.

C++

```
class NumArray {
public:
    NumArray(vector<int> &nums) {
        accu.push_back(0);
        for (int num : nums)
            accu.push_back(accu.back() + num);
    }

    int sumRange(int i, int j) {
        return accu[j + 1] - accu[i];
    }
private:
    vector<int> accu;
};


// Your NumArray object will be instantiated and called as such:
// NumArray numArray(nums);
// numArray.sumRange(0, 1);
// numArray.sumRange(1, 2);
```

Python

```
class NumArray(object):
    def __init__(self, nums):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        self.accu = [0]
        for num in nums:
            self.accu += self.accu[-1] + num,

    def sumRange(self, i, j):
        """
        sum of elements nums[i..j], inclusive.
        :type i: int
        :type j: int
        :rtype: int
        """
        return self.accu[j + 1] - self.accu[i]


# Your NumArray object will be instantiated and called as such:
# numArray = NumArray(nums)
# numArray.sumRange(0, 1)
# numArray.sumRange(1, 2)
```

written by jianchao.li.fighter original link here

Answer 3

```
class NumArray {
public:
    NumArray(vector<int> &nums) : psum(nums.size()+1, 0) {
        partial_sum( nums.begin(), nums.end(), psum.begin()+1);
    }

    int sumRange(int i, int j) {
        return psum[j+1] - psum[i];
    }
private:
    vector<int> psum;
};
```

written by rantos22 original link here

Range Sum Query 2D - Immutable(304)

Answer 1

```
private int[][] dp;

public NumMatrix(int[][] matrix) {
    if(   matrix           == null
       || matrix.length    == 0
       || matrix[0].length == 0   ){
        return;
    }

    int m = matrix.length;
    int n = matrix[0].length;

    dp = new int[m + 1][n + 1];
    for(int i = 1; i <= m; i++){
        for(int j = 1; j <= n; j++){
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1] -dp[i - 1][j - 1] + matrix[i -
1][j - 1] ;
        }
    }
}

public int sumRegion(int row1, int col1, int row2, int col2) {
    int iMin = Math.min(row1, row2);
    int iMax = Math.max(row1, row2);

    int jMin = Math.min(col1, col2);
    int jMax = Math.max(col1, col2);

    return dp[iMax + 1][jMax + 1] - dp[iMax + 1][jMin] - dp[iMin][jMax + 1] + dp[
iMin][jMin];
}
```

written by larrywang2014 original link here

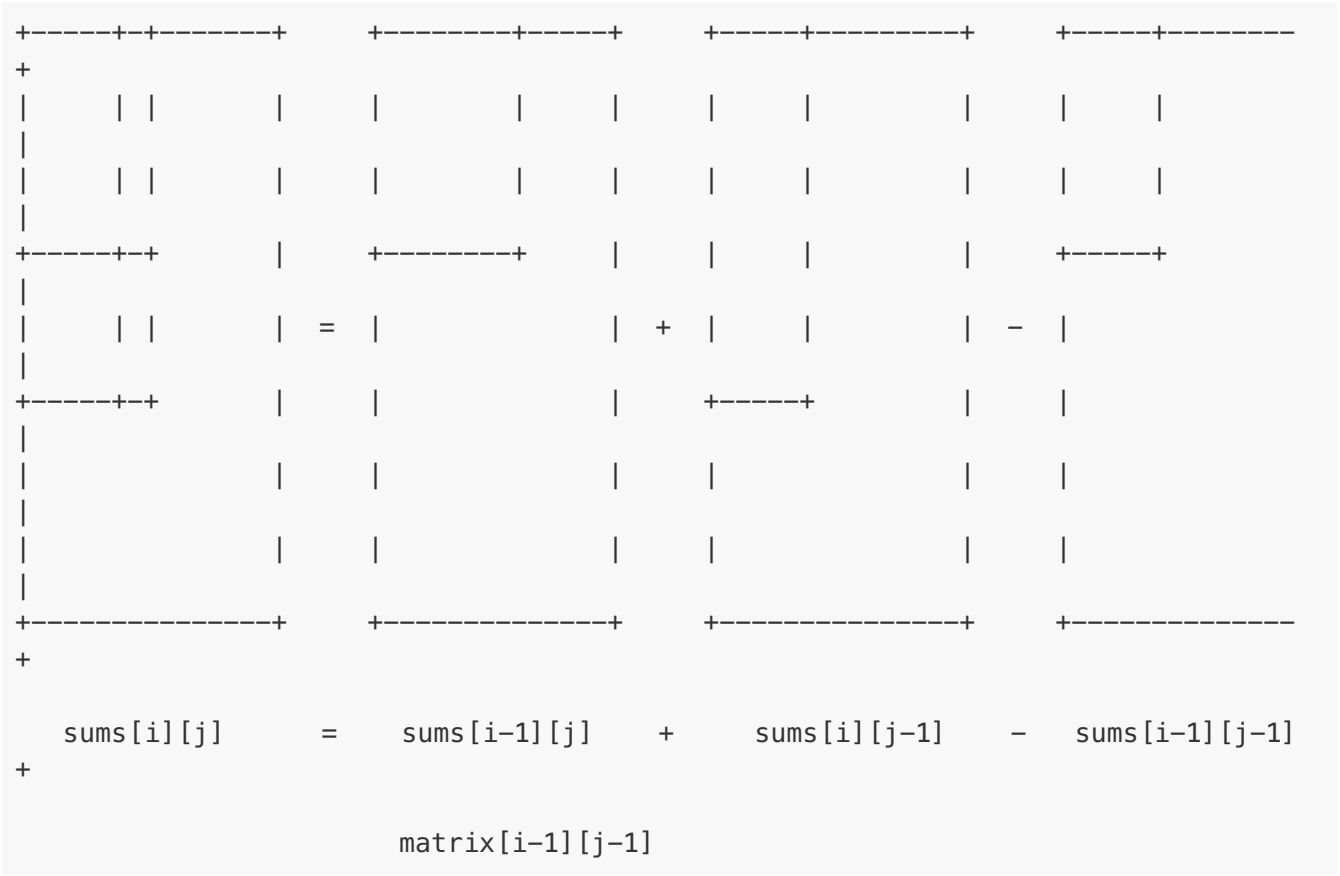
Answer 2

Construct a 2D array `sums[row+1][col+1]`

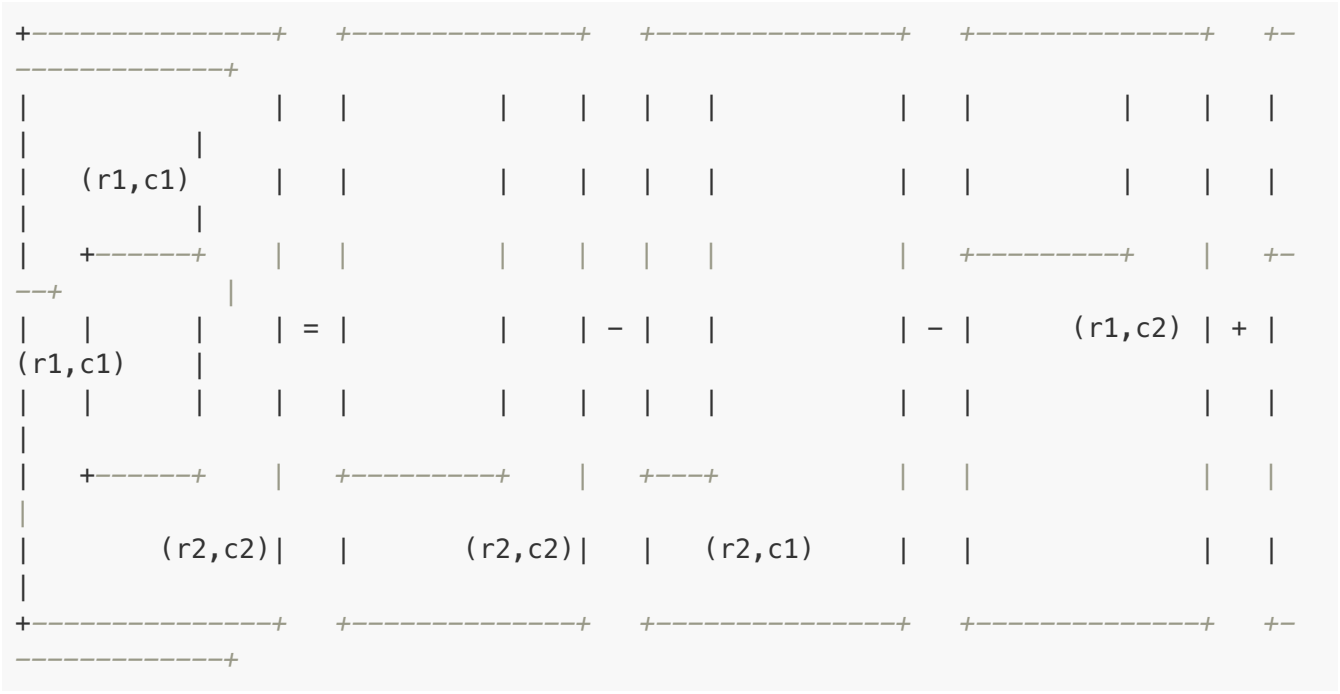
(**notice**: we add additional blank row `sums[0][col+1]={0}` and blank column `sums[row+1][0]={0}` to remove the edge case checking), so, we can have the following definition

`sums[i+1][j+1]` represents the sum of area from `matrix[0][0]` to `matrix[i][j]`

To calculate sums, the ideas as below



So, we use the same idea to find the specific area's sum.



And we can have the following code

```
class NumMatrix {
private:
    int row, col;
    vector<vector<int>> sums;
public:
    NumMatrix(vector<vector<int>> &matrix) {
        row = matrix.size();
        col = row>0 ? matrix[0].size() : 0;
        sums = vector<vector<int>>(row+1, vector<int>(col+1, 0));
        for(int i=1; i<=row; i++) {
            for(int j=1; j<=col; j++) {
                sums[i][j] = matrix[i-1][j-1] +
                             sums[i-1][j] + sums[i][j-1] - sums[i-1][j-1] ;
            }
        }
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return sums[row2+1][col2+1] - sums[row2+1][col1] - sums[row1][col2+1] + s
ums[row1][col1];
    }
};
```

written by haoel original link here

Answer 3

My `accu[i][j]` is the sum of `matrix[0..i][0..j]`, and `a(i, j)` helps with edge cases.

```
class NumMatrix {
public:
    NumMatrix(vector<vector<int>> &matrix) {
        accu = matrix;
        for (int i=0; i<matrix.size(); ++i)
            for (int j=0; j<matrix[0].size(); ++j)
                accu[i][j] += a(i-1, j) + a(i, j-1) - a(i-1, j-1);
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return a(row2, col2) - a(row1-1, col2) - a(row2, col1-1) + a(row1-1, col1
-1);
    }

private:
    vector<vector<int>> accu;
    int a(int i, int j) {
        return i >= 0 && j >= 0 ? accu[i][j] : 0;
    }
};
```

Afterthought

Instead of

```
accu[i][j] += a(i-1, j) + a(i, j-1) - a(i-1, j-1);
```

I could use

```
accu[i][j] += a(i, j) - sumRegion(i, j, i, j);
```

which is shorter but I think less clear. I do like already using `sumRegion` in the precomputation, though.

written by StefanPochmann original link here

Number of Islands II(305)

Answer 1

Union Find is an abstract data structure supporting `find` and `unite` on disjointed sets of objects, typically used to solve the network connectivity problem.

The two operations are defined like this:

`find(a,b)` : are `a` and `b` belong to the same set?

`unite(a,b)` : if `a` and `b` are not in the same set, unite the sets they belong to.

With this data structure, it is very fast for solving our problem. Every position is an new land, if the new land connect two islands `a` and `b`, we combine them to form a whole. The answer is then the number of the disjointed sets.

The following algorithm is derived from Princeton's lecture note on Union Find in Algorithms and Data Structures It is a well organized note with clear illustration describing from the naive QuickFind to the one with Weighting and Path compression. With Weighting and Path compression, The algorithm runs in `O((M+N) log* N)` where `M` is the number of operations (unite and find), `N` is the number of objects, `log*` is iterated logarithm while the naive runs in `O(MN)`.

For our problem, If there are `N` positions, then there are `O(N)` operations and `N` objects then total is `O(N log*N)`, when we don't consider the `O(mn)` for array initialization.

Note that `log*N` is almost constant (for `N` = 265536, `log*N` = 5) in this universe, so the algorithm is almost linear with `N`.

However, if the map is very big, then the initialization of the arrays can cost a lot of time when `mn` is much larger than `N`. In this case we should consider using a hashmap/dictionary for the underlying data structure to avoid this overhead.

Of course, we can put all the functionality into the Solution class which will make the code a lot shorter. But from a design point of view a separate class dedicated to the data sturcture is more readable and reusable.

I implemented the idea with 2D interface to better fit the problem.

Java

```
public class Solution {

    private int[][] dir = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}};

    public List<Integer> numIslands2(int m, int n, int[][] positions) {
        UnionFind2D islands = new UnionFind2D(m, n);
        List<Integer> ans = new ArrayList<>();
        for (int[] position : positions) {
            int x = position[0], y = position[1];
            int p = islands.add(x, y);
            for (int[] d : dir) {
                int q = islands.getID(x + d[0], y + d[1]);
```

```
                if (q > 0 && !islands.find(p, q))
                    islands.unite(p, q);
            }
            ans.add(islands.size());
        }
        return ans;
    }
}

class UnionFind2D {
    private int[] id;
    private int[] sz;
    private int m, n, count;

    public UnionFind2D(int m, int n) {
        this.count = 0;
        this.n = n;
        this.m = m;
        this.id = new int[m * n + 1];
        this.sz = new int[m * n + 1];
    }

    public int index(int x, int y) { return x * n + y + 1; }

    public int size() { return this.count; }

    public int getID(int x, int y) {
        if (0 <= x && x < m && 0<= y && y < n)
            return id[index(x, y)];
        return 0;
    }

    public int add(int x, int y) {
        int i = index(x, y);
        id[i] = i; sz[i] = 1;
        ++count;
        return i;
    }

    public boolean find(int p, int q) {
        return root(p) == root(q);
    }

    public void unite(int p, int q) {
        int i = root(p), j = root(q);
        if (sz[i] < sz[j]) { //weighted quick union
            id[i] = j; sz[j] += sz[i];
        } else {
            id[j] = i; sz[i] += sz[j];
        }
        --count;
    }

    private int root(int i) {
        for (;i != id[i]; i = id[i])
            id[i] = id[id[i]]; //path compression
        return i;
```

```
        return i;
    }
}
//Runtime: 20 ms
```

Python (using dict)

```
class Solution(object):
    def numIslands2(self, m, n, positions):
        ans = []
        islands = Union()
        for p in map(tuple, positions):
            islands.add(p)
            for dp in (0, 1), (0, -1), (1, 0), (-1, 0):
                q = (p[0] + dp[0], p[1] + dp[1])
                if q in islands.id:
                    islands.unite(p, q)
            ans += [islands.count]
        return ans

class Union(object):
    def __init__(self):
        self.id = {}
        self.sz = {}
        self.count = 0

    def add(self, p):
        self.id[p] = p
        self.sz[p] = 1
        self.count += 1

    def root(self, i):
        while i != self.id[i]:
            self.id[i] = self.id[self.id[i]]
            i = self.id[i]
        return i

    def unite(self, p, q):
        i, j = self.root(p), self.root(q)
        if i == j:
            return
        if self.sz[i] > self.sz[j]:
            i, j = j, i
        self.id[i] = j
        self.sz[j] += self.sz[i]
        self.count -= 1

#Runtime: 300 ms
```

written by dietpepsi original link here

Answer 2

This is a basic `union-find` problem. Given a graph with points being added, we can at least solve:

1. How many islands in total?
2. Which island is pointA belong to?
3. Are pointA and pointB connected?

The idea is simple. To represent a list of islands, we use **trees**. i.e., a list of roots. This helps us find the identifier of an island faster. If `roots[c] = p` means the parent of node c is p, we can climb up the parent chain to find out the identifier of an island, i.e., which island this point belongs to:

```
Do root[root[roots[c]]]... until root[c] == c;
```

To transform the two dimension problem into the classic UF, perform a linear mapping:

```
int id = n * x + y;
```

Initially assume every cell are in non-island set `{-1}`. When point A is added, we create a new root, i.e., a new island. Then, check if any of its 4 neighbors belong to the same island. If not, `union` the neighbor by setting the root to be the same. Remember to skip non-island cells.

UNION operation is only changing the root parent so the running time is `O(1)`.

FIND operation is proportional to the depth of the tree. If N is the number of points added, the average running time is `O(logN)`, and a sequence of `4N` operations take `O(NlogN)`. If there is no balancing, the worse case could be `O(N^2)`.

Here I've attached my 15ms solution. There can be at least two improvements: `union by rank` & `pass compression`. However I suggest first finish the basis, then discuss the improvements with interviewer.

Cheers!

```java
public class Solution {
    int[][] dirs = {{0, 1}, {1, 0}, {-1, 0}, {0, -1}};

    public List<Integer> numIslands2(int m, int n, int[][] positions) {
        List<Integer> result = new ArrayList<Integer>();
        if(m <= 0 || n <= 0) return result;

        int count = 0;                      // number of islands
        int[] roots = new int[m * n];       // one island = one tree
        Arrays.fill(roots, -1);

        for(int[] p : positions) {
            int root = n * p[0] + p[1];
            roots[root] = root;             // add new island
            count++;

            for(int[] dir : dirs) {
                int x = p[0] + dir[0];
                int y = p[1] + dir[1];
                int idNb = n * x + y;
                if(x < 0 || x >= m || y < 0 || y >= n || roots[idNb] == -1) conti
nue;

                int rootNb = findIsland(roots, idNb);
                if(root != rootNb) {        // if neighbor is in another island
                    roots[rootNb] = root;   // union two islands
                    count--;
                }
            }

            result.add(count);
        }
        return result;
    }

    public int findIsland(int[] roots, int id) {
        while(id != roots[id]) id = roots[id];
        return id;
    }
}
```

written by yavinci original link here

Answer 3

The basic idea is the Union-Find approach. We assign a root number for each island, and use an array to record this number. For each input, we check its four neighbor cells. If the neighbor cell is an island, then we retrieve the root number of this island. If two neighbor cells belong to two different islands, then we union them and therefore the total number of islands will become one less.

```
public List<Integer> numIslands2(int m, int n, int[][] positions) {
    //use an array to hold root number of each island
    int[] roots = new int[m*n];
    //initialize the array with -1, so we know non negative number is a root numb
er
    Arrays.fill(roots, -1);

    int[] xOffset ={0, 0, 1, -1};
    int[] yOffset = {1, -1, 0, 0};

    List<Integer> result = new ArrayList<Integer>();

    for(int[] position : positions){
        //for each input cell, its initial root number is itself
        roots[position[0]*n + position[1]] = position[0]*n + position[1];
        //count variable is used to count the island in current matrix.
        //firstly, we assume current input is an isolated island
        int count = result.isEmpty()? 1 : result.get(result.size()-1) + 1;
        //check neighbor cells
        for(int i = 0; i < 4; i++){
            int newX = xOffset[i] + position[0];
            int newY = yOffset[i] + position[1];
            //if we found one neighbor is a part of island
            if(newX >= 0 && newX < m && newY >= 0 && newY < n && roots[newX * n +
newY] != -1){
                //get the root number of this island
                int root1 = find(newX * n + newY, roots);
                //get the root number of input island
                int root2 = roots[position[0]*n + position[1]];
                //if root1 and root2 are different, then we can connect two isola
ted island together,
                // so the num of island - 1
                if(root1 != root2) count--;
                //update root number accordingly
                roots[root1] = root2;
            }
        }
        result.add(count);
    }

    return result;
}

public int find(int target, int[] roots){
    //found root
    if(roots[target] == target) return target;
    //searching for root and update the cell accordingly
    roots[target] = find(roots[target], roots);
    //return root number
    return roots[target];
}
```

written by hpplayer original link here

Additive Number(306)

Answer 1

use a helper function to add two strings.

Choose first two number then recursively check.

Note that the length of first two numbers can't be longer than half of the initial string, so the two loops in the first function will end when *i>num.size()/2* and *j>(num.size()-i)/2*, this will actually save a lot of time.

Update the case of heading 0s e.g. "100010" should return false

```
class Solution {
public:
        bool isAdditiveNumber(string num) {
            for(int i=1; i<=num.size()/2; i++){
                for(int j=1; j<=(num.size()-i)/2; j++){
                    if(check(num.substr(0,i), num.substr(i,j), num.substr(i+j)))
return true;
                }
            }
            return false;
        }
        bool check(string num1, string num2, string num){
            if(num1.size()>1 && num1[0]=='0' || num2.size()>1 && num2[0]=='0') re
turn false;
            string sum=add(num1, num2);
            if(num==sum) return true;
            if(num.size()<=sum.size() || sum.compare(num.substr(0,sum.size()))!=0
) return false;
            else return check(num2, num.substr(0,sum.size()), num.substr(sum.size
()));
        }
        string add(string n, string m){
            string res;
            int i=n.size()-1, j=m.size()-1, carry=0;
            while(i>=0 || j>=0){
                int sum=carry+(i>=0 ? (n[i--]-'0') : 0) + (j>=0?  (m[j--]-'0') :
0);
                res.push_back(sum%10+'0');
                carry=sum/10;
            }
            if(carry) res.push_back(carry+'0');
            reverse(res.begin(), res.end());
            return res;
        }
    };
```

written by zjh08177 original link here

Answer 2

The idea is quite straight forward. Generate the first and second of the sequence, check if the rest of the string match the sum recursively. `i` and `j` are length of the first and second number. `i` should in the range of `[0, n/2]`. The length of their sum should `>= max(i,j)`

Java Recursive

```
import java.math.BigInteger;

public class Solution {
    public boolean isAdditiveNumber(String num) {
        int n = num.length();
        for (int i = 1; i <= n / 2; ++i) {
            BigInteger x1 = new BigInteger(num.substring(0, i));
            for (int j = 1; Math.max(j, i) <= n - i - j; ++j) {
                if (num.charAt(i) == '0' && j > 1) break;
                BigInteger x2 = new BigInteger(num.substring(i, i + j));
                if (isValid(x1, x2, j + i, num)) return true;
            }
        }
        return false;
    }
    private boolean isValid(BigInteger x1, BigInteger x2, int start, String num)
{
        if (start == num.length()) return true;
        x2 = x2.add(x1);
        x1 = x2.subtract(x1);
        String sum = x2.toString();
        return num.startsWith(sum, start) && isValid(x1, x2, start + sum.length()
, num);
    }
}
// Runtime: 8ms
```

Since `isValid` is a tail recursion it is very easy to turn it into a loop.

Java Iterative

```java
public class Solution {
    public boolean isAdditiveNumber(String num) {
        int n = num.length();
        for (int i = 1; i <= n / 2; ++i)
            for (int j = 1; Math.max(j, i) <= n - i - j; ++j)
                if (isValid(i, j, num)) return true;
        return false;
    }
    private boolean isValid(int i, int j, String num) {
        if (num.charAt(i) == '0' && j > 1) return false;
        String sum;
        BigInteger x1 = new BigInteger(num.substring(0, i));
        BigInteger x2 = new BigInteger(num.substring(i, i + j));
        for (int start = i + j; start != num.length(); start += sum.length()) {
            x2 = x2.add(x1);
            x1 = x2.subtract(x1);
            sum = x2.toString();
            if (!num.startsWith(sum, start)) return false;
        }
        return true;
    }
}
// Runtime: 9ms
```

If no overflow, instead of BigInteger we can consider to use Long which is a lot faster.

Java Iterative Using Long

```java
public class Solution {
    public boolean isAdditiveNumber(String num) {
        int n = num.length();
        for (int i = 1; i <= n / 2; ++i)
            for (int j = 1; Math.max(j, i) <= n - i - j; ++j)
                if (isValid(i, j, num)) return true;
        return false;
    }
    private boolean isValid(int i, int j, String num) {
        if (num.charAt(i) == '0' && j > 1) return false;
        String sum;
        Long x1 = Long.parseLong(num.substring(0, i));
        Long x2 = Long.parseLong(num.substring(i, i + j));
        for (int start = i + j; start != num.length(); start += sum.length()) {
            x2 = x2 + x1;
            x1 = x2 - x1;
            sum = x2.toString();
            if (!num.startsWith(sum, start)) return false;
        }
        return true;
    }
}
// Runtime: 3ms
```

written by dietpepsi original link here

Answer 3

Backtracking with Pruning.

Java (3 ms):

```
public class Solution {
    public boolean isAdditiveNumber(String num) {
        if (num == null || num.length() < 3) return false;
        int n = num.length();
        for (int i = 1; i < n; i++) {
            if (i > 1 && num.charAt(0) == '0') break;
            for (int j = i+1; j < n; j++) {
                int first = 0, second = i, third = j;
                if (num.charAt(second) == '0' && third > second+1) break;
                while (third < n) {
                    Long result = (Long.parseLong(num.substring(first, second)) +

                                   Long.parseLong(num.substring(second, third)) )
;
                    if (num.substring(third).startsWith(result.toString())) {
                        first = second; second = third; third += result.toString(
).length();
                    }
                    else {
                        break;
                    }
                }
                if (third == n) return true;
            }
        }
        return false;
    }
}
```

Python (48 ms):

```python
class Solution(object):
    def isAdditiveNumber(self, num):
        """
        :type num: str
        :rtype: bool
        """
        if num is None or len(num) < 3:
            return False
        n = len(num)
        for i in range(1, n):
            if i > 1 and num[0] == '0':
                break
            for j in range(i+1, n):
                first, second, third = 0, i, j
                if num[second] == '0' and third > second + 1:
                    break
                while third < n:
                    result = str(int(num[first:second]) + int(num[second:third]))
                    if num[third:].startswith(result):
                        first, second, third = second, third, third + len(result)
                    else:
                        break
                if third == n:
                    return True
        return False
```

written by stpeterh original link here

Range Sum Query - Mutable(307)

Answer 1

```
public class NumArray {

    class SegmentTreeNode {
        int start, end;
        SegmentTreeNode left, right;
        int sum;

        public SegmentTreeNode(int start, int end) {
            this.start = start;
            this.end = end;
            this.left = null;
            this.right = null;
            this.sum = 0;
        }
    }

    SegmentTreeNode root = null;

    public NumArray(int[] nums) {
        root = buildTree(nums, 0, nums.length-1);
    }

    private SegmentTreeNode buildTree(int[] nums, int start, int end) {
        if (start > end) {
            return null;
        } else {
            SegmentTreeNode ret = new SegmentTreeNode(start, end);
            if (start == end) {
                ret.sum = nums[start];
            } else {
                int mid = start  + (end - start) / 2;
                ret.left = buildTree(nums, start, mid);
                ret.right = buildTree(nums, mid + 1, end);
                ret.sum = ret.left.sum + ret.right.sum;
            }
            return ret;
        }
    }

    void update(int i, int val) {
        update(root, i, val);
    }

    void update(SegmentTreeNode root, int pos, int val) {
        if (root.start == root.end) {
           root.sum = val;
        } else {
            int mid = root.start + (root.end - root.start) / 2;
            if (pos <= mid) {
                 update(root.left, pos, val);
            } else {
                 update(root.right, pos, val);
```

```
            }
            root.sum = root.left.sum + root.right.sum;
        }
    }

    public int sumRange(int i, int j) {
        return sumRange(root, i, j);
    }

    public int sumRange(SegmentTreeNode root, int start, int end) {
        if (root.end == end && root.start == start) {
            return root.sum;
        } else {
            int mid = root.start + (root.end - root.start) / 2;
            if (end <= mid) {
                return sumRange(root.left, start, end);
            } else if (start >= mid+1) {
                return sumRange(root.right, start, end);
            }  else {
                return sumRange(root.right, mid+1, end) + sumRange(root.left, sta
rt, mid);
            }
        }
    }
}
```

written by 2guotou original link here

Answer 2

The idea is using â€œbucketsâ€. Assume the length of the input array is n, we can partition the whole array into m buckets, with each bucket having k=n/m elements. For each bucket, we record two kind of information: 1) a copy of elements in the bucket, 2) the sum of all the elements in the bucket.

For example: If the input is [0,1,2,3,4,5,6,7,8,9], and we partition it into 4 buckets, formatted as {[numbers], sum}:

- bucket0: {[0, 1, 2], 3}
- bucket1: {[3, 4, 5], 12}
- bucket2: {[6, 7, 8], 21}
- bucket3: {[9], 9};

Updating is easy. You just need to find the right bucket, modify the element value, and change the â€œsumâ€ value in that bucket accordingly. The operation takes O(1) time.

Summation is a little complicated. In the above example, letâ€™s say we want to compute the sum in range [1, 7]. We can see, the numbers we want to accumulate are in bucket0, bucket1, and bucket2. Specifically, we only need parts of numbers in bucket0 and bucket2, and all the numbers in bucket1. Because the summation of all numbers in bucket1 have already been computed, we donâ€™t need to compute it again. So, instead of doing (1+2) + (3+4+5) + (6+7), we can just do (1+2) + 12 +

(6+7). We save two â€œ+â€ operations. If you change the size of buckets, the number of saved â€œ+â€ operations will be different. The questions is:

What is the best size that can save the most â€œ+â€ operations?

Here is my analysis, which might be incorrect.

We have:

- The number of buckets is **m**.
- The size of each bucket is **k**.
- The length of input array is **n**, and we have **mk=n**.

In the worst case (the query is [0, n-1]), we will first add all the elements in bucket0, then add from bucket1 to bucket(m-2), and finally add all the elements in bucket(m-1), so we do **2k+m-2** â€œ+â€ operations. We want to find the minimum value of **2k+m**. Because **2km=2n**, when **2k=m**, **2k+m** reaches the minimum value. (Need proof?) So we have **m = sqrt(2n)** and **k=sqrt(n/2)**.

Therefore, in the worst case, the best size of bucket is **k=sqrt(n/2)**, and the complexity is **O(2k+m-2)=O(2m-2)=O(m)=O(sqrt(2n))=O(n^0.5)**;

Thank you for pointing out any mistake!

```
class NumArray {
public:

    struct Bucket
    {
        int sum;
        vector<int> val;
    };

    int bucketNum;
    int bucketSize;
    vector<Bucket> Bs;

    NumArray(vector<int> &nums) {
        int size = nums.size();
        int bucketNum = (int)sqrt(2*size);
        bucketSize = bucketNum/2;
        while(bucketSize * bucketNum<size) ++bucketSize;

        Bs.resize(bucketNum);
        for(int i=0, k=0; i<bucketNum; ++i)
        {
            int temp = 0;
            Bs[i].val.resize(bucketSize);
            for(int j=0; j<bucketSize && k<size; ++j, ++k)
            {
                temp += nums[k];
                Bs[i].val[j] = nums[k];
            }
            Bs[i].sum = temp;
        }
    }
```

```
    void update(int i, int val) {
        int x = i / bucketSize;
        int y = i % bucketSize;
        Bs[x].sum += (val - Bs[x].val[y]);
        Bs[x].val[y] = val;
    }

    int sumRange(int i, int j) {
        int x1 = i / bucketSize;
        int y1 = i % bucketSize;
        int x2 = j / bucketSize;
        int y2 = j % bucketSize;
        int sum = 0;

        if(x1==x2)
        {
            for(int a=y1; a<=y2; ++a)
            {
                sum += Bs[x1].val[a];
            }
            return sum;
        }

        for(int a=y1; a<bucketSize; ++a)
        {
            sum += Bs[x1].val[a];
        }
        for(int a=x1+1; a<x2; ++a)
        {
            sum += Bs[a].sum;
        }
        for(int b=0; b<=y2; ++b)
        {
            sum += Bs[x2].val[b];
        }
        return sum;
    }
};
```

written by TTester original link here

Answer 3

```
class NumArray(object):
    def __init__(self, nums):
        self.update = nums.__setitem__
        self.sumRange = lambda i, j: sum(nums[i:j+1])
```

I added two lines, but I also removed two lines, so zero overall, right? Just kidding :-P

Not a serious solution, just showing some Python trickery. The `sumRange` takes linear time, but due to the test suite being weak, this solution gets accepted (in about

1200-1300ms).

written by StefanPochmann original link here

Range Sum Query 2D - Mutable(308)

Answer 1

```
public class NumMatrix {

    int[][] tree;
    int[][] nums;
    int m;
    int n;

    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0 || matrix[0].length == 0) return;
        m = matrix.length;
        n = matrix[0].length;
        tree = new int[m+1][n+1];
        nums = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    public void update(int row, int col, int val) {
        if (m == 0 || n == 0) return;
        int delta = val - nums[row][col];
        nums[row][col] = val;
        for (int i = row + 1; i <= m; i += i & (-i)) {
            for (int j = col + 1; j <= n; j += j & (-j)) {
                tree[i][j] += delta;
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        if (m == 0 || n == 0) return 0;
        return sum(row2+1, col2+1) + sum(row1, col1) - sum(row1, col2+1) - sum(ro
w2+1, col1);
    }

    public int sum(int row, int col) {
        int sum = 0;
        for (int i = row; i > 0; i -= i & (-i)) {
            for (int j = col; j > 0; j -= j & (-j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }
}
// time should be O(log(m) * log(n))
```

Explanation of Binary Indexed Tree : https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

written by novice00 original link here

Answer 2

We use colSums[i][j] = the sum of (matrix[0][j], matrix[1][j], matrix[2][j],.....,matrix[i - 1][j]).

```
private int[][] colSums;
private int[][] matrix;

public NumMatrix(int[][] matrix) {
    if(   matrix           == null
       || matrix.length    == 0
       || matrix[0].length == 0   ){
        return;
     }

     this.matrix = matrix;

     int m   = matrix.length;
     int n   = matrix[0].length;
     colSums = new int[m + 1][n];
     for(int i = 1; i <= m; i++){
         for(int j = 0; j < n; j++){
             colSums[i][j] = colSums[i - 1][j] + matrix[i - 1][j];
         }
     }
}
//time complexity for the worst case scenario: O(m)
public void update(int row, int col, int val) {
    for(int i = row + 1; i < colSums.length; i++){
        colSums[i][col] = colSums[i][col] - matrix[row][col] + val;
    }

    matrix[row][col] = val;
}
//time complexity for the worst case scenario: O(n)
public int sumRegion(int row1, int col1, int row2, int col2) {
    int ret = 0;

    for(int j = col1; j <= col2; j++){
        ret += colSums[row2 + 1][j] - colSums[row1][j];
    }

    return ret;
}
```

written by larrywang2014 original link here

Answer 3

I have written both the segmentation tree based solution and the indexed tree based solution for c++.

Both are very straight-forward. Theoretically, I believe the segmentation tree based

algorithm is more efficient with large m (row number) and n(column number), but it appears for the special set of test cases, and redundancy cost by the tree implementation, the second method is using less time here.

Method 1: Segmentation-tree based solution. Essentially, it is a divide and conquer algorithm that divide the whole matrix into 4 sub-matrices recursively. It can be shown that the algorithm is O(log(mn)) per update/query.

```
class NumMatrix {
    struct TreeNode {
        int val = 0;
        TreeNode* neighbor[4] = {NULL, NULL, NULL, NULL};
        pair<int, int> leftTop = make_pair(0,0);
        pair<int, int> rightBottom = make_pair(0,0);
        TreeNode(int v):val(v){}
    };
public:
    NumMatrix(vector<vector<int>> &matrix) {
        nums = matrix;
        if (matrix.empty()) return;
        int row = matrix.size();
        if (row == 0) return;
        int col= matrix[0].size();
        root = createTree(matrix, make_pair(0,0), make_pair(row-1, col-1));
    }

    void update(int row, int col, int val) {
        int diff = val - nums[row][col];
        if (diff == 0) return;
        nums[row][col] = val;
        updateTree(row, col, diff, root);
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        int res = 0;
        if (root != NULL)
            sumRegion(row1, col1, row2, col2, root, res);
        return res;
    }

private:
    TreeNode* root = NULL;
    vector<vector<int>> nums;
    TreeNode* createTree(vector<vector<int>> &matrix, pair<int, int> start, pair<
int, int> end) {
        if (start.first > end.first || start.second > end.second)
            return NULL;
        TreeNode* cur = new TreeNode(0);
        cur->leftTop = start;
        cur->rightBottom = end;
        if (start == end) {
            cur->val = matrix[start.first][start.second];
            return cur;
        }

        int midx = ( start.first + end.first ) / 2;
```

```
        int midx = (start.first + end.first) / 2;
        int midy = (start.second + end.second) / 2;
        cur->neighbor[0] = createTree(matrix, start, make_pair(midx, midy));
        cur->neighbor[1] = createTree(matrix, make_pair(start.first, midy+1), mak
e_pair(midx, end.second));
        cur->neighbor[2] = createTree(matrix, make_pair(midx+1, start.second), ma
ke_pair(end.first, midy));
        cur->neighbor[3] = createTree(matrix, make_pair(midx+1, midy+1), end);
        for (int i = 0; i < 4; i++) {
            if (cur->neighbor[i])
                cur->val += cur->neighbor[i]->val;
        }
        return cur;
    }

    void sumRegion(int row1, int col1, int row2, int col2, TreeNode* ptr, int &res
) {
        pair<int, int> start = ptr->leftTop;
        pair<int, int> end = ptr->rightBottom;
        // determine whether there is overlapping
        int top = max(start.first, row1);
        int bottom = min(end.first, row2);
        if (bottom < top) return;
        int left = max(start.second, col1);
        int right = min(end.second, col2);
        if (left > right) return;


        if (row1 <= start.first && col1 <= start.second && row2 >= end.first && co
l2 >= end.second) {
            res += ptr->val;
            return;
        }

        for (int i = 0; i < 4; i ++)
            if (ptr->neighbor[i])
                sumRegion(row1, col1, row2, col2, ptr->neighbor[i], res);

    }


    void updateTree(int row, int col, int diff, TreeNode* ptr){
        if (row >= (ptr->leftTop).first && row <= (ptr->rightBottom).first &&
            col >= (ptr->leftTop).second && col <= (ptr->rightBottom).second)
        {
            ptr->val += diff;
            for (int i = 0; i < 4; i++)
                if (ptr->neighbor[i])
                    updateTree(row, col, diff, ptr->neighbor[i]);

        }
    }
};
```

Method 2: the 2D indexed-tree solution. It is a simple generalization of the 1D

indexed tree solution. The complexity should be O(log(m)log(n)).

```
class NumMatrix {
public:
    NumMatrix(vector<vector<int>> &matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) return;
        nrow = matrix.size();
        ncol = matrix[0].size();
        nums = matrix;
        BIT = vector<vector<int>> (nrow+1, vector<int>(ncol+1, 0));
        for (int i = 0; i < nrow; i++)
            for (int j = 0; j < ncol; j++)
                add(i, j, matrix[i][j]);

    }

    void update(int row, int col, int val) {
        int diff = val - nums[row][col];
        add(row, col,diff);
        nums[row][col] = val;
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        int regionL = 0, regionS = 0;
        int regionLeft = 0, regionTop = 0;

        regionL = region(row2, col2);

        if (row1 > 0 && col1 > 0) regionS = region(row1-1, col1-1);

        if (row1 > 0) regionTop  = region(row1-1, col2);

        if (col1 > 0) regionLeft = region(row2, col1-1);

        return regionL - regionTop - regionLeft + regionS;
    }
private:
    vector<vector<int>> nums;
    vector<vector<int>> BIT;
    int nrow = 0;
    int ncol = 0;
    void add(int row, int col, int val) {
        row++;
        col++;
        while(row <= nrow) {
            int colIdx = col;
            while(colIdx <= ncol) {
                BIT[row][colIdx] += val;
                colIdx += (colIdx & (-colIdx));
            }
            row +=  (row & (-row));
        }
    }

    int region(int row, int col) {
        row++;
```

```
        col++;
        int res = 0;
        while(row > 0) {
            int colIdx = col;
            while(colIdx > 0) {
                res += BIT[row][colIdx];
                colIdx -= (colIdx & (-colIdx));
            }
            row -= (row & (-row));
        }
        return res;
    }
};
```

written by whnzinc original link here

Best Time to Buy and Sell Stock with Cooldown(309)

Answer 1

The series of problems are typical dp. The key for dp is to find the variables to represent the states and deduce the transition function.

Of course one may come up with a O(1) space solution directly, but I think it is better to be generous when you think and be greedy when you implement.

The natural states for this problem is the 3 possible transactions : `buy`, `sell`, `rest`. Here `rest` means no transaction on that day (aka cooldown).

Then the transaction sequences can end with any of these three states.

For each of them we make an array, `buy[n]`, `sell[n]` and `rest[n]`.

`buy[i]` means before day `i` what is the maxProfit for any sequence end with `buy`.

`sell[i]` means before day `i` what is the maxProfit for any sequence end with `sell`.

`rest[i]` means before day `i` what is the maxProfit for any sequence end with `rest`.

Then we want to deduce the transition functions for `buy` `sell` and `rest`. By definition we have:

```
buy[i]  = max(rest[i-1]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
rest[i] = max(sell[i-1], buy[i-1], rest[i-1])
```

Where `price` is the price of day `i`. All of these are very straightforward. They simply represents :

```
(1) We have to `rest` before we `buy` and
(2) we have to `buy` before we `sell`
```

One tricky point is how do you make sure you `sell` before you `buy`, since from the equations it seems that `[buy, rest, buy]` is entirely possible.

Well, the answer lies within the fact that `buy[i] <= rest[i]` which means `rest[i] = max(sell[i-1], rest[i-1])`. That made sure `[buy, rest, buy]` is never occurred.

A further observation is that and `rest[i] <= sell[i]` is also true therefore

```
rest[i] = sell[i-1]
```

Substitute this in to `buy[i]` we now have 2 functions instead of 3:

```
buy[i] = max(sell[i-2]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
```

This is better than 3, but

we can do even better

Since states of day `i` relies only on `i-1` and `i-2` we can reduce the O(n) space to O(1). And here we are at our final solution:

Java

```
public int maxProfit(int[] prices) {
    int sell = 0, prev_sell = 0, buy = Integer.MIN_VALUE, prev_buy;
    for (int price : prices) {
        prev_buy = buy;
        buy = Math.max(prev_sell - price, prev_buy);
        prev_sell = sell;
        sell = Math.max(prev_buy + price, prev_sell);
    }
    return sell;
}
```

C++

```
int maxProfit(vector<int> &prices) {
    int buy(INT_MIN), sell(0), prev_sell(0), prev_buy;
    for (int price : prices) {
        prev_buy = buy;
        buy = max(prev_sell - price, buy);
        prev_sell = sell;
        sell = max(prev_buy + price, sell);
    }
    return sell;
}
```

For this problem it is ok to use `INT_MIN` as initial value, but in general we would like to avoid this. We can do the same as the following python:

Python

```
def maxProfit(self, prices):
    if len(prices) < 2:
        return 0
    sell, buy, prev_sell, prev_buy = 0, -prices[0], 0, 0
    for price in prices:
        prev_buy = buy
        buy = max(prev_sell - price, prev_buy)
        prev_sell = sell
        sell = max(prev_buy + price, prev_sell)
    return sell
```

written by dietpepsi original link here

Answer 2

Here I share my no brainer weapon when it comes to this kind of problems.

---

1. Define States

To represent the decision at index i:

- `buy[i]` : Max profit till index i. The series of transaction is ending with a **buy**.
- `sell[i]` : Max profit till index i. The series of transaction is ending with a **sell**.

To clarify:

- Till index `i`, the **buy / sell** action must happen and must be the **last action**. It may not happen at index `i`. It may happen at `i - 1, i - 2, ... 0`.
- In the end `n - 1`, return `sell[n - 1]`. Apparently we cannot finally end up with a buy. In that case, we would rather take a rest at `n - 1`.
- For special case no transaction at all, classify it as `sell[i]`, so that in the end, we can still return `sell[n - 1]`. Thanks @alex153 @kennethliaoke @anshu2.

---

2. Define Recursion

- `buy[i]` : To make a decision whether to buy at `i`, we either take a rest, by just using the old decision at `i - 1`, or sell at/before `i - 2`, then buy at `i`, We cannot sell at `i - 1`, then buy at `i`, because of **cooldown**.
- `sell[i]` : To make a decision whether to sell at `i`, we either take a rest, by just using the old decision at `i - 1`, or buy at/before `i - 1`, then sell at `i`.

So we get the following formula:

```
buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
```

---

3. Optimize to O(1) Space

DP solution only depending on `i - 1` and `i - 2` can be optimized using O(1) space.

- Let `b2, b1, b0` represent `buy[i - 2], buy[i - 1], buy[i]`
- Let `s2, s1, s0` represent `sell[i - 2], sell[i - 1], sell[i]`

Then arrays turn into Fibonacci like recursion:

```
b0 = Math.max(b1, s2 - prices[i]);
s0 = Math.max(s1, b1 + prices[i]);
```

4. Write Code in 5 Minutes

First we define the initial states at `i = 0`:

- We can buy. The max profit at `i = 0` ending with a **buy** is `-prices[0]`.
- We cannot sell. The max profit at `i = 0` ending with a **sell** is `0`.

Here is my solution. Hope it helps!

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length <= 1) return 0;

    int b0 = -prices[0], b1 = b0;
    int s0 = 0, s1 = 0, s2 = 0;

    for(int i = 1; i < prices.length; i++) {
        b0 = Math.max(b1, s2 - prices[i]);
        s0 = Math.max(s1, b1 + prices[i]);
        b1 = b0; s2 = s1; s1 = s0;
    }
    return s0;
}
```

written by yavinci original link here

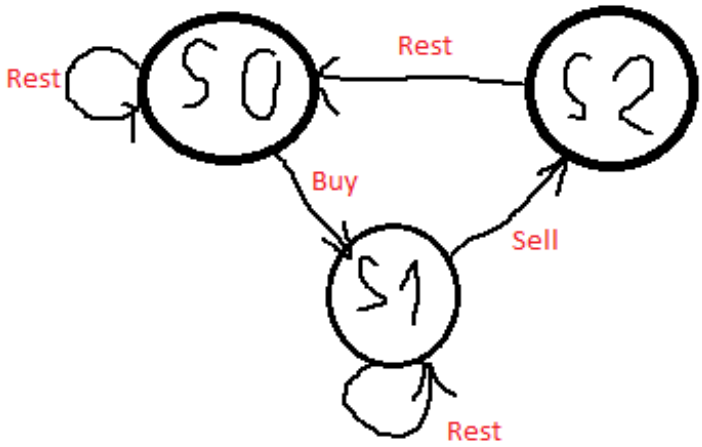
Answer 3

Hi,

I just come across this problem, and it's very frustating since I'm bad at DP.

So I just draw all the actions that can be done.

Here is the drawing (Feel like an elementary ...)



There are three states, according to the action that you can take.

Hence, from there, you can now the profit at a state at time i as:

```
s0[i] = max(s0[i - 1], s2[i - 1]); // Stay at s0, or rest from s2
s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]); // Stay at s1, or buy from s0
s2[i] = s1[i - 1] + prices[i]; // Only one way from s1
```

Then, you just find the maximum of s0[n] and s2[n], since they will be the maximum profit we need (No one can buy stock and left with more profit that sell right :))

Define base case:

```
s0[0] = 0; // At the start, you don't have any stock if you just rest
s1[0] = -prices[0]; // After buy, you should have -prices[0] profit. Be positive!
s2[0] = INT_MIN; // Lower base case
```

Here is the code :D

```
class Solution {
public:
    int maxProfit(vector<int>& prices){
        if (prices.size() <= 1) return 0;
        vector<int> s0(prices.size(), 0);
        vector<int> s1(prices.size(), 0);
        vector<int> s2(prices.size(), 0);
        s1[0] = -prices[0];
        s0[0] = 0;
        s2[0] = INT_MIN;
        for (int i = 1; i < prices.size(); i++) {
            s0[i] = max(s0[i - 1], s2[i - 1]);
            s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]);
            s2[i] = s1[i - 1] + prices[i];
        }
        return max(s0[prices.size() - 1], s2[prices.size() - 1]);
    }
};
```

written by npvinhphat original link here

Minimum Height Trees(310)

Answer 1

First let's review some statement for tree in graph theory:

> (1) A tree is an undirected graph in which any two vertices are connected by exactly one path.
>
> (2) Any connected graph who has `n` nodes with `n-1` edges is a tree.
>
> (3) The degree of a vertex of a graph is the number of edges incident to the vertex.
>
> (4) A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2.
>
> (5) A path graph is a tree with two or more vertices that is not branched at all.
>
> (6) A tree is called a rooted tree if one vertex has been designated the root.
>
> (7) The height of a rooted tree is the number of edges on the longest downward path between root and a leaf.

OK. Let's stop here and look at our problem.

Our problem want us to find the minimum height trees and return their root labels. First we can think about a simple case -- a path graph.

For a path graph of `n` nodes, find the minimum height trees is trivial. Just designate the middle point(s) as roots.

Despite its triviality, let design a algorithm to find them.

Suppose we don't know `n`, nor do we have random access of the nodes. We have to traversal. It is very easy to get the idea of two pointers. One from each end and move at the same speed. When they meet or they are one step away, (depends on the parity of `n`), we have the roots we want.

This gives us a lot of useful ideas to crack our real problem.

For a tree we can do some thing similar. We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer!

The time complexity and space complexity are both O(n).

Note that for a tree we always have `V = n`, `E = n-1`.

Java

```java
public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    if (n == 1) return Collections.singletonList(0);

    List<Set<Integer>> adj = new ArrayList<>(n);
    for (int i = 0; i < n; ++i) adj.add(new HashSet<>());
    for (int[] edge : edges) {
        adj.get(edge[0]).add(edge[1]);
        adj.get(edge[1]).add(edge[0]);
    }

    List<Integer> leaves = new ArrayList<>();
    for (int i = 0; i < n; ++i)
        if (adj.get(i).size() == 1) leaves.add(i);

    while (n > 2) {
        n -= leaves.size();
        List<Integer> newLeaves = new ArrayList<>();
        for (int i : leaves) {
            int j = adj.get(i).iterator().next();
            adj.get(j).remove(i);
            if (adj.get(j).size() == 1) newLeaves.add(j);
        }
        leaves = newLeaves;
    }
    return leaves;
}

// Runtime: 53 ms
```

Python

```
def findMinHeightTrees(self, n, edges):
    if n == 1: return [0]
    adj = [set() for _ in xrange(n)]
    for i, j in edges:
        adj[i].add(j)
        adj[j].add(i)

    leaves = [i for i in xrange(n) if len(adj[i]) == 1]

    while n > 2:
        n -= len(leaves)
        newLeaves = []
        for i in leaves:
            j = adj[i].pop()
            adj[j].remove(i)
            if len(adj[j]) == 1: newLeaves.append(j)
        leaves = newLeaves
    return leaves

# Runtime : 104ms
```

written by dietpepsi original link here

Answer 2

The basic idea is **"keep deleting leaves layer-by-layer, until reach the root."**

Specifically, first find all the leaves, then remove them. After removing, some nodes will become new leaves. So we can continue remove them. Eventually, there is only 1 or 2 nodes left. If there is only one node left, it is the root. If there are 2 nodes, either of them could be a possible root.

Time Complexity: Since each node will be removed at most once, the complexity is **O(n)**.

Thanks for pointing out any mistakes.

Updates: More precisely, if the number of nodes is V, and the number of edges is E. The space complexity is O(V+2E), for storing the whole tree. The time complexity is O(E), because we gradually remove all the neighboring information. As some friends pointing out, for a tree, if V=n, then E=n-1. Thus both time complexity and space complexity become O(n).

```
    class Solution {
    public:

        struct Node
        {
            unordered_set<int> neighbor;
            bool isLeaf()const{return neighbor.size()==1;}
        };

        vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges) {
```

```
    vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges) {

        vector<int> buffer1;
        vector<int> buffer2;
        vector<int>* pB1 = &buffer1;
        vector<int>* pB2 = &buffer2;
        if(n==1)
        {
            buffer1.push_back(0);
            return buffer1;
        }
        if(n==2)
        {
            buffer1.push_back(0);
            buffer1.push_back(1);
            return buffer1;
        }

        // build the graph
        vector<Node> nodes(n);
        for(auto p:edges)
        {
            nodes[p.first].neighbor.insert(p.second);
            nodes[p.second].neighbor.insert(p.first);
        }

        // find all leaves
        for(int i=0; i<n; ++i)
        {
            if(nodes[i].isLeaf()) pB1->push_back(i);
        }

        // remove leaves layer-by-layer
        while(1)
        {
            for(int i : *pB1)
            {
                for(auto n: nodes[i].neighbor)
                {
                    nodes[n].neighbor.erase(i);
                    if(nodes[n].isLeaf()) pB2->push_back(n);
                }
            }
            if(pB2->empty())
            {
                return *pB1;
            }
            pB1->clear();
            swap(pB1, pB2);
        }

    }
};
```

written by TTester original link here

Answer 3

The obvious method is to BFS for each node with the complexity of O(n^2) (and will get TLE).

Here is one insight for this problem: the root of MHT is the middle point of the longest path in the tree; hence there are at most two MHT roots.

How to find them? We can BFS from the bottom (leaves) to the top until the last level with <=2 nodes. To build the current level from the previous level, we can monitor the degree of each node. If the node has degree of one, it will be added to the current level. Since it only check the edges once, the complexity is O(n).

```
def findMinHeightTrees(self, n, edges):
    """
    :type n: int
    :type edges: List[List[int]]
    :rtype: List[int]
    """
    if n == 1: return [0]
    neighbors = collections.defaultdict(list)
    degrees = collections.defaultdict(int)
    for u, v in edges:
        neighbors[u].append(v)
        neighbors[v].append(u)
        degrees[u] += 1
        degrees[v] += 1

    # First find the leaves
    preLevel, unvisited = [], set(range(n))
    for i in range(n):
        if degrees[i] == 1: preLevel.append(i)

    while len(unvisited) > 2:
        thisLevel = []
        for u in preLevel:
            unvisited.remove(u)
            for v in neighbors[u]:
                if v in unvisited:
                    degrees[v] -= 1
                    if degrees[v] == 1: thisLevel += [v]
        preLevel = thisLevel

     return preLevel
```

written by todaytomoZ original link here

Sparse Matrix Multiplication(311)

Answer 1

UPDATE: Thanks to @stpeterh we have this `70ms` concise solution:

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int m = A.length, n = A[0].length, nB = B[0].length;
        int[][] C = new int[m][nB];

        for(int i = 0; i < m; i++) {
            for(int k = 0; k < n; k++) {
                if (A[i][k] != 0) {
                    for (int j = 0; j < nB; j++) {
                        if (B[k][j] != 0) C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
        return C;
    }
}
```

The followings is the original `75ms` solution:

The idea is derived from a CMU lecture.

> A sparse matrix can be represented as a sequence of rows, each of which is a sequence of (column-number, value) pairs of the nonzero values in the row.

So let's create a non-zero array for A, and do multiplication on B.

Hope it helps!

```java
public int[][] multiply(int[][] A, int[][] B) {
    int m = A.length, n = A[0].length, nB = B[0].length;
    int[][] result = new int[m][nB];

    List[] indexA = new List[m];
    for(int i = 0; i < m; i++) {
        List<Integer> numsA = new ArrayList<>();
        for(int j = 0; j < n; j++) {
            if(A[i][j] != 0){
                numsA.add(j);
                numsA.add(A[i][j]);
            }
        }
        indexA[i] = numsA;
    }

    for(int i = 0; i < m; i++) {
        List<Integer> numsA = indexA[i];
        for(int p = 0; p < numsA.size() - 1; p += 2) {
            int colA = numsA.get(p);
            int valA = numsA.get(p + 1);
            for(int j = 0; j < nB; j ++) {
                int valB = B[colA][j];
                result[i][j] += valA * valB;
            }
        }
    }

    return result;
}
```

written by yavinci original link here

Answer 2

Given A and B are sparse matrices, we could use lookup tables to speed up. At the beginining I thought two lookup tables would be necessary. After discussing with @yavinci, I think one lookup table for B would be enough. Surprisingly, it seems like detecting non-zero elements for both A and B on the fly without additional data structures provided the fastest performance on current test set.

However, I think such fastest performance could due to an imperfect test set we have for OJ right now: there are only 12 test cases. And, for an element `B[k, j]`, it would be detected for non-zero elements several times if we detecting both A and B on the fly, depending on how many `i`'s make elements `A[i, k]` non-zero. With this point, the additional data structures, like lookup tables, should save our time by focusing on only non-zero elements. If it is not, I am worried the set of OJ test cases probably is not good enough.

Anyway, I am posting my respective solutions below. Comments are welcome. Thanks @yavinci again for discussing with me.

Python solution with only one table for B (~196ms):

```
class Solution(object):
    def multiply(self, A, B):
        """
        :type A: List[List[int]]
        :type B: List[List[int]]
        :rtype: List[List[int]]
        """
        if A is None or B is None: return None
        m, n, l = len(A), len(A[0]), len(B[0])
        if len(B) != n:
            raise Exception("A's column number must be equal to B's row number.")
        C = [[0 for _ in range(l)] for _ in range(m)]
        tableB = {}
        for k, row in enumerate(B):
            tableB[k] = {}
            for j, eleB in enumerate(row):
                if eleB: tableB[k][j] = eleB
        for i, row in enumerate(A):
            for k, eleA in enumerate(row):
                if eleA:
                    for j, eleB in tableB[k].iteritems():
                        C[i][j] += eleA * eleB
        return C
```

Java solution with only one table for B (~150ms):

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        if (A == null || A[0] == null || B == null || B[0] == null) return null;
        int m = A.length, n = A[0].length, l = B[0].length;
        int[][] C = new int[m][l];
        Map<Integer, HashMap<Integer, Integer>> tableB = new HashMap<>();

        for(int k = 0; k < n; k++) {
            tableB.put(k, new HashMap<Integer, Integer>());
            for(int j = 0; j < l; j++) {
                if (B[k][j] != 0){
                    tableB.get(k).put(j, B[k][j]);
                }
            }
        }

        for(int i = 0; i < m; i++) {
            for(int k = 0; k < n; k++) {
                if (A[i][k] != 0){
                    for (Integer j: tableB.get(k).keySet()) {
                        C[i][j] += A[i][k] * tableB.get(k).get(j);
                    }
                }
            }
        }
        return C;
    }
}
```

Python solution without table (~156ms):

```
class Solution(object):
    def multiply(self, A, B):
        """
        :type A: List[List[int]]
        :type B: List[List[int]]
        :rtype: List[List[int]]
        """
        if A is None or B is None: return None
        m, n, l = len(A), len(A[0]), len(B[0])
        if len(B) != n:
            raise Exception("A's column number must be equal to B's row number.")
        C = [[0 for _ in range(l)] for _ in range(m)]
        for i, row in enumerate(A):
            for k, eleA in enumerate(row):
                if eleA:
                    for j, eleB in enumerate(B[k]):
                        if eleB: C[i][j] += eleA * eleB
        return C
```

Java solution without table (~70ms):

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int m = A.length, n = A[0].length, l = B[0].length;
        int[][] C = new int[m][l];

        for(int i = 0; i < m; i++) {
            for(int k = 0; k < n; k++) {
                if (A[i][k] != 0){
                    for (int j = 0; j < l; j++) {
                        if (B[k][j] != 0) C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
        return C;
    }
}
```

Python solution with two tables (~196ms):

```
class Solution(object):
    def multiply(self, A, B):
        """
        :type A: List[List[int]]
        :type B: List[List[int]]
        :rtype: List[List[int]]
        """
        if A is None or B is None: return None
        m, n = len(A), len(A[0])
        if len(B) != n:
            raise Exception("A's column number must be equal to B's row number.")
        l = len(B[0])
        table_A, table_B = {}, {}
        for i, row in enumerate(A):
            for j, ele in enumerate(row):
                if ele:
                    if i not in table_A: table_A[i] = {}
                    table_A[i][j] = ele
        for i, row in enumerate(B):
            for j, ele in enumerate(row):
                if ele:
                    if i not in table_B: table_B[i] = {}
                    table_B[i][j] = ele
        C = [[0 for j in range(l)] for i in range(m)]
        for i in table_A:
            for k in table_A[i]:
                if k not in table_B: continue
                for j in table_B[k]:
                    C[i][j] += table_A[i][k] * table_B[k][j]
        return C
```

Java solution with two tables (~160ms):

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        if (A == null || B == null) return null;
        if (A[0].length != B.length)
            throw new IllegalArgumentException("A's column number must be equal t
o B's row number.");
        Map<Integer, HashMap<Integer, Integer>> tableA = new HashMap<>();
        Map<Integer, HashMap<Integer, Integer>> tableB = new HashMap<>();
        int[][] C = new int[A.length][B[0].length];
        for (int i = 0; i < A.length; i++) {
            for (int j = 0; j < A[i].length; j++) {
                if (A[i][j] != 0) {
                    if(tableA.get(i) == null) tableA.put(i, new HashMap<Integer,
Integer>());
                    tableA.get(i).put(j, A[i][j]);
                }
            }
        }

        for (int i = 0; i < B.length; i++) {
            for (int j = 0; j < B[i].length; j++) {
                if (B[i][j] != 0) {
                    if(tableB.get(i) == null) tableB.put(i, new HashMap<Integer,
Integer>());
                    tableB.get(i).put(j, B[i][j]);
                }
            }
        }

        for (Integer i: tableA.keySet()) {
            for (Integer k: tableA.get(i).keySet()) {
                if (!tableB.containsKey(k)) continue;
                for (Integer j: tableB.get(k).keySet()) {
                    C[i][j] += tableA.get(i).get(k) * tableB.get(k).get(j);
                }
            }
        }
        return C;
    }
}
```

written by stpeterh original link here

Answer 3

```
class Solution {
public:
    vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B)
{
        int m = A.size(), n = A[0].size();
        vector<vector<int>> res(m, vector<int>(B[0].size(),0));

        // for(int i = 0; i < m; i++){
        //     for(int k = 0; k < n; k++){
        //          for(int j = 0; j < n; j++){
        //              res[i][j] += A[i][k] * B[k][j];
        //          }
        //     }
        // }
        // imporved on upper version, this is a math solution
        for(int i = 0; i < m; i++){
            for(int k = 0; k < n; k++){
                if(A[i][k] != 0)
                    for(int j = 0; j < B[0].size(); j++){
                        res[i][j] += A[i][k] * B[k][j];
                    }
            }
        }
        return res;
    }
};
```

written by qilong.ma.3 original link here

Burst Balloons(312)

Answer 1

Be Naive First

When I first get this problem, it is far from dynamic programming to me. I started with the most naive idea the backtracking.

We have n balloons to burst, which mean we have n steps in the game. In the i th step we have n-i balloons to burst, i = 0~n-1. Therefore we are looking at an algorithm of O(n!). Well, it is slow, probably works for n < 12 only.

Of course this is not the point to implement it. We need to identify the redundant works we did in it and try to optimize.

Well, we can find that for any balloons left the maxCoins does not depends on the balloons already bursted. This indicate that we can use memorization (top down) or dynamic programming (bottom up) for all the cases from small numbers of balloon until n balloons. How many cases are there? For k balloons there are C(n, k) cases and for each case it need to scan the k balloons to compare. The sum is quite big still. It is better than O(n!) but worse than O(2^n).

Better idea

We than think can we apply the divide and conquer technique? After all there seems to be many self similar sub problems from the previous analysis.

Well, the nature way to divide the problem is burst one balloon and separate the balloons into 2 sub sections one on the left and one one the right. However, in this problem the left and right become adjacent and have effects on the maxCoins in the future.

Then another interesting idea come up. Which is quite often seen dp problem analysis. That is reverse thinking. Like I said the coins you get for a ballon does not depend on the balloons already burst. Therefore instead of divide the problem by the first balloon to burst, we divide the problem by the last balloon to burst.

Why is that? Because only the first and last balloons we are sure of their adjacent balloons before hand!

For the first we have `nums[i-1]*nums[i]*nums[i+1]` for the last we have `nums[-1]*nums[i]*nums[n]`.

OK. Think about n balloons if i is the last one to burst, what now?

We can see that the balloons is again separated into 2 sections. But this time since the balloon i is the last balloon of all to burst, the left and right section now has well defined boundary and do not effect each other! Therefore we can do either recursive method with memoization or dp.

Final

Here comes the final solutions. Note that we put 2 balloons with 1 as boundaries and

also burst all the zero balloons in the first round since they won't give any coins. The algorithm runs in O(n^3) which can be easily seen from the 3 loops in dp solution.

Java D&C with Memoization

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] memo = new int[n][n];
    return burst(memo, nums, 0, n - 1);
}

public int burst(int[][] memo, int[] nums, int left, int right) {
    if (left + 1 == right) return 0;
    if (memo[left][right] > 0) return memo[left][right];
    int ans = 0;
    for (int i = left + 1; i < right; ++i)
        ans = Math.max(ans, nums[left] * nums[i] * nums[right]
        + burst(memo, nums, left, i) + burst(memo, nums, i, right));
    memo[left][right] = ans;
    return ans;
}
// 12 ms
```

Java DP

```
public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] dp = new int[n][n];
    for (int k = 2; k < n; ++k)
        for (int left = 0; left < n - k; ++left) {
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = Math.max(dp[left][right],
                nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][right]);
        }

    return dp[0][n - 1];
}
// 17 ms
```

C++ DP

```
int maxCoinsDP(vector<int> &iNums) {
    int nums[iNums.size() + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int dp[n][n] = {};
    for (int k = 2; k < n; ++k) {
        for (int left = 0; left < n - k; ++left)
            int right = left + k;
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = max(dp[left][right],
                     nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][rig
ht]);
        }

    return dp[0][n - 1];
}
// 16 ms
```

Python DP

```
def maxCoins(self, iNums):
    nums = [1] + [i for i in iNums if i > 0] + [1]
    n = len(nums)
    dp = [[0]*n for _ in xrange(n)]

    for k in xrange(2, n):
        for left in xrange(0, n - k):
            right = left + k
            for i in xrange(left + 1,right):
                dp[left][right] = max(dp[left][right],
                       nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][r
ight])
    return dp[0][n - 1]

# 528ms
```

written by dietpepsi original link here

Answer 2

```
int maxCoins(vector<int>& nums) {
    int N = nums.size();
    nums.insert(nums.begin(), 1);
    nums.insert(nums.end(), 1);

    // rangeValues[i][j] is the maximum # of coins that can be obtained
    // by popping balloons only in the range [i,j]
    vector<vector<int>> rangeValues(nums.size(), vector<int>(nums.size(), 0));

    // build up from shorter ranges to longer ranges
    for (int len = 1; len <= N; ++len) {
        for (int start = 1; start <= N - len + 1; ++start) {
            int end = start + len - 1;
            // calculate the max # of coins that can be obtained by
            // popping balloons only in the range [start,end].
            // consider all possible choices of final balloon to pop
            int bestCoins = 0;
            for (int final = start; final <= end; ++final) {
                int coins = rangeValues[start][final-1] + rangeValues[final+1][en
d]; // coins from popping subranges
                coins += nums[start-1] * nums[final] * nums[end+1]; // coins from
final pop
                if (coins > bestCoins) bestCoins = coins;
            }
            rangeValues[start][end] = bestCoins;
        }
    }
    return rangeValues[1][N];
}
```

written by The_Duck original link here

Answer 3

This solution is inspired by The_Duck with his C++ solution

https://leetcode.com/discuss/72186/c-dynamic-programming-o-n-3-1100-ms-with-comments

However, I would give an explanation based on my own understanding.

The basic idea is that we can find the maximal coins of a subrange by trying every possible final burst within that range. Final burst means that we should burst balloon i as the very last one and burst all the other balloons in whatever order. dp[i][j] means the maximal coins for range [i...j]. In this case, our final answer is dp[0][nums.length - 1].

When finding the maximal coins within a range [start...end], since balloon i is the last one to burst, we know that in previous steps we have already got maximal coins of range[start .. i - 1] and range[i + 1 .. start], and the last step is to burst ballon i and get the product of balloon to the left of i, balloon i, and ballon to the right of i. In this case, balloon to the left/right of i is balloon start - 1 and balloon end + 1. Why? Why not choosing other balloon in range [0...start - 1] and [end + 1...length] because the

maximal coins may need other balloon as final burst?

In my opinion, it's because this subrange will only be used by a larger range when it's trying for every possible final burst. It will be like [larger start.....start - 1, [start .. end] end + 1/ larger end], when final burst is at index start - 1, the result of this sub range will be used, and at this moment, start - 1 will be there because it's the final burst and end + 1 will also be there because is out of range. Then we can guarantee start - 1 and end + 1 will be there as adjacent balloons of balloon i for coins. That's the answer for the question in previous paragraph.

```
public class Solution {
public int maxCoins(int[] nums) {
    if (nums == null || nums.length == 0) return 0;

    int[][] dp = new int[nums.length][nums.length];
    for (int len = 1; len <= nums.length; len++) {
        for (int start = 0; start <= nums.length - len; start++) {
            int end = start + len - 1;
            for (int i = start; i <= end; i++) {
                int coins = nums[i] * getValue(nums, start - 1) * getValue(nums,
end + 1);
                coins += i != start ? dp[start][i - 1] : 0; // If not first one,
we can add subrange on its left.
                coins += i != end ? dp[i + 1][end] : 0; // If not last one, we ca
n add subrange on its right
                dp[start][end] = Math.max(dp[start][end], coins);
            }
        }
    }
    return dp[0][nums.length - 1];
}

private int getValue(int[] nums, int i) { // Deal with num[-1] and num[num.length
]
    if (i < 0 || i >= nums.length) {
        return 1;
    }
    return nums[i];
}
```

}

written by Alexpanda original link here

Super Ugly Number(313)

Answer 1

```
public int nthSuperUglyNumber(int n, int[] primes) {
    int[] ret    = new int[n];
          ret[0] = 1;

    int[] indexes  = new int[primes.length];

    for(int i = 1; i < n; i++){
        ret[i] = Integer.MAX_VALUE;

        for(int j = 0; j < primes.length; j++){
            ret[i] = Math.min(ret[i], primes[j] * ret[indexes[j]]);
        }

        for(int j = 0; j < indexes.length; j++){
            if(ret[i] == primes[j] * ret[indexes[j]]){
                indexes[j]++;
            }
        }
    }

    return ret[n - 1];
}
```

written by larrywang2014 original link here

Answer 2

Keep k pointers and update them in each iteration. Time complexity is O(kn).

```
int nthSuperUglyNumber(int n, vector<int>& primes) {
        vector<int> index(primes.size(), 0), ugly(n, INT_MAX);
        ugly[0]=1;
        for(int i=1; i<n; i++){
            for(int j=0; j<primes.size(); j++) ugly[i]=min(ugly[i],ugly[index[j]]
*primes[j]);
            for(int j=0; j<primes.size(); j++) index[j]+=(ugly[i]==ugly[index[j]]
*primes[j]);
        }
        return ugly[n-1];
}
```

written by zjh08177 original link here

Answer 3

I am using Java and failed because of this case.

500000
[7,19,29,37,41,47,53,59,61,79,83,89,101,103,109,127,131,137,139,157,167,179,181,199,
211,229,233,239,241,251]

OJ is expecting 127671181, however, the right answer should be greater than Integer.MAX_VALUE.

written by cavalier1991 original link here

Binary Tree Vertical Order Traversal(314)

Answer 1

```
def verticalOrder(self, root):
    cols = collections.defaultdict(list)
    queue = [(root, 0)]
    for node, i in queue:
        if node:
            cols[i].append(node.val)
            queue += (node.left, i - 1), (node.right, i + 1)
    return [cols[i] for i in sorted(cols)]
```

written by StefanPochmann original link here

Answer 2

class Solution { public:

```
vector<vector<int>> verticalOrder(TreeNode* root) {
    vector<vector<int>> output;
    if(!root){
        return output;
    }
    map<int, vector<int>> m;
    queue<pair<int, TreeNode*>> q;
    q.push(make_pair(0,root));
    while(!q.empty()){
        int size = q.size();
        for(int i = 0;  i < size; i++){
            TreeNode* t = q.front().second;
            int tmp = q.front().first;
            q.pop();
            m[tmp].push_back(t->val);
            if(t->left){
                q.push(make_pair(tmp - 1, t->left));
            }
            if(t->right){
                q.push(make_pair(tmp + 1, t->right));

            }
        }
    }
    for(auto& v : m){
        output.push_back(v.second);
    }
    return output;

}
```

};

written by ryderwang original link here

Answer 3

```
public class Solution {
    public List<List<Integer>> verticalOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) {
            return res;
        }
       //map's key is column, we assume the root column is zero, the left node wi
ll minus 1 ,and the right node will plus 1
        HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList
<Integer>>();
        Queue<TreeNode> queue = new LinkedList<>();
       //use a HashMap to store the TreeNode and the according cloumn value
        HashMap<TreeNode, Integer> weight = new HashMap<TreeNode, Integer>();
        queue.offer(root);
        weight.put(root, 0);
        int min = 0;
        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            int w = weight.get(node);
            if (!map.containsKey(w)) {
                map.put(w, new ArrayList<>());
            }
            map.get(w).add(node.val);
            if (node.left != null) {
                queue.add(node.left);
                weight.put(node.left, w - 1);
            }
            if (node.right != null) {
                queue.add(node.right);
                weight.put(node.right, w + 1);
            }
            //update min ,min means the minimum column value, which is the left m
ost node
            min = Math.min(min, w);
        }
        while (map.containsKey(min)) {
            res.add(map.get(min++));
        }
        return res;
    }
}
```

written by coolth original link here

Count of Smaller Numbers After Self(315)

Answer 1

The smaller numbers on the right of a number are exactly those that jump from its right to its left during a stable sort. So I do mergesort with added tracking of those right-to-left jumps.

Update, new version

```
def countSmaller(self, nums):
    def sort(enum):
        half = len(enum) / 2
        if half:
            left, right = sort(enum[:half]), sort(enum[half:])
            for i in range(len(enum))[::-1]:
                if not right or left and left[-1][1] > right[-1][1]:
                    smaller[left[-1][0]] += len(right)
                    enum[i] = left.pop()
                else:
                    enum[i] = right.pop()
        return enum
    smaller = [0] * len(nums)
    sort(list(enumerate(nums)))
    return smaller
```

Old version

```
def countSmaller(self, nums):
    def sort(enum):
        half = len(enum) / 2
        if half:
            left, right = sort(enum[:half]), sort(enum[half:])
            m, n = len(left), len(right)
            i = j = 0
            while i < m or j < n:
                if j == n or i < m and left[i][1] <= right[j][1]:
                    enum[i+j] = left[i]
                    smaller[left[i][0]] += j
                    i += 1
                else:
                    enum[i+j] = right[j]
                    j += 1
        return enum
    smaller = [0] * len(nums)
    sort(list(enumerate(nums)))
    return smaller
```

written by StefanPochmann original link here

Answer 2

Traverse from the back to the beginning of the array, maintain an sorted array of numbers have been visited. Use findIndex() to find the first element in the sorted array which is larger or equal to target number. For example, [5,2,3,6,1], when we reach 2, we have a sorted array[1,3,6], findIndex() returns 1, which is the index where 2 should be inserted and is also the number smaller than 2. Then we insert 2 into the sorted array to form [1,2,3,6].

```
public List<Integer> countSmaller(int[] nums) {
    Integer[] ans = new Integer[nums.length];
    List<Integer> sorted = new ArrayList<Integer>();
    for (int i = nums.length - 1; i >= 0; i--) {
        int index = findIndex(sorted, nums[i]);
        ans[i] = index;
        sorted.add(index, nums[i]);
    }
    return Arrays.asList(ans);
}
private int findIndex(List<Integer> sorted, int target) {
    if (sorted.size() == 0) return 0;
    int start = 0;
    int end = sorted.size() - 1;
    if (sorted.get(end) < target) return end + 1;
    if (sorted.get(start) >= target) return 0;
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (sorted.get(mid) < target) {
            start = mid + 1;
        } else {
            end = mid;
        }
    }
    if (sorted.get(start) >= target) return start;
    return end;
}
```

Due to the O(n) complexity of ArrayList insertion, the total runtime complexity is not very fast, but anyway it got AC for around 53ms.

written by mayanist original link here

Answer 3

```java
private void add(int[] bit, int i, int val) {
    for (; i < bit.length; i += 1) bit[i] += val;
}

private int query(int[] bit, int i) {
    return bit[i];
}

public List<Integer> countSmaller(int[] nums) {
    int[] tmp = nums.clone();
    Arrays.sort(tmp);
    for (int i = 0; i < nums.length; i++) {
        nums[i] = Arrays.binarySearch(tmp, nums[i]);
    }
    int[] bit = new int[nums.length];
    Integer[] ans = new Integer[nums.length];
    for (int i = nums.length - 1; i >= 0; i--) {
        ans[i] = query(bit, nums[i]);
        add(bit, nums[i]+1, 1);
    }
    return Arrays.asList(ans);
}
```

written by lasclocker2 original link here

Remove Duplicate Letters(316)

Answer 1

Given the string s, the greedy choice (i.e., the leftmost letter in the answer) is the smallest s[i], s.t. the suffix s[i ..] contains all the unique letters. (Note that, when there are more than one smallest s[i]'s, we choose the leftmost one. Why? Simply consider the example: "abcacb".)

After determining the greedy choice s[i], we get a new string s' from s by

1. removing all letters to the left of s[i],
2. removing all s[i]'s from s.

We then recursively solve the problem w.r.t. s'.

The runtime is O(26 * n) = O(n).

```
public class Solution {
    public String removeDuplicateLetters(String s) {
        int[] cnt = new int[26];
        int pos = 0; // the position for the smallest s[i]
        for (int i = 0; i < s.length(); i++) cnt[s.charAt(i) - 'a']++;
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch < s.charAt(pos)) pos = i;
            if (--cnt[ch - 'a'] == 0) break;
        }
        return s.length() == 0 ? "" : s.charAt(pos) + removeDuplicateLetters(s.su
bstring(pos + 1).replaceAll("" + s.charAt(pos), ""));
    }
}
```

written by lixx2100 original link here

Answer 2

The basic idea is to find out the smallest result letter by letter (one letter at a time). Here is the thinking process for input "cbacdcbc":

1. find out the last appeared position for each letter; c - 7 b - 6 a - 2 d - 4
2. find out the smallest index from the map in step 1 (a - 2);
3. the first letter in the final result must be the smallest letter from index 0 to index 2;
4. repeat step 2 to 3 to find out remaining letters.

- the smallest letter from index 0 to index 2: a
- the smallest letter from index 3 to index 4: c
- the smallest letter from index 4 to index 4: d
- the smallest letter from index 5 to index 6: b

so the result is "acdb"

Notes:

- after one letter is determined in step 3, it need to be removed from the "last appeared position map", and the same letter should be ignored in the following steps
- in step 3, the beginning index of the search range should be the index of previous determined letter plus one

---

```
public class Solution {

    public String removeDuplicateLetters(String s) {
        if (s == null || s.length() <= 1) return s;

        Map<Character, Integer> lastPosMap = new HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            lastPosMap.put(s.charAt(i), i);
        }

        char[] result = new char[lastPosMap.size()];
        int begin = 0, end = findMinLastPos(lastPosMap);

        for (int i = 0; i < result.length; i++) {
            char minChar = 'z' + 1;
            for (int k = begin; k <= end; k++) {
                if (lastPosMap.containsKey(s.charAt(k)) && s.charAt(k) < minChar)
{
                    minChar = s.charAt(k);
                    begin = k+1;
                }
            }

            result[i] = minChar;
            if (i == result.length-1) break;

            lastPosMap.remove(minChar);
            if (s.charAt(end) == minChar) end = findMinLastPos(lastPosMap);
        }

        return new String(result);
    }

    private int findMinLastPos(Map<Character, Integer> lastPosMap) {
        if (lastPosMap == null || lastPosMap.isEmpty()) return -1;
        int minLastPos = Integer.MAX_VALUE;
        for (int lastPos : lastPosMap.values()) {
             minLastPos = Math.min(minLastPos, lastPos);
        }
        return minLastPos;
    }

}
```

written by WHJ425 original link here

Answer 3

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        unordered_map<char, int> cnts;
        string ret;
        stack<char> stk;
        vector<bool> isVisited(26, false);
        for (char each : s) cnts[each] ++;
        for (int i = 0; i < s.size(); cnts[s[i]] --, ++ i) {
            if (isVisited[s[i] - 'a'] || (!stk.empty() && stk.top() == s[i])) con
tinue;
            while (!stk.empty() && stk.top() > s[i] && cnts[stk.top()] > 0) {
                isVisited[stk.top() - 'a'] = false;
                stk.pop();
            }
            stk.push(s[i]);
            isVisited[s[i] - 'a'] = true;
        }
        while (!stk.empty()) {
            ret.push_back(stk.top());
            stk.pop();
        }
        reverse(ret.begin(), ret.end());
        return ret;
    }
};
```

written by halibut735 original link here

Remove Duplicate Letters(316)

Answer 1

Given the string s, the greedy choice (i.e., the leftmost letter in the answer) is the smallest s[i], s.t. the suffix s[i ..] contains all the unique letters. (Note that, when there are more than one smallest s[i]'s, we choose the leftmost one. Why? Simply consider the example: "abcacb".)

After determining the greedy choice s[i], we get a new string s' from s by

1. removing all letters to the left of s[i],
2. removing all s[i]'s from s.

We then recursively solve the problem w.r.t. s'.

The runtime is O(26 * n) = O(n).

```
public class Solution {
    public String removeDuplicateLetters(String s) {
        int[] cnt = new int[26];
        int pos = 0; // the position for the smallest s[i]
        for (int i = 0; i < s.length(); i++) cnt[s.charAt(i) - 'a']++;
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch < s.charAt(pos)) pos = i;
            if (--cnt[ch - 'a'] == 0) break;
        }
        return s.length() == 0 ? "" : s.charAt(pos) + removeDuplicateLetters(s.su
bstring(pos + 1).replaceAll("" + s.charAt(pos), ""));
    }
}
```

written by lixx2100 original link here

Answer 2

The basic idea is to find out the smallest result letter by letter (one letter at a time). Here is the thinking process for input "cbacdcbc":

1. find out the last appeared position for each letter; c - 7 b - 6 a - 2 d - 4
2. find out the smallest index from the map in step 1 (a - 2);
3. the first letter in the final result must be the smallest letter from index 0 to index 2;
4. repeat step 2 to 3 to find out remaining letters.

- the smallest letter from index 0 to index 2: a
- the smallest letter from index 3 to index 4: c
- the smallest letter from index 4 to index 4: d
- the smallest letter from index 5 to index 6: b

so the result is "acdb"

Notes:

- after one letter is determined in step 3, it need to be removed from the "last appeared position map", and the same letter should be ignored in the following steps
- in step 3, the beginning index of the search range should be the index of previous determined letter plus one

---

```
public class Solution {

    public String removeDuplicateLetters(String s) {
        if (s == null || s.length() <= 1) return s;

        Map<Character, Integer> lastPosMap = new HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            lastPosMap.put(s.charAt(i), i);
        }

        char[] result = new char[lastPosMap.size()];
        int begin = 0, end = findMinLastPos(lastPosMap);

        for (int i = 0; i < result.length; i++) {
            char minChar = 'z' + 1;
            for (int k = begin; k <= end; k++) {
                if (lastPosMap.containsKey(s.charAt(k)) && s.charAt(k) < minChar)
{
                    minChar = s.charAt(k);
                    begin = k+1;
                }
            }

            result[i] = minChar;
            if (i == result.length-1) break;

            lastPosMap.remove(minChar);
            if (s.charAt(end) == minChar) end = findMinLastPos(lastPosMap);
        }

        return new String(result);
    }

    private int findMinLastPos(Map<Character, Integer> lastPosMap) {
        if (lastPosMap == null || lastPosMap.isEmpty()) return -1;
        int minLastPos = Integer.MAX_VALUE;
        for (int lastPos : lastPosMap.values()) {
             minLastPos = Math.min(minLastPos, lastPos);
        }
        return minLastPos;
    }

}
```

written by WHJ425 original link here

Answer 3

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        unordered_map<char, int> cnts;
        string ret;
        stack<char> stk;
        vector<bool> isVisited(26, false);
        for (char each : s) cnts[each] ++;
        for (int i = 0; i < s.size(); cnts[s[i]] --, ++ i) {
            if (isVisited[s[i] - 'a'] || (!stk.empty() && stk.top() == s[i])) con
tinue;
            while (!stk.empty() && stk.top() > s[i] && cnts[stk.top()] > 0) {
                isVisited[stk.top() - 'a'] = false;
                stk.pop();
            }
            stk.push(s[i]);
            isVisited[s[i] - 'a'] = true;
        }
        while (!stk.empty()) {
            ret.push_back(stk.top());
            stk.pop();
        }
        reverse(ret.begin(), ret.end());
        return ret;
    }
};
```

written by halibut735 original link here